Configuration Management in Terms of Logical Structures

Yi-Jing Lin

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-31
October 1995

Configuration Management in Terms of Logical Structures

Yi-Jing Lin
Ph.D. Dissertation

Department of Computer Science
Brown University
Providence, Rhode Island 02912

May 1996

Configuration Management in Terms of Logical Structures

by
Yi-Jing Lin
Sc.B., National Taiwan University, 1987
Sc.M., Brown University, 1992

Thesis
Submitted in partial fulfillment of the requirement for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University

May 1996

Abstract

When designing software, programmers usually think in terms of modules that are represented as functions and classes. But with existing configuration management systems, programmers have to deal with versions and configurations that are organized by files and directories. This is inconvenient and error-prone, since there is a gap between handling source code and managing configurations.

In this thesis we present a framework for programming environments that handles configuration management directly in terms of functions and classes in the source code. We define the operations required for this purpose, study their semantics, and find a general strategy to support them. We show that our framework can handle a large module as a single unit, and argue that this ability can simplify configuration management issues in software reuse, cooperative programming, and software maintenance. We also present a formal model developed to serve as the foundation of our framework and a prototype environment used to verify our ideas.

Our framework is centered around a set of objects. Each of those objects represents a function or a class in the source code and contains all the necessary data and operations to build object code and control versions of the function or class it is representing. Dependencies among functions and classes are described by links between objects, and high-level configuration management tasks are carried out by invoking the operations of individual objects. In contrast, most existing configuration management systems describe the relations between software components with separate scripts and carry out their functions with a set of tools that sit on top of a passive database or file system.

Acknowledgments

When I arrived at Brown University in 1989, I was young and naive and thought that getting a Ph.D. degree would be easy. It turned out to be one of the greatest challenges I had ever undertaken. Without help from the following people, at some point I might have given up.

My deepest thanks go to my advisor, Steve Reiss. He led me into the field of programming environments and his guidance, support, and encouragement have made this thesis possible. He taught me how to build a framework on a solid conceptual foundation and at the same time make it practical by thinking from a user's point of view. He also demonstrates by example, that it is possible to have great academic achievements and enjoy life simultaneously.

I would like to thank the other members of my thesis committee – David Notkin, Peter Wegner, and Stan Zdonik. Their comments and suggestions have substantially improved the content and presentation of this thesis. David taught me about the importance of validating a new approach. Peter helped me in organizing my ideas and presenting them more concisely. Stan helped me in understanding the database-related problems in my work. And nothing can be more delightful than hearing Stan and Peters' insightful comments about the object-oriented paradigm.

My sister Yi-Ping Cheng has been an immeasurable source of support over the past six years. She fed me, took care of me, and helped me in making some of the toughest decisions. We shared a great time in New England with my wonderful brother-in-law Tsun-Jen Cheng, my nephew Yoshang, and my niece Chia-Chia. With them, I never felt that I was actually ten thousand miles away from home.

During my darkest moments in 1990, Professor John Savage gave me great encouragement and helped me out trouble. In the past year, he again encouraged me and helped me in selling my ideas to industry. Jyh-Han Lin and Lian-Yau Tsai gave me invaluable suggestions about my

career and helped me in numerous things, both academic and personal.

Clarence Clark and Art Brooks of IBM helped me to find the topic of this research and gave me numerous comments in the early stages of this work. Hsueh-I Lu gave me invaluable encouragement and suggestions that led to the improvement of POEM. He and Yi-Jen Chiang also contributed to the theoretical part of this work. Philip Klein, Bob Zeleznik, Shuang Ji, Matthias Wloka, David Langworthy, and Lloyd Greenwald gave insightful comments on my framework. Nate Huang, Anthony Cassandra, Katrina Avery, and Mitch Cherniack did a great job on improving my writing skills. John Bazik, Max Salvas, Peter Galvin and Jeff Coady have been great sources of help and information through the years.

Nate Huang, Bob Zeleznik, Steve Reiss, Philip Hubbard, Tony Davis, Lloyd G. Greenwald, Anthony Cassandra, Jeremy Katz, Matt Corkum, Matthias Wloka, and Timothy Woodcock were my teammates in the CS softball team – The Dingers Deluxe. Winning the championship of the 1993 Brown High Intensity Intramural Softball League was a great achievement for a bunch of old nerds, and it was one of the brightest moments in my athletic career. I will never forget these “comrades” in my life.

Finally, I would like to thank my parents, Wei-Yao and Shu-Ching Lin. Without them, all these cannot be possible.

Contents

List of Figures

Chapter 1

Introduction

Although programmers usually think in terms of functions and classes when designing software, existing programming environments do not allow their users to handle *configuration management* directly with functions and classes. Configuration management activities, like *version control* and *system building*, are usually organized by files and directories. When controlling versions, programmers deal with versions of files and versions of directories. When describing the system building process, programmers specify the dependencies between source files and object files. This is inconvenient and error-prone. Many attractive properties supported by functions and classes, like abstraction and encapsulation, are unavailable in configuration management. Besides, since source code and configuration management are decomposed into two different structures, programmers have to maintain the mapping between them.

In this thesis we present a framework for programming environments that handles configuration management directly in terms of functions and classes in the source code. We define the operations required for this purpose, study their semantics, and find a general strategy to support them. We show that our framework can handle a large module as a single unit, and argue that this ability can simplify configuration management issues in software reuse, cooperative programming, and software maintenance. We also describe a prototype environment we have implemented to verify our ideas.

Our framework is based on a model with unified support for system building and version control. This model is centered around a set of objects that represent functions and classes. Most activities are carried out by the operations of these objects. In contrast, most existing configuration management systems carry out their functions with a set of tools that sit on top of a passive database or file system.

Our framework has a strong object-oriented flavor, but it is not limited to supporting object-oriented programming. Any programming language that supports separate compilation can fit into our framework. Currently, we are focusing on supporting C and C++, though our framework can also handle languages like Modula-2, Modula-3, Ada, and Pascal without major modification.

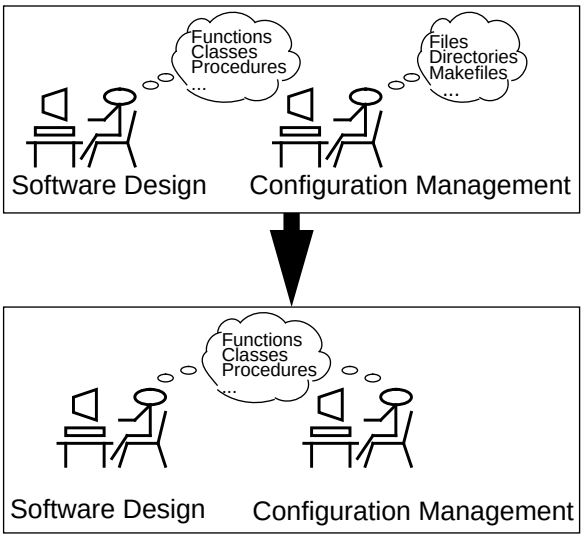


Figure 1-1: From CM with files and directories to CM with logical structures

There are limitations to our approach, though. Since the operations of our framework are defined in terms of functions and classes, they cannot properly handle documents generated before a project is decomposed into functions and classes. Our environment is designed to support configuration management in software design, implementation, and maintenance, but not activities like requirement specifications and feasibility studies. Also, our framework is aimed to support the development of new programs. It cannot manage existing code without some significant transformation.

Our framework addresses some issues that are not traditionally dealt with by configuration management systems, like software design and editing. However, it still does not constitute a full-fledged programming environment. Facilities for debugging and run-time analysis can be integrated with our framework but are not discussed in this thesis.

1.1 Configuration Management

A typical software system consists of a large and diverse collections of components. These components include source code, binary code, specifications, documentation, etc. Some of these

components are shipped to customers as the final product, and others are used only by the developers. During the development process, different versions of these components may be generated. The specific version of each component from which a given version of the final product is built is called the *configuration* of that version of the final product.

Configuration management (CM) is the discipline of controlling the evolution of software systems [50]. It is the art of identifying, organizing, and controlling changes made to a software system developed by a programming team. Its goal is to maximize productivity and quality in software development and maintenance.

Among the activities of configuration management, *system building* and *version control* play central roles. System building is the process of combining the components of a system into the final product. As illustrated in Figure 1-2, if we have a C program that consists of a main function and two modules *A* and *B*: the main function is kept in file *M.c*, the interface and implementation of *A* are kept in files *A.h* and *A.c*, and the interface and implementation of *B* are kept in files *B.h* and *B.c* respectively. In order to generate an executable program from these source files, we have to compile *A.c*, *B.c*, and *M.c* into object files and then link the object files. If any source file is modified, we have to recompile the ".c" files that are affected and link all the object files again. For example, if *A.c* and *M.c* both include the header file *A.h* and *A.h* is modified, then we have to recompile both of *A.c* and *M.c* and link the object files again.

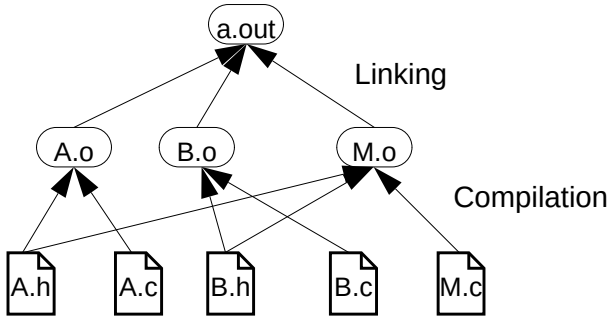


Figure 1-2: Building the executable program of a software system

The system-building process has been automated by tools like MAKE [32]. In the example above, we can describe the dependencies between source files in a *makefile*. After changes are made to source files, MAKE will decide which object files are out of date according to the makefile, and issue necessary compilation and linking commands to generate a new executable program.

Version control is the process of managing versions of software components created during the software engineering process. In the example above, we may want to create a version of *A.c* and put it aside before making some major changes, so that we can revert to that version in the future if the changes fail to work. We may also want to create versions of *A.c* that use different algorithms, deal with different operating systems, or meet different requirements.

The version-control process has been automated by tools like SCCS [72] and RCS [82], which support mechanisms to store, retrieve, and identify versions of files. Those tools also reduce their space consumption by storing only the differences between versions.

A good support for version control and system building is vital to software reuse, cooperative programming, and software maintenance. When reusing a large module, programmer should be able to select a specific version of that module and utilize the existing system building process for that module. When software is developed cooperatively by multiple programmers, programmers should be able to use different versions of a module simultaneously, so that interference and conflictions among colleague programmers can be avoided. When maintaining a software system, programmers should be able to access old versions of the system and rebuild them easily.

A broader definition of configuration management may also include *process management*, which is concerned with the enforcement of a specific software development policy. For example, a configuration management tool might insure that a change request had been approved before allowing the code actually to be changed. A configuration management tool might also generate reports about the current status of a software development project. These activities usually rely on the more basic configuration management mechanisms like system building and version control. However, we do not cover them in this thesis.

1.2 The Problem

Although existing configuration management systems support rich and diverse functions for system building and version control, they are similar in that their mechanisms are mostly defined in terms of files and directories. An obvious limitation to handling version control in terms of files and directories is that we cannot have versions of subfile entities like functions and classes. But a more important problem is that we cannot handle the versions of high-level functions and classes with simple operations.

Suppose we are developing a C program using a version control tool that supports versions of files and versions of directories, and we want to keep the current status of a function *F* in

this program. An easy solution to this problem is to create a version of the file that contains *F*. However, a version of the containing file does not guarantee a fixed status of *F*. Since *F* may use functions in other files, we have to create versions of the files that *F* depends on. Then since the files *F* depends on may depend on other files, we have to create versions of those files too. Furthermore, since different versions of *F* may use different functions and thus depend on different files, we have to create versions of the makefiles that manage *F* and the files *F* depends on. At the same time, since there are multiple versions of each source file and makefile, we have to tag them properly so we know which version of *F* maps to which versions of source files and makefiles. If we use mismatched versions of files, the result will be unpredictable.

Determining which files *F* depends on is not trivial, especially when *F* is a high-level function that directly or indirectly uses many lower-level functions. The files *F* depends on may scattered in multiple directories. If we miss a file that *F* actually depends on, then the behavior of *F* may change when we revert to an earlier version. A safe bet is to create versions for all the files *F* may *possibly* depends on. But this is usually overkill. Starting with the intention of keeping the current status of a single function, we may end up in creating a version of the whole directory or even the whole project, which contains many functions that *F* does not use.

The major problem with handling system building in terms of files is that the dependencies among files seldom match the decomposition structures of functions and classes. Understanding and maintaining the mapping between these two structures is usually difficult and tedious.

Since MAKE and its descendents are the most popular system building tools in existing environments, let us take it as an example. A typical makefile describes the dependencies between files as a three-layer directed acyclic graph (DAG), in which a set of executables depend on a set of object files and the object files depend on a set of source files. This DAG of dependencies is very shallow, and apparently cannot reflect the logical structure of a source program with more levels of decomposition. For large projects, we can combine object files into library files before linking them into the executables, and thus create more levels in this hierarchy. However, unless we create a library for all major functions and classes and use many levels of libraries of libraries, this structure still cannot faithfully reflect the true structure of the source code.

The situation is even worse when we are developing large programs that use multiple makefiles. Since makefiles lack formal communication channels like parameters and return values, the exchange of information between makefiles can only rely on implicit protocols. If we have a makefile *A* that uses the result of another makefile *B*, then the author of *A* has to know which

object files are generated by *B*. If we add, rename, or delete a file from the module managed by *B*, then makefile *A* has to be modified carefully. Since the mapping between the source code and the dependencies described in makefiles is not trivial, this modification may be difficult. Also, if we have another makefile *C* that relies on the result of makefile *A*, then the author of *C* has to know not only what *A* generates but also what *B* generates.

The major benefit of decomposing a program into modules that consist of functions and classes is the separation of concern. A function or a class encapsulates its internal complexities, so that its clients can deal only with its simplified interface. But as we have shown, this separation of concern is not available if we handle configuration management in terms of files. We have to know the internal structure of a function or a class when we are handling its versions and configurations.

1.3 A New Framework

In this research we develop a framework for programming environments that handles configuration management directly in terms of the functions and classes in source code. We achieve this by representing each function and class as an object, and describing its dependencies on other functions and classes by a small set of links. We also associate each object with the operations that build its object code and manage its versions.

In our framework, high-level configuration management tasks are carried out by the operations of individual objects. For example, if we want to build the object code of a function *F*, we can send a build message to the object that represents *F*. The build operation of that object will build the object code of *F*, and propagate the build messages along its links to all objects representing functions and classes used by *F*. In this way, we build not only the object code for *F*, but also object code for all functions and classes on which *F* depends. Similarly, we can create a version of *F* by sending a snapshot message to the object representing *F*. The snapshot message will be propagated to objects representing the functions and classes on which *F* depends, and cause a new version to be created for each of them. After automatically establishing proper links among all those newly created versions, our framework delivers a new version of *F* with a fixed behavior.

Objects in our framework contain all the necessary data and operations to build object code and control versions of the functions and classes they are representing. There is no need to write separate scripts (e.g. makefiles) to describe the system building and version control processes. We can reuse existing functions and classes by establishing links to the object representing them, or restructure a program by rearranging the links between objects that represent its functions

and classes. We can also have multiple programmers working cooperatively on a project by sharing objects representing the functions and classes they create.

1.4 Contributions

The major contribution of this research is the new framework for programming environments that handles configuration management directly in terms of functions and classes. We argue that programming environments based on this framework will be easier to use. We formalize our framework and prove the properties we claim. We also implement a prototype environment to demonstrate that this framework is practical.

Easier to Use

Our framework handles configurations in terms of functions and classes, instead of files or directories. Programmers can create and access versions of functions and classes that have smaller granularity than files. Programmers can also handle versions of high-level functions and classes as a single unit without considering their internal structures. When specifying the system building process, programmers describe the logical relations between functions and classes, instead of relations between source files and object files.

Our framework also hides the low-level operations of configuration management from programmers. In the system building process, object code is handled automatically. Programmers do not have to know what object files are generated and where they are located. When managing versions, programmers can use old versions of programs directly. There is no need to create separate directories (or workspaces) and issue check-out and check-in commands.

With better support for system building and version control, we proceed to argue that our framework can make cooperative programming, software reuse, and software maintenance easier. Programmers in a programming team can divide their work according to the logical structure of their projects, like functions and classes, instead of the physical structures like files and directories. Programmers can reuse high-level functions and classes from existing projects with simple operations. Programmers can also rearrange the relations between functions and classes without causing problems in the existing system building and version control processes. All these abilities are demonstrated with an example that contains a series of operations to be carried out in the software-development process. We also systematically compare the features of our framework and other configuration management systems.

The use of our framework is further simplified by the use of graphical user interfaces. In our prototype environment POEM, programmers can browse and modify the structure of programs through a graphical editor. The implementation of such a graphical user interface is greatly simplified by the object-oriented nature of our framework. Objects that represent functions and classes can be mapped directly to icons, and relations between them can be mapped directly to edges. Users of our graphical interface are interacting with the *real structure* of programs instead of the *artificial views*. The distance between programmers and software is shorter and there is less mapping information to be maintained by the system.

A Unified Model

Our framework is based on a unified model that integrates mechanisms for both system building and version control. It is not just a set of fragmental solutions, and it is not just a better user interface to existing tools. We show that our model can be formalized and that from the formalization we can derive several useful properties.

A major advantage of our model is that it can solve various problems more naturally. Some existing programming environments have mechanisms to solve problems like automatic object-code handling, sharing object code between programmers, visualization of programs, and customization of programming environment. But most of those solutions rely on expensive databases and algorithms. In our model, these problems can be handled with simple mechanisms.

Practical

To verify our ideas, we implemented a prototype environment called POEM that allows multiple programmers to develop software cooperatively over a local area network. It supplies a graphical user interface that enables programmers to modify the structures of programs directly, and an interpretive object-oriented language that allows programmers to program POEM itself.

POEM is an open and extensible environment. It uses existing compilers, linkers, and editors, but is not bound to specific programming languages or tools. Using the script language we provide, programmers can incorporate new tools into POEM or customize POEM to meet their special needs.

The implementation of POEM involves about 8,000 lines of C++ code on top of Motif, ObjectStore [49], YACC, and LEX, and about 1,000 lines of code in the script language of POEM. Considering the rich functionality POEM supplies, the implementation is rather easy. This is pos-

sible because our model is rather simple and because we utilize many existing tools.

In our preliminary test, the performance of POEM is more than satisfactory. We have developed several test programs and a part of POEM itself under POEM. Because most of its operations are described in a interpretive language, POEM is slow on basic operations like deciding whether a piece of source code has been modified. But since programmers can carry out complex tasks with fewer operations, POEM may save time and effort for programmers.

1.5 Thesis Outline

In the next chapter, we review previous work in related disciplines. We discuss software configuration management systems, programming environments using object-oriented technologies, cooperative programming techniques, generic environments, and efforts to organize software artifacts in a programming environment.

In Chapter 3, we present a tour through the interactive environment of our prototype system, POEM. Here we discuss not how tasks are carried out internally, but rather how programmers use the environment to build and manage software systems. We use an example that contains a series of operations to be carried out in the process of developing a software system, and compare POEM with other existing configuration management tools.

Chapter 4 describes the basic mechanisms of the POEM model and discusses how these mechanisms can help in software reuse, cooperative programming, and software maintenance. In Chapter 5, we gives a formal description of our model and derive some useful properties. Those properties help us to understand what may happen and what may not in our framework.

Chapter 6 describes the implementation issues. Instead of describing all aspects of the implementation of POEM, we focus on the general structure, the script language used to define classes of objects in POEM, and a new technique to improve the performance of version control systems.

We conclude this thesis in Chapter 7 by summarizing our contributions and listing some future directions of this research. We also discuss the limitation and drawbacks of our approach and propose some possible solutions.

Chapter 2

Related Work

This chapter reviews previous and current work that has influenced our research. Since the major problems addressed by this thesis are in the areas of configuration management, cooperative programming, generic environments, and organization of software artifacts, we discuss the important work in these areas. Additionally, since our approach is centered around objects, we also discuss other efforts to utilize object-oriented technology in programming environments.

Some of the systems discussed here are very large and have rich sets of features covering different research areas. Because they are difficult to compare as a whole, we discuss their individual features in different sections instead of giving general descriptions.

2.1 Configuration Management Systems

MAKE

MAKE [32] and its descendents are the most commonly used system building tools. MAKE keeps track of the relationships among files of a program, and issues the commands needed to make the files consistent after changes are made. A program is considered inconsistent if some of its files are older than the files they depend on. MAKE detects this kind of inconsistency and makes sure that only the necessary commands are issued when bringing a program up to date.

MAKE has rather weak support for the modulization of makefiles. There is no formal channel of communication between makefiles. If we want to use multiple makefiles in a large project, then the passing of arguments has to rely on implicit protocols. The lack of formal arguments also makes the reuse of makefiles more difficult.

MAKE is a file-based system. Users must describe the dependencies between files in a *makefile*. A makefile usually describes the structure of a program in a three-level directed acyclic

graph (DAG). Executable files in the top level depend on the object files in the middle level, and the object files depend on the source files in the bottom level. Since this three-level structure seldom reflect the logical structure of programs, maintaining the integrity between source code and makefiles is not trivial. To avoid this problem, users can create more hierarchical makefiles by combining object files into libraries files before linking them into executables. However, handling libraries in MAKE is cumbersome, and mimicking the structure of an arbitrary program using flat-structured libraries is difficult.

There are several tools that simplify the writing of makefiles. The *implicit rules* of MAKE issue appropriate compilation commands according to the file name extension of source files. IMAKE [16] allows users to specify system models at a higher level by supporting a set of macros that expand into rules for makefiles. Makedepend [17] parses source programs to generate dependencies used by MAKE. However, these tools simplify only the specification of compilation rules, not that of the linking rules. Users still have to specify which object files and which libraries should be linked into a specific executable file. For large systems that contain hundreds of files in multiple directories, this task is not trivial. Another drawback of makedepend is its performance. Users of makedepend have to re-parse all the source files every time they add or remove a "#include" directive, which is quite expensive for large systems. Besides, since the output of makedepend contains dependencies on system header files that are seldom modified, system building processes based on the output of makedepend are usually slower than those based on hand-written makefiles.

SCCS

Source Code Control System (SCCS) [72], one of the earliest tools for version control, maintains different versions of files without unnecessary code duplication. It controls system updates by ensuring that no part of the system can be updated by more than one programmer at any one time. It also records information about who made the change, when and where it was made, and why.

RCS

The Revision Control System (RCS) [82] automates the storing, retrieval, logging and identification of revisions and provides selection mechanisms for composing configurations. RCS retains the best features of SCCS, but offers a simpler user interface, flexible selection rules, inte-

gration with MAKE and improved identification.

DSEE and ClearCase

Apollo's Domain Software Engineering Environment (DSEE) [51] provides source code control, system building management, release control, advice management, task management, and user-defined dependency tracking with automatic notification. Users of DSEE specify *configuration threads* that select components to be used in building a product. The selection can be based on certain characteristics, equivalencies, or compatibilities. DSEE's centralized *derived object pool* holds binary file that can be reused between users. When building a system, it searches the derived object pool for binary files with exact match before recompiling the source. The major problem with DSEE is that it is tightly bound to the Apollo DOMAIN architecture, it relies on special supports from DOMAIN and cannot be ported to other operating systems.

DSEE uses *system models* to describe the relationships among the components of a system. Since the system model of a large system can become very complex, DSEE provides a way of structuring the system model as a hierarchy of nested blocks. In this respect, the difference between a Makefile and a DSEE system model is analogous to the difference between FORTRAN and a block-structured language like Pascal.

ClearCase [3][4][5] from Atria is a logical successor of DSEE and was developed by the original developers of DSEE. It supports most of the key facilities of DSEE, but does not rely on special support from the underlying operating system. It runs on multiple operating systems, including UNIX and Windows NT, and supports version control for directories. ClearCase also keeps track of the versions of compiler and run-time libraries used to build a program.

A major difference between ClearCase and DSEE is that ClearCase uses conventional makefiles instead of proprietary system models. Doing so increases ClearCase's compatibility with other existing tools, but also makes the management of system building less organized.

SHAPE

SHAPE [58] integrates a dedicated version control system and a enhanced MAKE program on the basis of a common object model. Its object model comprises multiple versions of software objects as well as conventional file system objects. The SHAPE approach allows a sufficiently integrated tool system for engineering software configuration management while retaining the flexibility of the basic toolbox philosophy. It permits the use of 'off-the-shelf' tools,

e.g. editors or compilers.

The philosophy of SHAPE is similar to DSEE. Its *configuration selection rules* are similar to the configuration threads in DSEE. But SHAPE does not need special support from the underlying operating system. It also augments the version selection mechanisms with a built-in general *version status model*. Each version of file in SHAPE is tagged as *busy, saved, proposed, published, accessed,* or *frozen*. These tags are not only intended to say something about the relative quality of a particular object version, but also reflect whether an object version is in a developer's private experimentation domain or in the project's official domain.

Adele

Adele [8][9][26][27][28][29][30] is a configuration management system from the University of Grenoble. Its features include version control, system modeling, automatic system building, workspace, and process support. Adele is intended to serve as the kernel of a software engineering environment.

The database of Adele is based on the entity-relationship model. Basically, a system is modeled as a set of *interfaces* and *realizations*. An interface *is_realized* by a set of realizations, and a realization *depends_on* other interfaces. Interfaces, realizations, is_realized relations and depends_on relations together form an acyclic graph. When building a specific version of the system, only one realization is selected for each interface.

NSE and TeamWare

The Network Software Environment (NSE) [21] from Sun Soft uses a database that manages the UNIX directory structure and derived files in addition to the source code. NSE supports cooperative programming via *environments* that represent workspaces. Workspaces support nested transactions with a protocol for merging and updating files between a child and a parent workspace.

TeamWare [78] is the successor of NSE. It includes graphical tools for version control (VerTool), workspace and directory management (CodeMgrTool), source file configuration archiving (FreezePtTool), project build acceleration (PMake), and automatic file merging (FileMerge). While NSE is based on a proprietary file system, TeamWare relies on standard UNIX utilities and services like SCCS and NFS. Development teams that are already using SCCS can quickly adopt TeamWare.

NSE and TeamWare are based on a copy-modify-merge philosophy. Programmer do not lock files to prevent modification by other programmers. Instead, they proceed with their concurrent modifications and merge their results later. A problem with this approach is the duplication of code, source files and object files are duplicated in multiple workspaces even if they are not modified. Another concern is the merging of files. Since file merging is a difficult problem that cannot be automated easily, programmers' intervention is required in most cases.

CVS

CVS [12] is a front end to RCS. It extends the notion of revision control from a collection of files in a single directory to a hierarchical collection of directions consisting of revision controlled files. CVS keeps a single copy of the master sources called the source *repository.* Individual programmers check out copies of the sources to their own workspace and make modifications there. Unlike SCCS and RCS, CVS does not use locks to prevent concurrent editing of the same file. Instead, it uses a copy-modify-merge philosophy similar to that of NSE and TeamWare.

Like NSE and TeamWare, the major problems of CVS are its space consumption and its dependence on merging mechanisms. CVS does not use any symbolic or hard links to facilitate file sharing between programmers. If a program is being developed by *N* programmers, *N* copies of its source and object code exist in the file system. About file merging, the author of CVS argues that simultaneous editing of the same file is rare in practice and that most merging conflictions can be solved easily.

Aide-de-Camp

Aide-de-Camp (ADC) [75] provides an entity-relationship database to store file attributes and relationship among files, and a set of commands to manage logical changes made on system configurations. Its operations are centered around the concept of *change set*, which is a logical change consisting of modifications to a set of files in a software system. Aide-de-Camp differs from other configuration management systems in that it focuses on the differences between file versions instead of the files versions themselves.

Users of Aide-de-Camp start a logical change to software systems by creating a new change set. Aide-de-Camp sets up a local workspace called *cleanroom* for each change set, so that users working on different change sets will not interfere one another. When the modification is finished, all files associated with a change set must be checked in at the same time.

Vesta

The Vesta configuration management system [14][20][39][52] aims to increase the reusability of system models. It uses a functional language to describe system models, and emphasizes the modulization and parameterization of system models. However, since the nature of system building requires lots of parameters for each system building function, and since it is impractical to supply all these parameters on each call to these functions, VESTA has to introduce a complex binding mechanism to manage the parameters of system building functions.

Automatic Handling of Derived Objects

Several systems are designed to manage system building at a higher level. Cedar [80] and DSEE [51] use *source-oriented system models*. Derived objects are not directly managed by programmers, but occasionally users still need to address them indirectly via some functions. In Vesta, derived objects are passed along as the result of building functions and are not directly managed by users. CaseWare [19] uses an object-oriented build facility that stores the information about how to build a particular type of object in the object type definition itself, rather than in an external makefile. Also, the build context information is placed in objects.

Ada Programming Support Environments

In Ada [7], since packages and subprograms are units of source code as well as units for compilation, the gap between handling system building and managing source code is smaller than in other languages. In addition, an Ada environment is responsible for deciding which units need to be recompiled based on the logical relations between them [18]. However, Ada organizes compiled packages in libraries, which have flat structures and thus cannot directly capture the relationship between packages and subprograms. Ada also needs additional mechanisms to handle a mapping between versions of libraries and their elements. CMVC [61] presents one such mechanism.

Version Control in Terms of Objects

Version control in terms of objects is studied in the area of software development environments as well as in the area of computer-aided design (CAD). Zdonik describes an object-oriented database system that includes a built-in version control mechanism [92]. PCTE [13][81] supports an versioned object base for constructing software engineering environments. Katz surveys version modeling in engineering databases and proposes a unified framework [45]. The configuration

thread of DSEE allows programmers to choose versions by specifying version selection rules. SHAPE and ClearCase also supply similar functionality.

Configuration Management Models and Concepts

Feiler classified the models of configuration management systems into checkout/checkin model, composition model, long transaction model, and change set model [31]. According to this classification, SCCS, RCS, and MAKE represent the traditional *checkout/checkin model*, in which repository management and system building are quite independent. DSEE, SHAPE, and Adele use the *composition model*. Developers in these systems describe the components of a system by a file called *system model*, and select the desired version for each component by version selection rules. NSE and TeamWare use the *long transaction model*. They support the evolution of systems as a series of atomic changes. Developers operate on configurations rather than individual components, and a change is performed in a transaction. Aide-de-Camp (ADC) uses the *change set model*. It groups related modifications to different components as a logical change.

Dart analyzed and compared the concepts and functionality supported by different configuration management systems [23]. She observed that most concepts in configuration management can be seen as extensions to, or generalizations of, other concepts. She also noticed that it is hard to extract clear concepts from existing configuration management systems because there is no commonality in terminology.

JASON

Wiebe [83] proposed a generic SCM approach that lets users make decisions on the basic software configuration management (SCM) issues, and thereby tailor the behavior of the generic SCM system. A mathematical model of the generic SCM approach is supplied, and a particular generic SCM system called JASON is implemented.

The major advantage of JASON is its flexibility. Users can specify classes of objects and configurations, families of versions, configuration consistency constraints and build plans. These specifications act as schemas for software object bases that contain the actual software object objects and configurations. Thus JASON users control the way in which JASON organizes and manages software objects.

Although intended to be generic, the JASON model is somewhat biased toward the compositional model. JASON directly supports mechanisms like version families, system templates

and selection rules, which are fundamental in the compositional model but not in the long-transaction model and change set model. JASON does not directly support workspaces and long transactions, which are the basic mechanisms for the long transaction model, or change set, which is the basic mechanism for the change set model.

2.2 Cooperative Programming

Cooperative programming has been recognized as an important issue in software engineering, but its nature is not yet well understood. MARVEL [6] sets the goal of supporting cooperative programming as to reduce the interference between programmers. It uses forward and backward chaining of rules to facilitate communication between programmers, and uses a two-phase locking to detect interference. When detected, interference is resolved according to some consistency constraints defined in the process model. ConversationBuilder [44] supports collaborative software development with a hypertext system. Coordination is supported according to protocols specified by users.

A good version control system is a simple, inexpensive but effective tool to help cooperative programming. By using stable versions created by other programmers, a programmer can avoid interference from other programmers. DSEE uses *configuration threads* to help programmers select versions of software objects in a cooperative programming environment. ClearCase and SHAPE also rely on this mechanism.

2.3 Organization of Software Artifacts

The lack of associations between related software artifacts in traditional environments has long been recognized. A popular approach to this problem is to use hypertext. ISEA [15] automatically creates hyperlinks between program analysis data and hypertext documentation. Kiosk [22] uses hypertext in selecting reusable software components from software libraries. HyperWeb [33] uses hypermedia to help programmers on understanding and maintaining software. ConversationBuilder [44] also uses hypertext to describe shared objects in collaborative software development. The hypertext approach has the flexibility to establish links between any two arbitrary software artifacts. However, too many unrestricted links in a software development environment are just like too many goto statements in a program; they cause navigation problems and increase cognitive overhead. Also, setting hyperlinks directly between fine-grained entities conflicts with the principle of encapsulation. Modifying one part of a program may invalidate hyperlinks from other

parts.

FIELD [59][69][70] supports tools that visualize the call graphs, class diagrams, and make dependencies of a program. It also lets programmers navigate in a program according to the logical relations among modules. Users can move to a certain module by clicking the mouse on the corresponding icon. However, they cannot edit the structure of a program by editing these diagrams directly. The editor of SMARTsystem [68] supports a *filer* mechanism that allows users to hide irrelevant code when editing source files.

Donald Knuth proposed the *literate programming* approach to writing more easily understood and maintained programs, and developed an environment called WEB to support his view [11][48]. In this approach, software is organized for humans, not the computers. A source file in WEB contains both code in a programming language and documentation in a text-formatting language, and is fed to different translators to generate executable code for computer and documents for humans. Basically, documents and code are decomposed in parallel. A problem with this approach is that while the generated documents are more readable, authoring the source files is harder. Also, it is difficult to write programs in this style when we do not have full knowledge about the problem being solved and have to try different algorithms.

2.4 Generic Environments

On the basis of the observation that different programming projects have different requirements on the environment, people have been seeking methods to generate automatically specific environments from generic ones. Gandalf [38] facilitates semiautomatic generation of a set of related software-development environments by supplying a generation environment to the designers. An editor generator takes a description of the language that is to be manipulated and produces as output a syntax-sensitive editor for the language. JASON, as discussed earlier, lets users tailor the behavior of a generic configuration management system. Garlan [35] proposed a transformational approach to generating application-specific environments.

Process-centered environments support facilities to generate software environments with highly customized software process policies. Osterweil argues that software processes are software too, and software environments is best viewed as a vehicle for the specification, compilation, and execution of process programs [66]. Several process-centered environments, including Arcadia [79], MARVEL [47], OZ [10], Adele2 [9], and Darwin [60], are built to validate this view.

Our framework shares interests with these process-centered environments in that we want

to make programming environments themselves programmable. But while most process-centered environments use imperative or rule-based process programs, our approach has a strong object-oriented flavor. Also, process-centered environments are designed to cover all stages of the software engineering life cycle. In contrast, our framework is focused on the latter stages of software development. In addition, our framework does not address problems like quality control and project scheduling.

2.5 Separation of Interface and Implementation

Our framework requires that a class or function be separated into an interface and an implementation. Separating interface and implementation is commonly accepted as good programming style. Programming languages like Ada, Modula-2, and Modula-3 directly support this idea at the language level. Parnas suggested using information hiding as the basic criterion to decompose systems into modules, so that software will be more flexible and more comprehensible [67]. Gandalf [38], the first project to apply this notion in programming environments, proposed a module concept in which a set of versions implement a single interface. Adele [27] describes the relation between interfaces and implementations as an AND/OR graph. ISTAR [24] uses a *contractual approach*, in which every activity in the software process has the character of a contract. It also defines formal channels of communication between tasks.

2.6 Programming Environments and Object-Oriented Technologies

Our framework is based on the object-oriented paradigm [85]. Because of the success and popularity of the object-oriented technology, people have been trying to apply it in building software development environments. RPDE [40][65] uses objects to implement internal components of a programming environment, especially the abstract syntax tree. PCTE [34] supports an object base as the basis for software development environments. Render and Campbell describe an object-oriented model of configuration management that represents most software entities as objects [71]. But objects in that model are rather passive and low-level. Most tools are built on top of objects instead of implemented as operations of objects. The Attributed File System of SHAPE enable users to associate attributes with files. Several advanced environments, including Adele [9][29], MARVEL [47], Darwin [60], and Arcadia [79], use objects for data integration. Cactis [43] is an object-oriented database designed to support software environments.

Rumbaugh [73] proposed a framework to control the propagation of operations between

objects. The propagation policy is based on attributes associated with relations. Our framework also relies on the propagation of operations to handle composite objects. But instead of using the full power of that rather complicated model, we found that treating all operations equally and using workareas to control the propagation is sufficient to meet our requirements.

2.7 Program Restructuring

In order to reduce the cost of software maintenance caused by the need to restructure software systems with degrading structure, people have been trying to build tools that automatically restructure software systems while preserving their semantics. Griswold and Notkin [36][37] presented a set of automatable transformations and studied their impact on structures. They also described a model to building meaning-preserving restructuring transformations and a tool for restructuring Scheme programs. Opdyke [64] defines a set of restructuring operations for object-oriented programs.

Our framework shares interest with these tools in that we want to make software restructuring easier, but we do not try to automate the restructuring of the source code itself. Instead, we are interested in simplifying the configuration management issues in software restructuring: whether the restructuring is carried out manually or automatically by a tool, the system building and version control processes should be restructured accordingly. In this respect, our framework and these tools can complement each other in simplifying the general problem of software restructuring.

Chapter 3

A Tour of POEM

This chapter introduces the POEM programming environment. We describe the procedures to create programs, to build executable code, and to create and manage versions of software. We also discuss the scenario to reuse a module, to develop programs cooperatively, and to maintain existing programs.

The purpose of this chapter is to give readers the general idea of our framework. We attempt to give readers the "feel" of POEM by simulating a live demonstration. We describe the features of POEM from a user's point of view, instead of discussing the internal mechanisms. A detailed and rigorous discussion of our framework is given in the following chapters. Also, since POEM has a rich set of functionality, it is not practical to describe them all in this chapter. Some more advanced features are mentioned in the following chapters, and some less important features are omitted entirely.

In order to make clear the differences between our framework and existing approaches, we compare POEM with other configuration management systems in this tour. We choose MAKE [32] as the basis for comparing system building features, because it is the most commonly used system building tool in current programming environments. Moreover, most more advanced configuration management systems, like ClearCase [4][5], TeamWare [78] and SHAPE [58], also base their system building mechanism on MAKE-like tools. For similar reasons, we choose RCS [82] and SCCS [72] as the basis for comparing version control features. But as there is more variety in version control mechanisms, we also discuss other advanced systems when they support important features.

The current implementation of POEM runs on top of the X window system. We assume that readers are familiar with WIMP (Window-Icon-Menu-Pointer) user interfaces, but specific

knowledge about the X window system is not required.

3.1 Overview

Figure 3-1 shows a typical screen of POEM. There are two *workarea windows* on the left hand side and three editors on the right hand side. The small boxes in the workarea windows are *software units*, each of which represents a function or a class. If we click the mouse button on a software unit, POEM loads its *interface*, *implementation*, and *documentation* into the corresponding editors on the right hand side. The interface specifies the resources provided by a software unit. It is similar to a ".h" file in traditional C/C++ programming environments. The implementation contains the code that realizes what is specified in its corresponding interface. It is similar to a ".c" file in traditional C/C++ programming environments. The documentation is the textual description of a software unit. Programmers can put anything they think appropriate there.

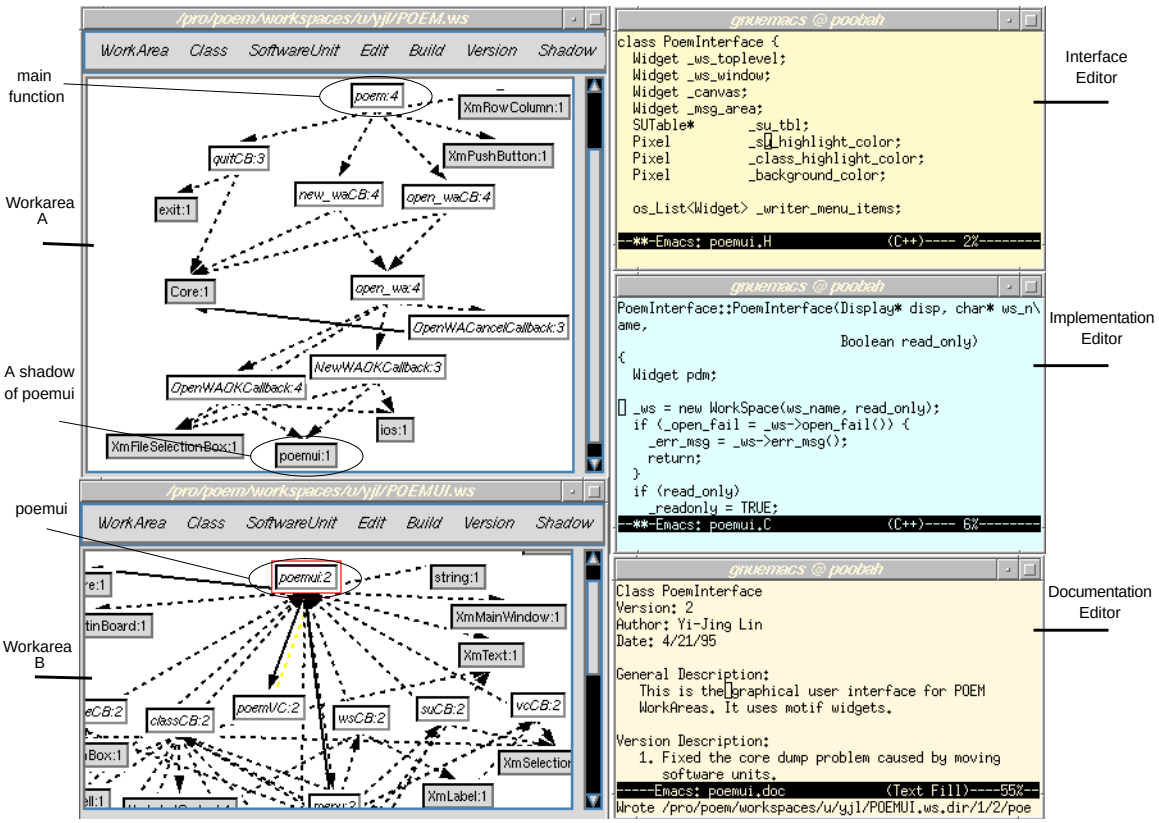


Figure 3-1: A typical POEM screen

The dotted arrows between software units are *uses_interface* links. A *uses_interface* link from software unit *A* to software unit *B* means that the implementation of *A* depends on the inter-

face of *B*. It is analogous to including a ".h" file in a ".c" file in traditional environments. The solid arrows between software units are *t_uses_interface* links. A *t_uses_interface* links from software unit *A* to software unit *B* means that the interface of *A* depends on the interface of *B*. It is analogous to including a ".h" file in another ".h" file in traditional environments.

A workarea window supports the functionality of a typical graphical editor. Programmers can create new software unit boxes, move them around, and draw links between them interactively.

POEM uses existing C and C++ compilers in the UNIX operating system as its back end. It does not require modification to the syntax of C and C++. But since the *uses_interface* links and *t_uses_interface* links already describe the exchanging of definitions between software units, the "#include" directives should not be used when the corresponding links exist. Using links and "#include" directives simultaneously is redundant and may cause integrity problems. The current implementation of POEM does not detect redundant "#include" directives, but users should be able to add this feature using the script language of POEM, which is discussed in Chapter 6.

3.2 System Building

The example in Figure 3-1 is actually the implementing of POEM under POEM itself. The main function of POEM is represented as a software unit located near the top of workarea *A*. We call it the *root software unit* of our program. To build the executable code of POEM, we select this software unit by clicking the mouse button on it and choose *BuildExe* under the *Build* menu. POEM tries to compile all the software units whose object code is outdated, and link their object code into a executable program. If there is some error in the program, POEM shows the error message in another window and indicates which software unit caused the problem. Otherwise, we can run the program by choosing *Run* under the *Build* menu.

What POEM does about system building is similar to what MAKE does: it checks the dependencies between components of a program and compiles only the outdated ones. But there are three differences. First, we do not have to write a separate makefile. The software units and their links in the workarea windows work like a graphical makefile and give POEM all the necessary dependency information. Second, we do not have to know the locations of the object code. The handling of object code is hidden from users. Third, the relations we specified in the workareas are the logical relations between functions and classes, not the compilation and linking dependencies between files. By looking at the graph shown in workarea windows, we can easily get the

big picture of the general structure of our program.

If we want to build the same example using MAKE, several makefiles are required. We list those makefiles in Appendix A.

Sometimes we may want to change the compilation options of a class or a function. For example, we may want to turn on the debugging option of a function, so that the compiler will generate extra debugging information for it. To do so, we select the software unit that correspond to the function and choose *Set_Build_Options* under the *Build* menu. A dialog box will appear as shown in Figure 3-2.

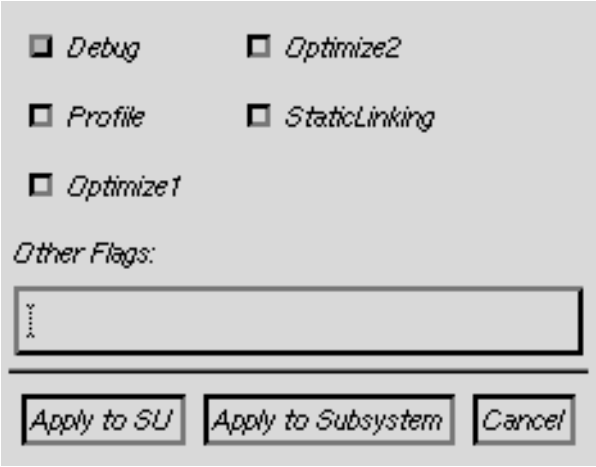


Figure 3-2: *Set_Build_Flag* dialog

After checking the desired options, we can choose either *Apply_to_SU* or *Apply_to_Subsystem* at the bottom of this dialog box. If we choose *Apply_to_SU*, then the new setting affects only the selected software unit. If we choose *Apply_to_Subsystem*, then the new setting will be applied to all the software units that can be reached from the selected software unit via links. For example, if the root software unit *poem* is selected, then the new setting is applied to the whole program.

Compared with MAKE and other existing system building tools, setting build options in POEM is more convenient for two reasons. First, we do not have to remember the flags for every option. The options listed in the dialog are comprehensible *logical options*, not physical flags like "-g", "-xpg.", and "-xO1." We can understand what they mean without looking up manuals of the compilers.

Second, we can set the parameters in terms of functions and classes. For example, if we

know that the bottleneck of our program happens in a certain function or class, we can turn on the optimization flag for that function or class only. We can thus speed up our program without having to spend extra time optimizing other non-critical functions and classes. In contrast, build parameters in MAKE can only be set in terms of makefiles or make rules. Programmers using MAKE can change the global variables that define the default build parameters in a makefile. But in this case all the source files managed by that makefile are affected and MAKE will try to recompile all of them. Alternatively, programmers can set the options for individual make rules, but in this case we have to use explicit rules instead of implicit ones, and doing so for each target in a makefile is laborious and error-prone.

In POEM, if we want to change the compiler being used by our program, we can select the root software unit of our program and choose *Set_Compiler* under the *Build* menu. Another dialog box will appear, as in Figure 3-3:

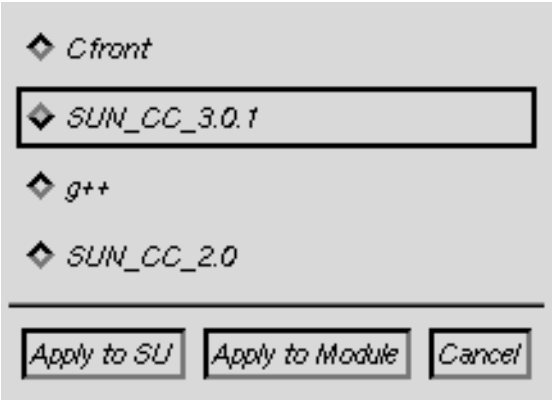


Figure 3-3: *Set_Compiler* dialog

We can select the compiler we want to use in this dialog, and choose *Apply_to_Subsystem* to apply the new setting to the whole program. We can also choose *Apply_to_SU* in this dialog to set compiler for individual software units, provided that the object code generated by different compilers is compatible,

After changing the compiler, we need not go back and set the build options again, even if the new compiler uses different flags for the same option. For example, while SUN compilers use “-xpg” to turn on the profiling option, GNU compilers use “-pg.” If we set the profiling flag and then switch from the SUN compiler to the GNU compiler, there is no need to set the profiling option again. What we set in the *Set_Build_Flags* dialog are logical options; POEM maps them to the actual flags before generating the compilation commands.

In contrast, if we want to switch compilers under MAKE, we have to compare the documentation of both compilers, find their differences, and then scan all the makefiles to locate the necessary changes. This procedure is very time-consuming, especially when we are using explicit make rules.

3.3 Version Control

At the top of workarea B in Figure 3-1, we see a software unit labeled *poemui*. This software unit is the C++ class that implements the graphical user interface of POEM. To keep its current status, we can select it and choose *Snapshot* under the *Version* menu. The result of this operation is shown in Figure 3-4.[1] In this figure, the black boxes are *fixed versions* of software units that are created by the *snapshot* operation. POEM creates a fixed version for all the modified software units that can be reached from *poemui* and links them together in the same ways as in the current working version. POEM also makes sure that if a software unit is fixed, then all the software units it can reach via links are also fixed.

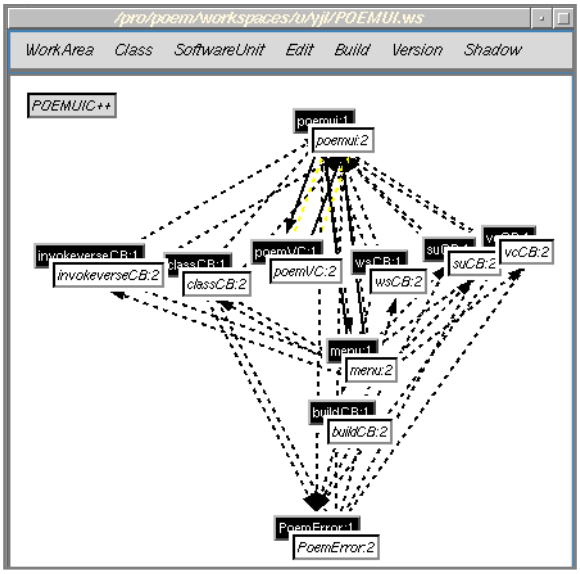


Figure 3-4: Making a snapshot of a class

Fixed software units are read-only, but they can be used in the same way as other software

1. The workarea in Figure 3-4 looks somewhat different from workarea B in Figure 3-1, because we have hidden the gray boxes shown in Figure 3-1. The meaning of those gray boxes is discussed in Section 3.4.

units. If we click on a fixed software unit, POEM brings up its interface, implementation, and documentation in the editors. Other software units can also use a fixed software unit by pointing links to it. There is no need for check-in or check-out operations. POEM uses RCS internally to save space on fixed software units, but most of those actions are hidden from users.

POEM allows multiple versions of a software unit to reside in the same workarea. There is no need to create a separate directory or workspace for each version of a project, since operations on one version of a software unit do not interfere with other versions. In contrast, keeping multiple versions around in existing configuration management systems is usually more difficult. In RCS and SCCS, we have to create separate directories for each checked out-version. In ClearCase and TeamWare, we have to create separate workspaces or views. To access different versions we have to change directories or switch between workspaces or views.

Version selection in POEM is carried out by setting the destination of links. For example, if we create a software unit called *OpenWA* and want to use the old version of *poemui* in this software unit, then we can establish a link from *OpenWA* to the old version of *poemui, as* illustrated in Figure 3-5. If we want to use the newer version of *poemui* instead, we can simply make the link point to the newer version of *poemui*.

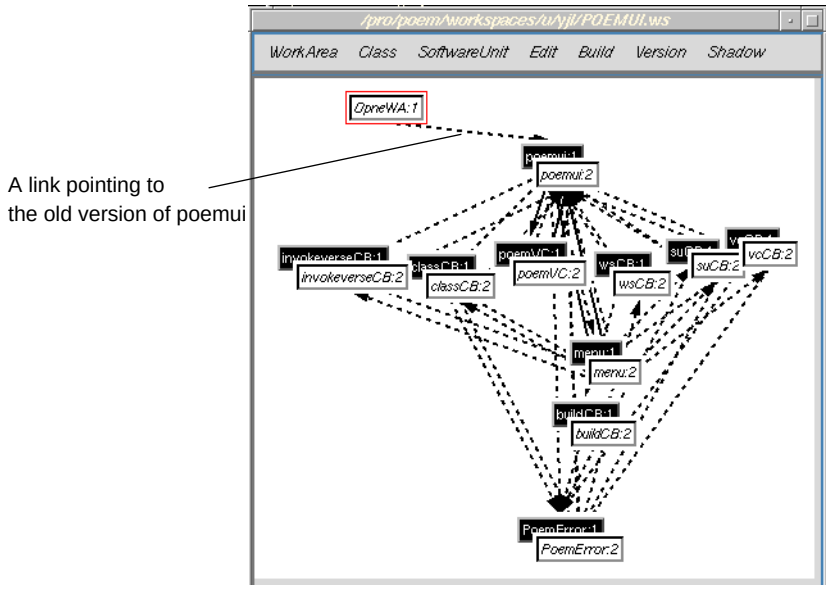


Figure 3-5: Selecting versions of a subsystem

When choosing between versions of a software unit in POEM, we do not have to know which software units are used by those versions. For example, when we selected the old version of

poemui in Figure 3-5, we were also implicitly selecting versions for the software units used by the old version of *poemui*: there was no need to select a version for each software unit used by *poemui*. This complies with the software engineering principle of encapsulation, which states that when using a class or function we should not be concerned with its internal structures.

Now suppose that we want to try an alternative approach to implementing the class *poemui*. To do this, we create a new workarea, select the old version of *poemui* in the original workarea, and choose *Revise_Subsystem* under the *Version* menu. As illustrated in Figure 3-6, a version branch of the whole subsystem is created with the same content as the original one. We can work on this new branch as we did on the original version.

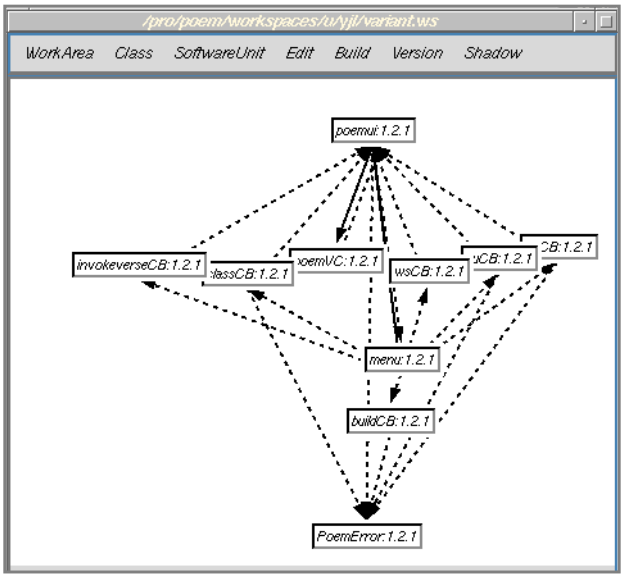


Figure 3-6: A version branch of the class *poemui*

In Figure 3-6, we created a new version branch for every software unit used by *poemui*. If we want a new version branch for *poemui* only, we can choose *Revise_SU* under the *Version* menu instead. A new version branch of *poemui* is created with its links pointing to the same versions of software units used by the original version of *poemui*. Besides, although we created the new version branch in a separate workarea in Figure 3-6, POEM also allows us to create the version branch in the same workarea where the original branch resides. As stated before, POEM allows multiple versions to reside in the same workarea. The only restriction to the *Revise_subsystem* and *Revise_SU* operations is that they can be applied to fixed software units only.

In existing version control tools, files and directories are the units of version control.

There is no simple mechanisms to create versions of functions and classes. With those tools, if we want to keep the current status of the class *poemui*, we have to check in the file that contains *poemui* and all the files that contain classes or functions used by *poemui*. Determining which files *poemui* actually depends on is not trivial. If we miss any of them, then the behavior of *poemui* may be changed when we revert to this version in the future. On the other hand, if we simply make a version for all the files that *poemui* may possibly use, then we may create versions for some files that *poemui* does not depend on.

Creating extra versions for files that *poemui* does not depend on seems harmless, but they can actually cause trouble when we revert to this version of *poemui* in the future. Using existing version control tools, we usually associate a label with all the files that are checked in together. This makes it easier to check them out together in the future. But if we check in some files that *poemui* does not depend on, then those files will also be checked out together when we try to check out the files *poemui* depends on. This side effect will cause trouble because we are also reverting functions and classes that are irrelevant to *poemui* to their earlier status.

3.4 Cooperative Programming

Cooperative programming in POEM is based on workareas. In Figure 3-1, we are the owner of workarea *A*, but workarea *B* is owned by another programmer working on a different machine. POEM allows a programmer to use other programmers' workareas as a client.

Software units in one workarea may have links pointing to software units in other workareas. In the workareas in Figure 3-1, the white boxes are locally defined software units; the gray boxes are actually the *shadows* of software units defined in other workareas. For example, the gray box labeled *poemui:1* near the bottom of workarea *A* is a shadow of the first version of *poemui*, which is located at the top of workarea *B*. The shadow boxes are created automatically when users try to draw a link across the workarea boundaries.

The software unit *poemui:1* is a C++ class that uses many other functions and classes. Some of those functions and classes are defined in other workareas not shown in Figure 3-1. But in spite of this complexity, *poemui:1* is represented as a single shadow box in its client, workarea *A*. When we are using a software unit from another workarea, we can ignore all its internal complexity. Again, this complies with the encapsulation principle in software engineering.

As a client of workarea *B*, we cannot modify its contents, but we can use the software units it contains by pointing links to them as usual. To reduce interference, programmers are

advised to use only the older, stable versions of software units developed by other programmers.

Earlier configuration management tools like MAKE and RCS do not have direct support for cooperative programming. Programmers using those tools in a programming team usually work in different directories to avoid mutual interference and use separate makefiles to manage files in their directories. When the development of software in a directory reaches a certain stage, the owner should check the source files in that directory into a common version repository, so that other programmers can retrieve them from other directories later.

There are two major problems in that approach. First, since MAKE does not support formal parameters and return values, managing the communication between multiple makefiles is difficult. For instance, to use the class *poemui* developed by another programmer, as in our own example, we have to know the set of object files generated by the makefile that manages *poemui*. Since *poemui* uses many functions and classes as its submodules, the makefile that manages *poemui* may generate many object files; we have to know all those files and specify them in our own makefile. If *poemui* depends on some libraries, we also have to know those libraries and specify them in our makefile.

The second problem is the difficulty of updating the software developed by other programmers. Every so often, a programmer has to check out the latest version of code developed by other programmers. Knowing which files to check out is important: if we check out mismatched files, then the status of our program becomes unpredictable. Let us take an example. Suppose that the main part of *poemui* and one of its major submodules are developed by two programmers. The owner of the submodule made some important changes and checked in the files that contain the submodule, but the owner of *poemui* has not yet made the necessary changes to incorporate those changes yet. If we check out the latest version of every file at this moment, then we get the modified version of the submodule and the version of *poemui* that does not know how to cope with the new modification. This will definitely cause trouble.

Some more advanced configuration management systems, like DSEE [51], SHAPE, and ClearCase, have better support for cooperative programming than MAKE and RCS. Users of those systems can set up *version selection rules* to choose versions of files for their private workspaces. This makes version selection in large projects much simpler. However, the two problems discussed above still remain. Since those systems still use MAKE-like tools to handle system building, programmers have to know the internal structure of modules developed by others. Since the version selection rules choose versions in terms of files, programmers may still get mismatched files for a

function or a class if they do not specify their version selection rules carefully. Also, the usefulness of version selection rules depends on the proper labeling of file versions. As discussed in the previous section, labeling versions of files is rather tricky in certain cases.

We have neither of those two problems in POEM. When we use a function or a class, we do not have to know what object files it generates. Similarly, when we select a version of a function or class, we do not have to select versions for the submodules it uses.

3.5 Software Reuse

In POEM, software libraries are represented as special workareas. Figure 3-7 shows the workarea for the Motif library. Each software unit in this workarea corresponds to a Motif widget class or a Motif function. These software units behave just like normal ones. If we click the mouse on a library software unit, POEM loads its header file into the interface editor and its manual pages into the documentation editor.

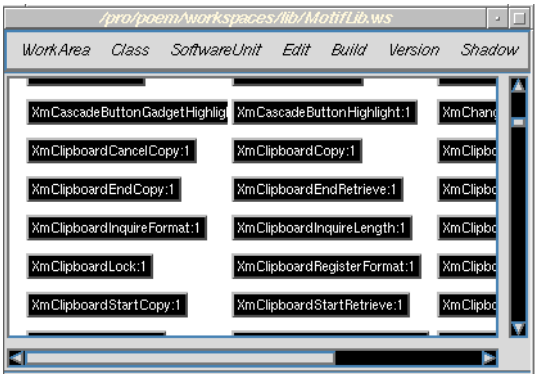


Figure 3-7: The workarea for the Motif library

To use one of these software units, we can simply establish a link to it; POEM will create a shadow for the reused library software unit in the client workarea. For example, the shadow labeled *XmRowColumn* in workarea *A* of Figure 3-1 was created when we tried to establish a link to the software unit in the Motif library. Again, these shadows of library software units behave like normal ones. We can click the mouse on them to get their interfaces and manuals loaded into the editors.

In existing environments, reusing a library function or class is more tedious. For example, to reuse the Motif widget class *XmRowColumn*, programmers have to locate its archive file (e.g. /cs/lib/motif-1.2/libXm.so.1.2) and its header file (e.g. /cs/include/motif-1.2/Xm/RowColumn.h), insert the directive “#include <RowColumn.h>” into the source program, and then specify the

paths of the header files and archive file in the makefiles. If the source file that use *XmRowColumn* is compiled by one makefile but linked into the executable by another makefile, then we have to put the path of the header files in the first makefile but the path of the archive file in the other. At the same time, programmers have to locate the corresponding manual pages in some remote directory (e.g. /cs/data/motif-1.2/man). This is both tedious and error-prone. If there are multiple versions of the Motif library, then programmers may accidentally get the header file from one version and the archive file from another version. The manual pages read by programmers may not match the software either.

A more important advantage of POEM is that we can directly reuse functions and classes that are not made into a special "library format." For example, if we want to reuse the class represented by the software unit *poemui*, which is defined in workarea *B* of Figure 3-1, we can simply establish links from our new software units to *poemui*. This is shown in Figure 3-8. Although *poemui* is not in a library workarea and it depends on many other software units, the reuse can be accomplished with a single operation. There is no need to copy or modify software units in the original workarea.

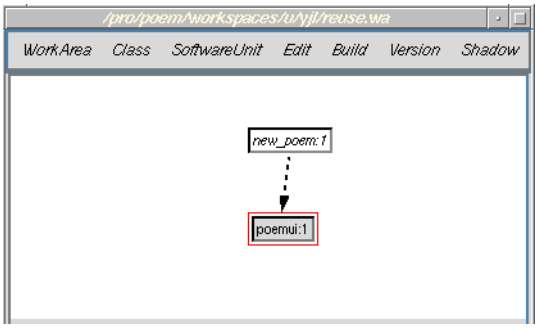


Figure 3-8: Reusing a class.

Actually, from a user's point of view there is no difference between a reused software unit and a locally created one. All software units in a project can be accessed in the same way. We can follow the links to a reused software unit just like browsing any other software units. If we want to modify the source code of a reused software unit, we can invoke the *revise* operation of the reused software unit to get a modifiable version. This ability to tailor a reused software unit is important when we are reusing high-level functions and classes.

In contrast, reusing a function or class that is not made into a special "library format" (e.g. a ".a" file or a ".so" file in UNIX) in a traditional environment is rather difficult. We have to know

the location of all the object files it generates and all the libraries it uses, and then modify makefiles to incorporate all these object files and library files. If the reused function or class is a high-level one, it may have many object files generated in different directories and use many libraries. Consequently, modifying makefiles to incorporate this kind of function or class is especially difficult.

Since it is difficult to reuse functions and classes that are not made into library archives, software reuse in existing environments is practical only when the reusable code is made into library archives. Making large modules into library archives, however, is not trivial and programmers seldom do this with their code. As a result, many opportunities for software reuse are missed.

3.6 Software Maintenance

With POEM, we are trying to help software maintenance in three ways. First, we want to make it easier to understand existing programs. Second, we want to make it easier to retrieve and use old versions. Third, we want to make it easier to restructure an existing program.

To maintain an existing program, the first step is to understand its general structure. POEM helps programmers do this by visualizing the relations between classes and functions in its workarea windows. Functions and classes are represented as software unit boxes and the exchange of interfaces between them is represented as links. If we are interested in any function or class, we can click the mouse button on the corresponding software unit to load its interface, implementation, and documentation into the editors. It does not matter whether the software unit is an active version, a fixed version, or just a shadow of some other software unit.

In this respect, the workarea windows of POEM are a navigation tool: they support a convenient mechanism to let programmers move among functions and classes in a program according to their logical relationship. In contrast, browsing a certain function or class in traditional environments involves locating the file that contains the function or class, loading the file into an editor, and then scrolling to the portion of file that defines it.

Some advanced environments, like FIELD, also support visualization tools that allow programmers to navigate in programs according to their logical structure. While our idea for this mechanism originates from FIELD, we made an improvement by allowing programmers to modify the logical structure of their programs. In FIELD, the diagrams shown in visualization tools are read-only.

POEM also makes it easy to understand the structural changes between versions. In Figure

3-9, we have added a new software unit to the current version of *poemui*. We can easily see the difference between the versions by looking at this diagram.

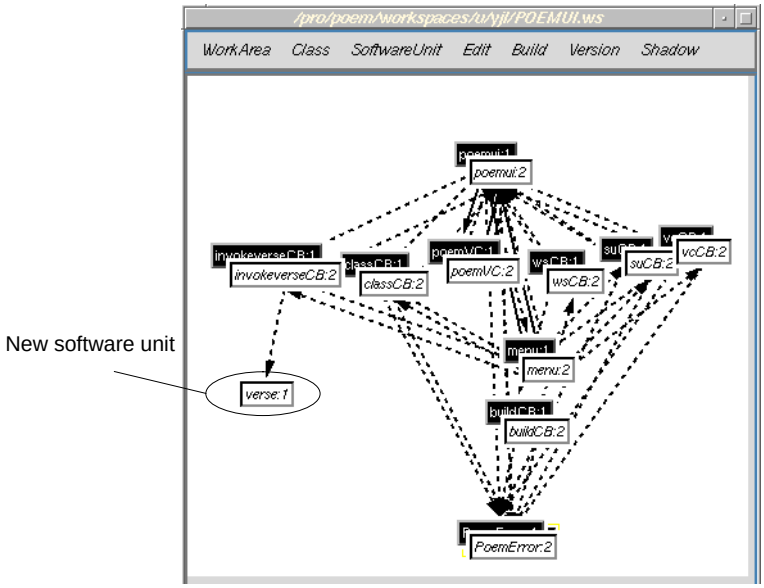


Figure 3-9: Changing the structure of a program

There are many other visualizing tools that can help programmers understand the structures of their software [59][62][41]. POEM differs from them in that the diagrams are not generated by parsing source programs, and the diagrams can be edited directly.

Using POEM, we do not have to use separate operations, like check-out, to retrieve the old versions of software. In Figure 3-9, the old version of *poemui* can be used directly. We can browse its content by clicking the mouse on it, or build its object code by choosing *Build* under the *Build* menu. There is no need for check-out operations.

In Figure 3-9, we changed the structure of our program by adding a new software unit to the current version. But we can build the object code of the new version and the old version as usual. We can select the new version of *poemui* and build its object code by choosing *Build* under the *Build* menu. Similarly, we can select the old version of *poemui* and build its object code by choosing *Build* under the *Build* menu. This is rather simple in POEM. In most other configuration management tools, however, changing the structure of a program requires extra operations. We usually have to make a version of the makefile and label it carefully before making the modification. When we want to build the old version of *poemui*, we have to check out the old version of the makefile before building its object code. When we want to build the current version of *poemui*, we

have to make sure that we are using the current version of the makefile. If we do not get the correct version of makefile, the building will fail.

3.7 Comparison

We conclude this chapter by comparing POEM with several existing configuration management systems, RCS, MAKE, CVS [12], DSEE, SHAPE, ClearCase, TeamWare and Vesta [52]. RCS and MAKE represent early version control and system building tools. They complement each other and are usually used together. CVS is a more advanced version control tool based on RCS. CVS is also paired with MAKE because it does not support system building mechanisms. DSEE is one of the earliest configuration management systems that integrate supports for version control and system building. It is commonly considered as the best configuration management system of the late eighties, and has influenced the design of most later systems. SHAPE is a popular free configuration management toolkit. It follows the philosophy of DSEE but also makes some significant improvements. ClearCase and TeamWare are two of the most popular commercial configuration management systems in the current market. ClearCase follows the composition model pioneered by DSEE; TeamWare follows the long transaction model pioneered by NSE. Vesta, a research system from DEC, uses a unique functional model and shares many common goals with our POEM approach.

The purpose of the following comparison is to argue that a configuration management system following the POEM approach is easier to use than existing configuration management systems. Limited by our resources at a university, the current implementation of POEM supports only the essential mechanisms and does not have the rich functionality of commercial products. The implementation of secondary features of POEM is discussed in Chapter 7 as a future direction of this research. The drawbacks of our approach are also discussed there.

In general, we do not compare the performance of configuration management systems in terms of speed. Since such systems spend most of their time in compilation and archiving files and since the tools for those operations are very similar, the differences in speed of configuration management systems is usually small and can be narrowed by better implementation skills. The only exception might be the avoidance of unnecessary compilation: a system building tool can save a lot of time if it can reuse the result of earlier builds.

Since we already discussed the advantages of some of POEM's features in the previous sections, we do not reiterate them below. We make explanations only when special features are

supported by the systems being compared.

3.7.1 System Building

The major advantages of POEM in system building are that it needs no separate makefiles or system models, it handles object code automatically, and it allows users to change system building options in terms of functions and classes. This is summarized in Table 3-1.

	Description of system building process	**Automatic object-code handling**	**Units of customization**
RCS/MAKE	makefiles	No	files, makefiles
CVS/MAKE	makefiles	No	files, makefiles
DSEE	DSEE system models	Partial (referred indirectly)	files, threads
SHAPE	shapefiles	No	files, shapefiles
ClearCase	makefiles	No	files, makefiles
TeamWare	makefiles	No	files, makefiles
Vesta	Vesta system models	Yes	packages
POEM	links of software units	Yes	functions, classes

Table 3-1: System building features

Users of POEM describe the system building processes by setting links between software units that correspond to the "#include" directives in traditional environments and represent the logical relationships between functions and classes. Most existing systems use separate makefiles or variants of makefiles (e.g. shapefiles) to describe the system building process. The only exceptions are DSEE and Vesta. DSEE describes the structures of a project with a *system model* that contains a set of nested blocks. Vesta uses a set of descriptions (also called system models*)* that contain autonomous functions. The system models of DSEE and Vesta are more structural than makefiles, but they still describe the structures of projects in terms of the compilation and linking processes instead of the logical relations between modules. And like makefiles, they are written in languages that are totally different from the source code. Understanding Vesta system models is especially difficult because of its complicated scoping rules and binding mechanisms.

DSEE and Vesta are also the only two systems other than POEM that support automatic object-code handling. Users of DSEE sometimes still have to refer to object files indirectly by special keywords (e.g. %result and %result_of). Users of Vesta handle the object files implicitly as the return values of functions.

Users of POEM can change the system building options in terms of functions and classes.

But most existing systems let their users change system building options in terms of files or makefiles. Users either change the translation options for a single source file or set the default options for all files managed by a makefile. DSEE allows its users to set translation options for files in a thread according to rules. For example, the following option rule specifies that the DEBUG option should be used for all reserved elements but not for other elements:

```
-reserved -use_options DEBUG
```

Vesta allows its users to set translation options in terms of *packages*, which contain all the files managed by a function in the Vesta system models. Since a package usually maps to a function or a class in the source code, customization of system building processes is more convenient in Vesta. But as discussed in section 2.1, Vesta relies on very complicated binding rules to manage the options.

3.7.2 Version Control

The major advantages of POEM in version control is that it handles version creation and selection in terms of functions and classes, it allows direct access to old versions, and it lets its users use multiple versions of the same software artifact simultaneously. This is summarized in Table 3-2.

	Units of version creation	**Units of version selection**	**Method to retrieve old versions**	**Accessing multiple versions simultaneously**
RCS/MAKE	files	files, files with the same label	Check out	Create separate directories, then check-out files
CVS/MAKE	files, directories	files, directories, files with the same label	Check out	Create separate directories, then check-out files
DSEE	files	files, files with certain attributes	Set version rules in threads	Create separate threads and switch between them
SHAPE	files	files, files with certain attributes	Set version selection rules	Create separate directories, then check-out files
ClearCase	files, directories	files, files with certain attributes	Set configuration rules in config spec	Create separate views and switch between them
TeamWare	files, directories	files, files with the same label	Check out	Create separate directories, then check-out files
Vesta	packages	packages	Check out	Create separate directories, then check-out files
POEM	functions, classes	functions, classes	(No explicit operation required)	(No explicit operation required)

Table 3-2: Version control features

Most existing systems manage versions in terms of files and directories. The only exception is Vesta, which allows its users to create and select versions of packages. DSEE, SHAPE and

ClearCase let their users select versions of files with selection rules, but they still do not allow version selection in terms of functions and classes.

Using POEM, old versions of software units can be accessed directly. But in most existing systems, retrieving old versions is handled by check-out operations. The exceptions are DSEE, SHAPE, and ClearCase. We can access old versions in DSEE and ClearCase by setting version rules in *threads* and *configuration rules* in *configuration specifications*, respectively. DSEE and ClearCase can create *views* of file systems that show only the selected versions. In SHAPE, if we want to retrieve old versions of files for browsing, then explicit check-out operations are required. But if the old versions are required for the purpose of system building, then SHAPE retrieves them automatically according to the user-specified selection rules.

POEM allows multiple versions of the same software unit to exist in the same workarea. But in other existing systems, only one version of the same software artifact can exist in a directory or a workspace. There is no exception among the existing systems we list here. We have to create separate directories, threads, or views if we want to use different versions simultaneously.

3.7.3 Cooperative Programming

The major advantages of POEM in cooperative programming is that it allows programmers to share software in terms of functions and classes, it makes the synchronization between programmers easier, and it makes possible sharing of object code between programmers. This is summarized in Table 3-3.

	Units of sharing between programmers	**Methods to bring updates from colleagues**	**Sharing object code between programmers**
RCS/MAKE	files	check out from repositories	No
CVS/MAKE	files, directories	check out from repositories	No
DSEE	files, threads	Automatic or change threads	Yes
SHAPE	files	Automatic or change selection rules	Yes
ClearCase	files, directories	Automatic or change views	Yes
TeamWare	files	“Bring over” from parent workspaces	No
Vesta	packages	Change system models	Yes
POEM	functions, classes	Set links	Yes

Table 3-3: Cooperative programming features

POEM lets programmers in a programming team divide their work in terms of functions and classes. But in most existing systems, programmers in a programming team deal with files and

directories generated by their colleagues. The only exception is Vesta, which allows programmers to share software in terms of packages.

Using POEM, programmers get the new versions of software developed by their colleagues by setting links to the new versions of software units. Existing systems use different methods to synchronize the changes made by colleagues. Using RCS or CVS, programmers make their changes public by checking in their files and get the changes from their colleagues by checking out files. Users of TeamWare communicate with each other by synchronizing their private workspaces with a shared parent workspace. Users of DSEE, SHAPE and ClearCase may set selection rules that automatically use the most recent versions from their colleagues. This is very convenient but may cause management problems: versions of files selected by a set of selection rules may be incompatible, and versions of files used in a system building process may be changed without being noticed by programmers.

POEM, as well as DSEE, SHAPE, ClearCase, and Vesta, promote the sharing of object code between programmers. All these systems avoid unnecessary compilations by reusing the object code generated by other programmers. Since compilations are the most time-consuming tasks in configuration management, reducing the number of compilations can save a lot of time for programmers.

3.7.4 Software Reuse

The major advantages of POEM in software reuse is that it allows reuse directly in terms of functions and classes, it makes reuse of library modules easier, and it simplifies reuse of modules that are not made into library archives. This is summarized in Table 3-4.

	Units of reuse	**Methods to reuse libraries**	**Methods to reuse software not made into libraries**
RCS/MAKE	files, libraries	Add “#include” directives, specify include directories and libraries directories in makefiles	Copy source files and parts of makefiles, or specify full path names in new makefiles
CVS/MAKE	files, libraries	Add “#include” directives, specify include directories and libraries directories in makefiles	Copy source files and parts of makefiles, or specify full path names in new makefiles
DSEE	files, libraries	Add “#include” directives, specify library names in system models	Copy source files and parts of makefiles, or specify full path names in new makefiles
SHAPE	files, libraries	Add “#include” directives, specify include directories and libraries directories in makefiles	Copy source files and parts of shapefiles, or specify full path names in new shapefiles
ClearCase	files, libraries	add “#include” directives, specify include directories and libraries directories in makefiles	Copy source files and parts of makefiles, or specify full path names in new makefiles
TeamWare	files, libraries	Add “#include” directives, specify include directories and libraries directories in makefiles	Copy source files and parts of makefiles, or specify full path names in new makefiles
Vesta	files, packages	Add import lists in source code, specify library names in system models (for Modula 2)	Specify “import” clauses in system models
POEM	functions, classes	Set links	Set links

Table 3-4: Software reuse features

POEM lets its users reuse software in terms of functions and classes. But users of most other existing systems have to reuse software in terms of files or libraries. The only exception is Vesta, which allows software to be reused in terms of packages.

Users of POEM can reuse a function or a class by setting a link to the corresponding software unit. It does not matter if the reused function or class is in a library or not. But using other existing systems, we have to modify makefiles or system models as well as the source code. This process is especially difficult when the reused software modules are not made into libraries. Again, the only exception is Vesta, which allows packages to be reused directly without being made into libraries.

3.7.5 Software Maintenance

The major advantages of POEM in supporting software maintenance is that it allows programmers to navigate through their programs in terms of the logical structures, and it does not need extra actions to change the structure of software or to use versions of software with different

structures. This is summarized in Table 3-5.

	Navigation in programs	**Extra actions required to change the structure of software**	**Extra actions to use versions of software with different structures**
RCS/MAKE	Changing directories and scrolling files	Check in makefiles, label versions of source files and makefiles	Check out makefiles and source files with certain labels
CVS/MAKE	Changing directories and scrolling files	Check in makefiles, label versions of source files and makefiles	Check out makefiles and source files with certain labels
DSEE	Changing directories and scrolling files	Create new versions of system models, label versions of source files and system models	Specify *model threads* and configuration threads
SHAPE	Changing directories and scrolling files	Check in shapefiles, label versions of source files and shapefiles	Check out shapefiles
ClearCase	Changing directories and scrolling files	Check in makefiles, label versions of source files and makefiles	Specify configuration specifications
TeamWare	Changing directories and scrolling files	Check in makefiles, label versions of source files and makefiles	Check out makefiles and source files with certain labels
Vesta	Changing directories and scrolling files	None	None
POEM	Following links and clicking mouse buttons	None	None

Table 3-5: Maintenance features

Using POEM, we can navigate in our programs by following links between software units. Since the links represent the logical relationships between functions and classes, we can get to the related functions or classes more easily. This ability is further enhanced by the graphical user interface of POEM. In contrast, browsing programs in other systems is carried out in terms of directories and files, which only roughly reflect the logical structure of programs.

Users of POEM and Vesta can change the structures of their programs between versions without extra actions. But using other systems, users have to create versions of makefiles or system models and then label them carefully. If the versions of makefiles and system models are not labeled properly, then it is very difficult to revert to earlier versions. The labeling of makefiles and system models is rather tricky because the same version of makefile or system model may map to many versions of source files.

Chapter 4

The Framework

This chapter discusses the conceptual basis of our framework. We describe the basic concepts and the mechanisms for system building and version control. We discuss how this framework can be used to support software reuse and cooperative programming. We also describe some advanced features that can be used to improve the handling of global definitions and version variants.

4.1 Basic Concepts

The model of POEM has four major concepts: *software units*, *subsystems*, *workareas* and *classes of software units*. A software unit is used to represent a function or a class. A subsystem consists of all the software units on which a function or a class depends. A workarea contains a set of closely related software units that are developed by the same programmer. A class defines the behavior of a certain kind of software unit. In this section, we discuss these basic concepts and their roles in system building and version control.

4.1.1 Software Units

From a programmer's point of view, POEM consists of a set of software units that are interconnected by links. A software unit usually represents a function or a class in C and C++. But if desired, we can put a set of closely related functions or classes into a single software unit. Individual functions and classes in the same software unit, however, cannot be accessed separately.

As illustrated in the inset of Figure 4-1, a software unit contains a set of *data attributes*, a set of *operations*, and a set of *links*. Data attributes are encapsulated and not directly accessible to programmers. Operations can be invoked by other software units or directly by programmers. Links are considered as parts of the software units from which they originate, but can be modified

directly by programmers. POEM allows its users to define new types of links. But in our experience, the uses_interface links and t_uses_interface links are all we need to support the basic functions of our framework.

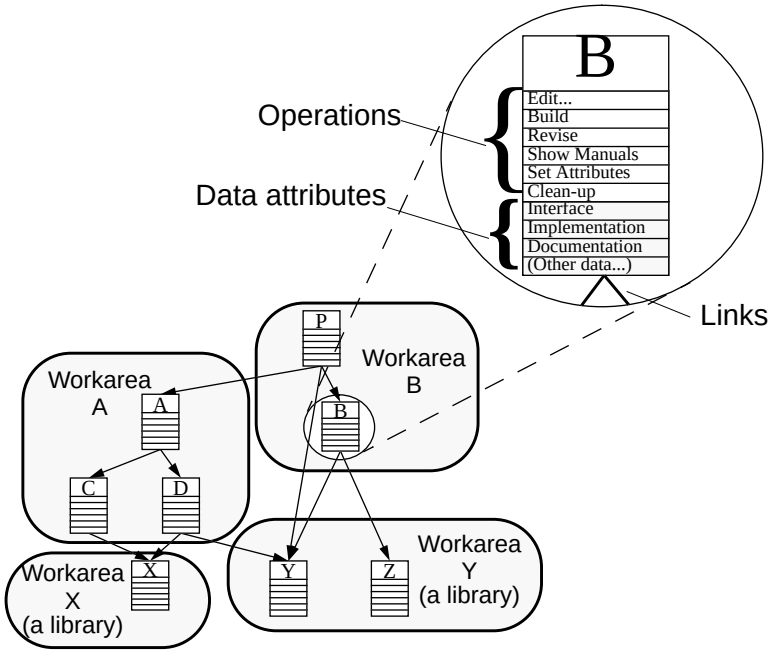


Figure 4-1: The POEM model

The uses_interface links are used to capture the implementation-interface dependencies between software units. A uses_interface link from a software unit *X* to another software unit *Y* implies that the implementation of *X* depends on the interface of *Y*. When the implementation of *X* is being compiled, the interface of *Y* will be included. Figure 4-2 shows the source code of a simple C++ program and the corresponding representation by software units and uses_interface links.

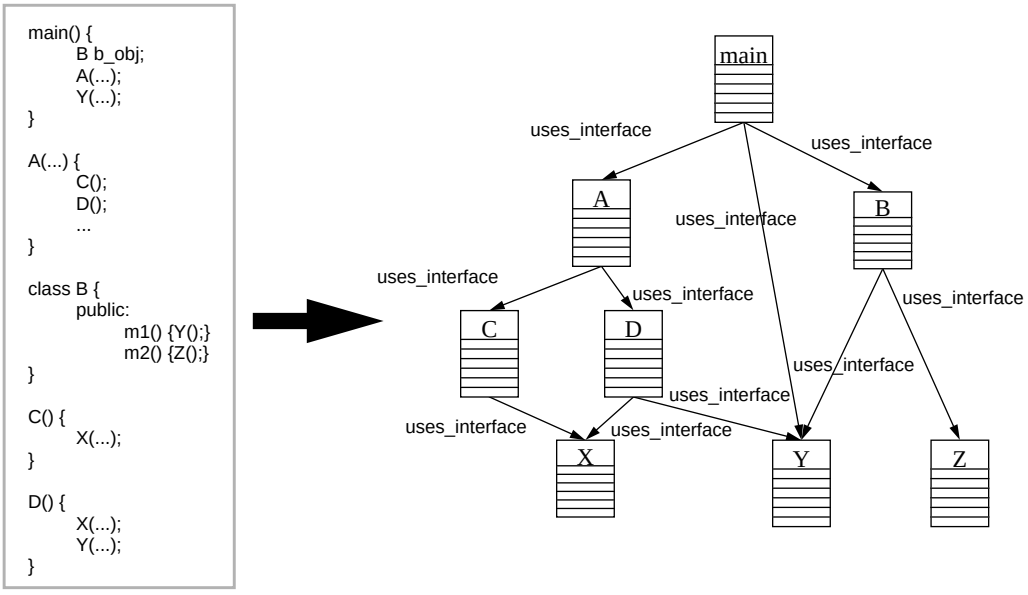


Figure 4-2: Representing a program with software units and uses_interface links

The t_uses_interface[1] links are used to capture the dependencies between interfaces of software units. A uses_interface link from a software unit *X* to another software unit *Y* implies that the interface of *X* depends on the interface of *Y*. Whenever the interface of *Y* is included in a compilation, the interface of *X* will automatically be included.

Figure 4-3 shows the source code of another simple C++ program and the corresponding representation with software units. In this example, the interface of *C* depends on the interfaces of *D* and *E* because *C* is a subclass of *E* and uses a data member of class *D*, the interface of *D* depends on the interface of *F* because *D* is a subclass of *F*, and the implementation of *E* depends on the interface of *G* because *E* has a member function *E::foo()* that uses class *G*. If some other software unit, say *A*, points a uses_interface or t_uses_interface link to *C*, then the interfaces of *C*, *D*, *E*, and *F* will all be included when *A* is compiled. Notice that the interface of *G* will not be included because the link between *E* and *G* is a normal uses_interface link and there is no path of t_uses_interface links from *C* to *G*.

1. T_uses_interface is an abbreviation for “transitively uses interface.”

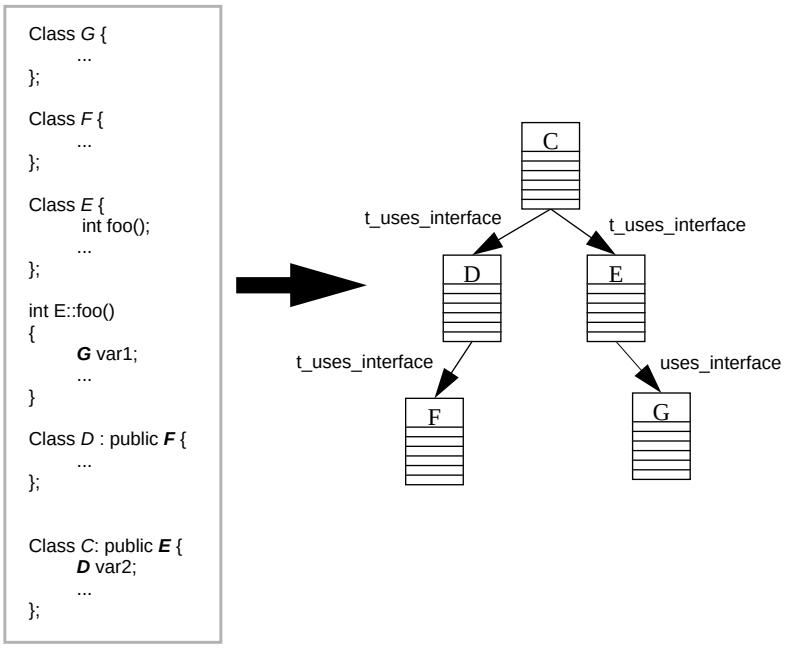


Figure 4-3: t_uses_interface links

The interfaces of *C*, *D*, *E*, and *F* are together called the *effective interface* of *C*. They are interfaces of the software units that can be reached from *C* by t_uses_interface links. Members of an effective interface are ordered. Our model guarantees that if there is a t_uses_interface link from *X* to *Y*, then the interface of *Y* will always appear before the interface of *X* in any effective interface, provided that there are no cycles of t_uses_interface links. A proof of this property is given in the next chapter.

Our model allows cycles of uses_interface links but not cycles of t_uses_interface links. Cycles of t_uses_interface links implies cyclic dependencies between declarations, which are not allowed C or C++. In these circumstances, forward declarations should be used to break the cycles.

The uses_interface and t_uses_interface links capture the relations that are traditionally represented by the "#include" directives in C and C++. A uses_interface link is analogous to including a ".h" file in a ".c" file under traditional C/C++ programming environments. A t_uses_interface is analogous to including a ".h" file in another ".h" file.

Whenever possible, we should use normal uses_interface links instead of t_uses_interface links. A t_uses_interface link represents a stronger bondage between two software units. If there is a t_uses_interface link from *X* to *Y*, then modifications made on the interface

of *Y* also change the effective interface of *X*, which causes all the software units that use *X* to be recompiled. In contrast, if the link between *X* and *Y* is a normal uses_interface link, then we can change the interface of *Y*, or even replace *Y* with another software unit, without affecting the effective interface of *X* or causing the recompilation of the clients of *X*.

Theoretically, uses_interface links alone are enough to describe the dependencies between software units. For example, in Figure 4-4 we have three software units *X*, *Y* and *Z*. *X* has a uses_interface link pointing to *Y*, and *Y* has a t_uses_interface link pointing to *Z*. If we replace the t_uses_interface link from *Y* to *Z* with a normal uses_interface, and add a uses_interface link from *X* to *Z*, then each software unit gets the same set of interfaces: *Y* gets the interface of *Z*, and *X* gets the interfaces of both *Y* and *Z*. With this observation, some earlier configuration management systems, like GANDALF and ADELE, do not allow users to specify the dependencies between interfaces. All the dependencies specified in those systems are between interfaces and implementations.

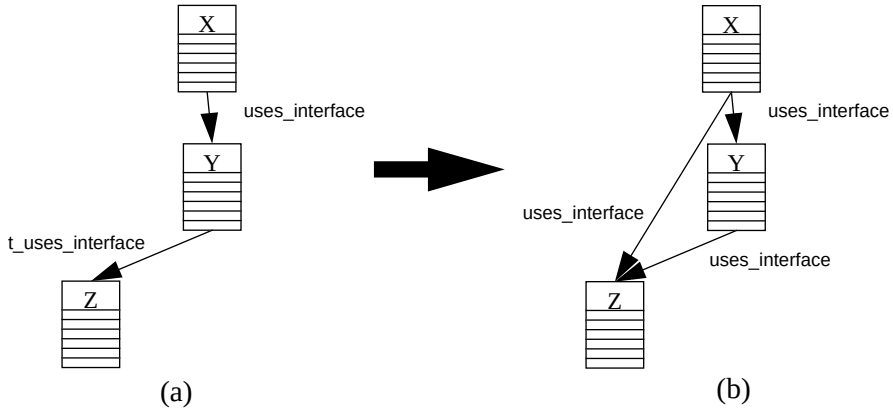


Figure 4-4: Specifying dependencies without t_uses_interface links.

The t_uses_interface links, however, can make things easier. When we point a uses_interface link from *X* to *Y* in Figure 4-4(a), we do not have to know that the interface of *Y* depends on the interface of *Z*. But in Figure 4-4(b), this dependency is important when we uses *Y*. If we forget to point a uses_interface link from *X* to *Z*, we will not be able to compile *X* successfully.

4.1.2 Subsystems

The *subsystem* designated by a software unit *X*, denoted as *S(X)*, is the set of software units that can be reached from *X* via links. *X* is called the *root software unit* of *S(X)*. A subsystem

can usually be operated as a single unit by invoking the operations of its root software unit. For example, invoking the build operation of a software unit *X* generates the derived objects of the whole subsystem *S(X)*, not just those of the root software unit *X*.

According to our definition, subsystems may overlap each other. For example, in Figure 4-2 subsystem *S(A)* contains software units *A*, *C*, *D*, *X* and *Y*, and subsystem *S(B)* contains *B*, *Y*, and *Z*. *Y* appears in both subsystems.

4.1.3 Workareas

Software units are partitioned into mutually exclusive *workareas*. Workareas serve two purposes in our framework. First, they divide the name space of software units into manageable units. Second, they define the boundaries between programming tasks. Each workarea has an *owner* programmer. Only the owner can edit software units in a workarea. The ownership of a workarea, however, can be transferred between programmers.

Each workarea contains some mutable software units that are under development and some immutable ones that represent old versions. The mutable ones are called *active versions*; the immutable ones are called *fixed versions*. Usually, a programmer edits active software units in one workarea, and accesses fixed software units in other workareas at the same time. Fixed software units in remote workareas can be accessed directly without check-in or check-out operations.

The concept of workarea is different from the concept of *workspace*, which is used in many existing configuration management systems [4][19][21][58][78]. In systems that use workspaces, each programmer creates his or her own workspace that copies all the files from a common workspace. To save space, some of those systems supply mechanisms to make *virtual copies*, so that shared files can be logically but not physically duplicated. In our model, workareas are used to partition the set of software units. Software units are not duplicated either physically or logically.

An advantage of using workareas instead of workspaces is that we can handle the sharing of software artifacts between programmers more naturally. Two programmers who use the same version of a subsystem automatically share all the source objects and derived objects of that subsystem. In DSEE [51] and SHAPE [58], the sharing of derived objects is handled by more complicated mechanisms.

4.1.4 Classes of Software Units

Different kinds of software units need different data attributes and different implementation of their operations. For example, some software units used in a C program may contain YACC code instead of plain C code. These software units need an extra data attribute to store the intermediate C code generated by YACC, and different implementation of operations to handle this difference.

To meet this requirement, we define a set of *classes* for commonly used software units, like those for C, C++, and YACC. A class defines a set of data attributes and a set of operations. Users can create new software units as instances of a class. Upon creation, a software unit inherits all the data attributes and operations from the class from which it is created. Users can also create new classes of software units using a language called VERSE. The syntax of VERSE is described in Appendix B, and an example of defining classes with VERSE is given in Appendix C.

A class may inherit from another class. Subclasses of a class inherit the operations and data attributes of their parent class, and may add new operations and new data attributes. A subclass can also redefine the operations that are already defined in its parent class.

Figure 4-5 shows the inheritance relationship among classes of software units in the current implementation of POEM. We supply a set of system-defined classes for commonly used software units, like those for C and C++. Users can also define their own classes the subclasses of the system-defined ones.

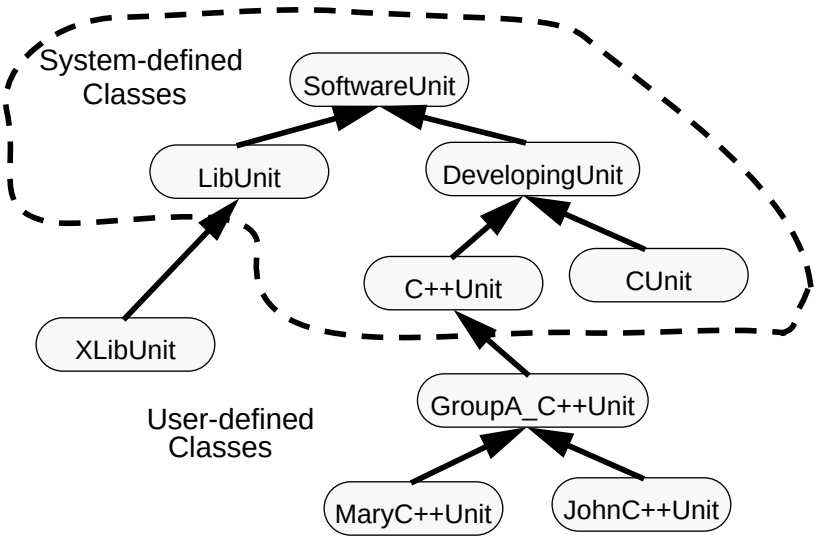


Figure 4-5: Classes of software units

Inheritance helps us in three ways. First, it facilitates the sharing of code among classes. Common operations have to be defined only once, instead of duplicated in each class. For instance, since editing a C program is essentially the same as editing a C++ program, the Edit operation can be defined in DevelopingUnit, which is the common parent class of CUnit and C++Unit.

Second, inheritance guarantees certain degrees of compatibility among the classes of software units. In our framework, we require that all new classes be created as subclasses of existing ones. A subclass can either inherit or redefine the operations of its parent class, but not hide them. This insures that the operations defined in a class are available in all the subclasses. For example, the operations defined in the class SoftwareUnit will be available in all software units in POEM, because SoftwareUnit is the common ancestor of all classes.

Third, inheritance enables users to customize their programming environments incrementally. Users can create subclasses from existing ones and change their behavior incrementally along the inheritance chain. Since our framework carries out most of its tasks by invoking the operations of individual software units, changing the operations of software units has the effect of customizing the whole environment. Users can augment their environment by adding new operations to classes, or change the behavior of their environments by redefining existing operations. For example, a programing team can create a subclass of the system-defined class C++Unit, such as GroupA_C++Unit in Figure 4-5, and change the default compiler of the new subclass. Individual member of the programming team can then create personal subclasses from the team's class, and further customize the behavior of these subclasses to meet their own requirements.

4.2 System Building

With our framework, we want to achieve three major goals in handling system building. First, we want to simplify the specification of the system building process. After establishing the links among software units, programmers should be able to generate the executable of a program without writing a separate makefile. Second, we want to free programmers from manipulating object code. The identification of object files and libraries should be handled automatically, so that programmers can use a large subsystem without knowing what derived objects it generates. Lastly, we want to let programmers customize the system building process in terms of subsystems. For example, programmers should be able to turn on the debugging flag of a subsystem with a single action, no matter how many software units it contains.

System building in our environment is handled by the build operations of software units.

To generate the executable of a program rooted at software unit *X*, programmers invoke the build operation of *X*. If the building is successful, then the executable file is returned as the result of the build operation. If the building fails, then the error messages are associated with the software units containing the erroneous code. Programmers can also build the object code of individual subsystems by invoking the build operations of their root software units.

Internally, this system building process is carried out collaboratively by the build operations of all software units in a subsystem. Each software unit compiles its source code if its object code is outdated, propagates the build message along its links, and then collects the object code from the subsystems it uses. As illustrated in Figure 4-6, the propagation of build messages is essentially a depth-first traversal of the graph formed by software units and their links.

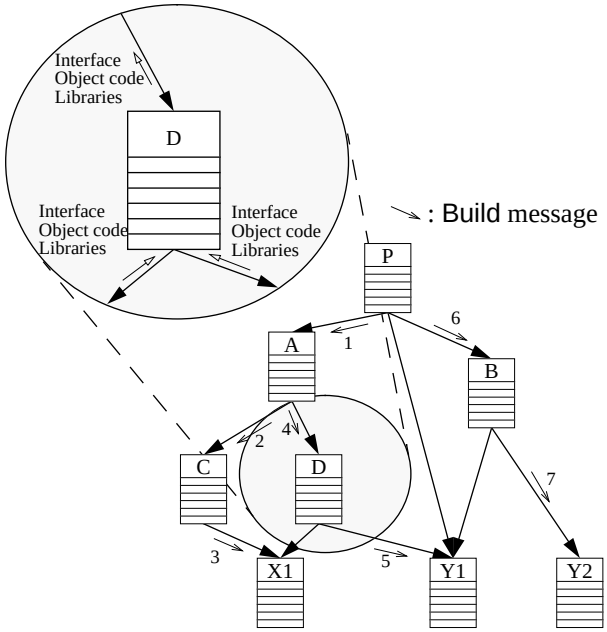


Figure 4-6: The system building process

During the system building process, software artifacts are passed along the uses_interface links and t_uses_interface links. When compiling, a software unit gets the interfaces of its submodules along its links. In the linking stage, object code and libraries are also passed upstream along the links. This flow of software artifacts is illustrated in the inset in Figure 4-6.

An important problem we have to deal with is the cycles formed by links. Cycles of links are sometimes unavoidable when software units are mutually dependent, but may cause infinite loops in the propagation of build messages. One way to avoid this problem is to pass the set of

software units already visited in the propagation as an argument to the build operation. The build operation is propagated only to those software units that are not yet in the set.

The default parameters used to build the derived objects of a software unit are stored in each software unit, including the compiler and compilation flags being used. Programmers can change these parameters by invoking the set_attributes operations of a software unit. For example, programmers can request a software unit to put debugging information in its object code by turning on its debugging flag. By default, the set_attributes messages are propagated along links until they hit the workarea boundaries, so that the same setting takes effect on all the high-level software units of the subsystem. This propagation can be prohibited when programmers want to confine the effects of the new setting to the root software unit.

4.3 Version Control

Our version control system has two major goals. First, we want to provide facilities that let programmers create, select, and use versions in terms of subsystems. Second, we want to simplify access to old versions while still minimizing the space they occupy.

In terms of version control, we classify software units into *fixed versions* and *active versions*. A fixed version is an immutable copy; an active version is a writable working copy. An active version provides a *snapshot* operation that generates an fixed version as its predecessor. A fixed version provides a *revise* operation that creates a new active version as its successor. In the beginning, every new software unit is created as an active version. By repeatedly applying snapshot and revise operations, we can create a version history tree for each software unit, as illustrated in Figure 4-7.

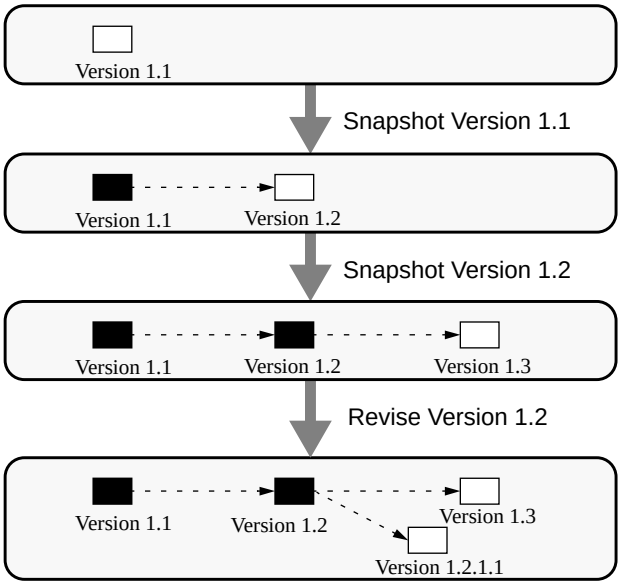


Figure 4-7: Version control of a single software unit

In our framework, invoking the snapshot operation or revise operation on a software unit *X* affects the whole subsystem *S(X)*. When the snapshot operation is invoked, our framework generates fixed versions for all the modified software units in *S(X)* and connects them together in the same way as in the original version. When the revise operation of an active software unit is invoked, our framework creates a subsystem that contains active software units in the local workarea and fixed software unit in remote workareas. The revise operation does not create active versions for software units in remote workareas because remote workareas are usually owned by other programmers and usually we do not want to directly modify code owned by others.

Figure 4-8 shows the effect of applying revise and snapshot operations to a subsystem. Notice that, since we do not modify *C* and *X* in Figure 4-8(c), no new snapshots are created for them in Figure 4-8(d). However, although *D* is not directly modified either, a new snapshot of *D* is created because it uses a modified version of *Y*.

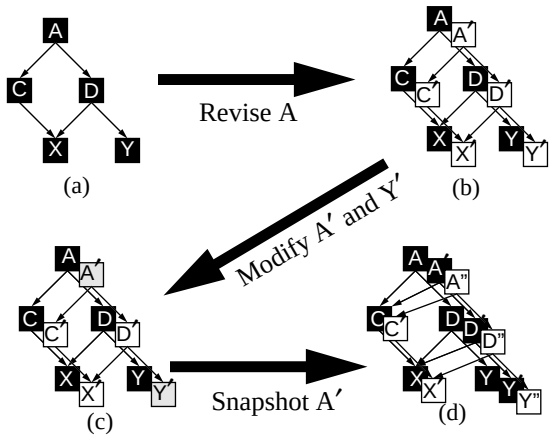


Figure 4-8: Version control of a subsystem

In our framework, a fixed software unit designates a fixed subsystem. We insure that if a software unit *X* is fixed, then all the software units in *S(X)* are also fixed. With this property we can guarantee that, given the same argument in build operations, *S(X)* always generates the same object code.

Like the system building process, the revise and snapshot operations on a subsystem are carried out by the collaboration of individual software units. When revising a subsystem, the revise message is propagated along the link until it hits the boundaries of a workarea. Each software unit then uses its own revise operation to process its data attributes. The snapshot operation on a subsystem is carried out similarly by propagating the snapshot message.

The way in which our model handles composite objects is similar to that of PCTE [13][34]. But there are two major differences. First, while all the attributes of a *stable* object in PCTE are immutable, a fixed software unit in our framework may modify its attributes internally. For example, it may delete its object code and compress its source objects to save space. The only requirement is that each fixed software unit should keep enough information so that the same object code can be regenerated. Second, instead of using the same predefined operations for all objects, we allow different software units to have different implementation of their revise and snapshot operations. This is necessary because different classes of software units may have different data attributes.

In our framework, versions are selected in terms of subsystems. When we make a uses_interface or t_uses_interface link to a certain version of software unit *X*, we are also choosing a certain version of *S(X)*. We do not have to know which versions of software units *S(X)* con-

tains or whether different versions of *S(X)* use different sets of software units.

Compared with DSEE and SHAPE, version selection in our framework is more hierarchical. The selection of versions is in terms of subsystems instead of files, and each software unit hides the selections of its subsystems from its parent software unit. In DSEE and SHAPE, versions of files are selected by *selection rules*. Selection rules may choose files that are not compatible, since the reliability of two separate files does not imply that their combination is also reliable. While this problem is less likely to arise in our framework, we have to insure that only one version of a software unit is used by a version of a subsystem. For example, in Figure 4-9 two versions of D are used in the same version of subsystem *S(X)*. We say that *S(X)* is *versionally inconsistent*. Versional inconsistency causes problems in system building. Either the linker will complain about the redundancy, or a version of the trouble-making software unit will be selected randomly. A possible solution to this problem is to detect these conflicts during the system building process and let users set some rules to resolve them.

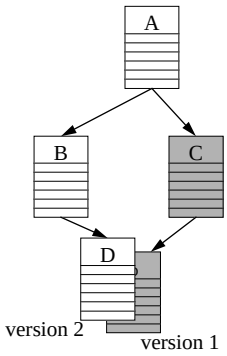


Figure 4-9: A versionally inconsistent subsystem

In fact, the problem of versional inconsistency is not caused by our framework. Our framework only makes it obvious. If we develop the subsystem of Figure 4-9 in DSEE or SHAPE, we may develop and test *C* with the old version of *D*, and develop and test *B* with the new version of *D*. Although the selection rules of DSEE and SHAPE may insure that only one version of *D* is selected when we build *A*, selecting either version of *D* still may cause trouble in *B* or *C*.

Our framework also aims to simplify the access to old versions while reducing the space they occupy. Old versions in our framework are effectively immutable, but we allow the implementation to change them as long as the changes do not result in information loss. To save space, a software unit may delete its derived objects and check in its source code in its snapshot operation.

We require that all these space conserving actions be hidden from users, so that a fixed version can be accessed in the same way as an active version. For example, if a fixed software unit checked in all of its source code into some version repository, then when its edit operation is invoked, it should check out its source code internally before loading the code into editors. We also require that the snapshot operation not change the semantics of a subsystem. A fixed subsystem should keep enough information so that it can regenerate the same derived objects.

Since a fixed software unit may check out its source objects and regenerate its derived objects internally, we need an operation to remove these objects when the fixed software unit is no longer used. We define a clean-up operation for this purpose. A clean-up operation reduces the space occupied by a subsystem without affecting its behavior. Again, we propagate the clean-up messages along links, and let individual software units decide what to do with their data attributes. The clean-up operation may be invoked either explicitly by programmers or internally by the system. When a workarea is short of disk space, it may invoke the clean-up operations of its least-recently-used software units.

4.4 Software Reuse

The ability to handle a large module as a single unit simplifies incorporating reusable modules into new projects. It makes the reuse of library modules easier. It also helps in reusing modules that are not made into libraries.

In existing environments, reusable software is usually represented as libraries. To reuse a library module, programmers have to locate its archive file and its header file and then modify the description of the system building process (e.g. a makefile). At the same time, they need to locate the corresponding manual pages in some remote directory. This is both tedious and error-prone. If there are multiple versions of the same library, then programmers may accidentally get the header file from one version and the archive file from another version. The manual pages read by programmers may not match the software either.

In our framework, we represent library functions and classes as software units that have different internal structures but bear the same interface as other software units. To reuse one such software unit, programmers can simply establish a link to it. With this single action, its header file, archive file, and manual pages are all incorporated into the new project. It also guarantees that all these components come from the same version.

Another problem with existing environments is that software modules can be reused easily

only when they are made into library archive (e.g. ".a" files and ".so" files in UNIX). Since making large modules into library archives is not trivial, programmers seldom do this with their code. As a result, many chances for software reuse are missed. In our framework, functions and classes can be reused directly without being made into a special library format. Because we designate each subsystem by a single root software unit regardless of its internal complexity, a large subsystem can be used in the same way as a library software unit.

Let us take an example. As illustrated in Figure 4-10(a), if we have a new software unit *E* that wants to reuse the subsystem rooted at software unit *A*, we can simply point a link from *E* to *A*. With this single operation, the reuser gets not only the object code of the reused subsystem *S(A)* but also its source code, manual pages, version history, and the operations to handle it. Besides, it does not matter if *E* and *A* reside in different workareas or if *A* uses software units defined in other workareas. In contrast, if we want to reuse a module that is not made into a library archive in a traditional environment, we have to know the location of all the object files it generates and all the libraries it uses. Modifying makefiles to incorporate a large module is usually difficult.

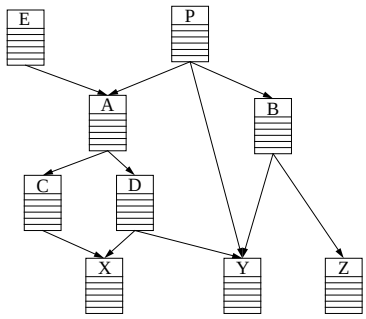


Figure 4-10: Reusing a subsystem

Our framework does not differentiate between a reused module and a locally created one. All software units in a project are treated the same. We can follow links to a reused module just like browsing any other modules. If users want to modify the source code of a reused module, they can invoke the revise operation of the reused module to get a modifiable version. The ability to tailor a reused module is important when we are reusing large modules.

4.5 Cooperative Programming

The major goal of our support for cooperative programming is to let programmers deal directly with functions and classes, instead of files or individual objects, that are generated by other programmers. Our framework also enables us to reduce the interference among program-

mers, and increase the sharing of software artifacts more naturally.

Our scenario for cooperative programming follows the general strategy of DSEE and SHAPE. Programmers should use the older but more stable versions of code from their colleagues and work on the newer but experimental versions of their own code. But instead of dealing with *files*, our framework allows a programmer to choose and use *subsystems* generated by other programmers.

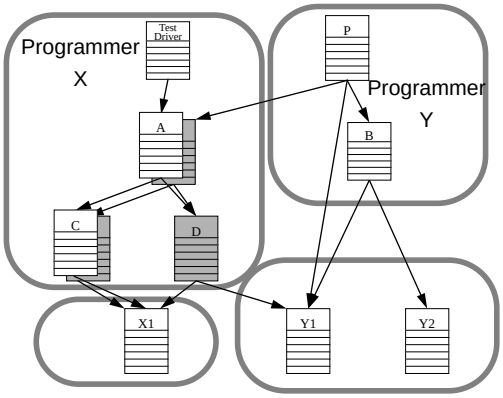


Figure 4-11: Cooperative programming

Figure 4-11 shows a simple example of cooperative programming. Suppose that a program *P* is cooperatively developed by two programmers *X* and *Y*. Programmer *X* works on subsystem *S(A)*. Programmer *Y* works on the main program *P* and subsystem *S(B)*. In order to use *S(A)*, programmer *Y* points a uses_interface link from software unit *P* to software unit *A*. With this single link, the whole subsystem *S(A)* is incorporated into the program. Programmer *Y* does not have to know which software units *S(A)* contains or what derived object it will generate. All the internal complexity of *S(A)* is hidden from programmer *Y*. This is especially helpful when *S(A)* is very large or when *S(A)* uses subsystems developed by other programmers.

Workareas can be used to reduce the interference between programmers. As long as we do not use the active software units defined in other programmers' workareas, their ongoing work does not interfere with ours. Workareas, however, do not confine us to isolated spaces. Fixed software units defined in other programmers' workareas can always be accessed directly: no check-out operation is required and they can be treated just like fixed software units in local work areas. Programmers can browse them, send build messages to them, and create their version successors in local work areas.

Our framework also handles the sharing of software artifacts among programmers natu-

rally. If two programmers use the same version of a subsystem, then they will automatically shared all the source objects and derived objects of that subsystem. For example, since the same version of subsystem *S(D)* is used by both programmer *A* and programmer *B* in Figure 4-11, all its data attributes will be shared. In systems like DSEE and SHAPE, the sharing of derived objects are handled by more complicated mechanisms.

In Chapter 6 we will discuss a new technique called *versioning by delegation* that is used in POEM to improve the management of versions. With that technique, programmers *X* and *Y* in the example above also share the attributes of software units *A* and *C* that are not modified in the new versions.

4.6 Advanced Issues

In the previous sections of this chapter, we have discussed the basic features of our framework. Most programs, even some very complicated ones, can be written with only those features. However, some advanced features can be added into our framework to simplify certain problems. We discuss three of these features in this section. This first two let us handle global definitions more naturally, and the third allows us to select variants of a program more easily.

We also discuss a method that can transform an arbitrary C and C++ program from a traditional programming environment into our framework. This method is rather simple and efficient. But since it does not consider the semantics of the software it transforms, the resulting programs usually do not benefit greatly from our framework. The major reason we discuss this method here is to show that our framework is flexible enough to handle unstructured programming styles.

4.6.1 Global Definitions and uses_globals Links

Global definitions are required when multiple functions or classes want to share information. For example, functions in a subsystem may want to share a set of constants, macro definitions, and global variables. Traditional C/C++ programming environments let us handle global definitions in two ways. We can put the global definitions either in the beginning of a ".c" file and share them among the classes and functions in that file, or in a separate ".h" file that is included by multiple ".c" files.

Similar to the second approach in traditional environments, we can handle global definitions in our framework with software units and uses_interface links. As illustrated in Figure 4-12, if we have three software units *A*, *B*, and *C* that want to share a set of global definitions, we can

store the global definitions as the interface of a separate software unit *A′*, and then set uses_interface links from *A*, *B*, and *C* to *A′*.

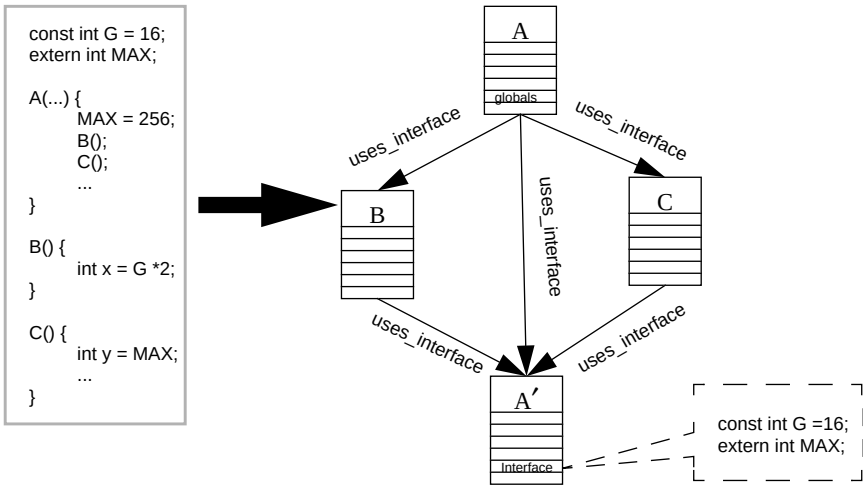


Figure 4-12: Handling global definitions without uses_globals links

There are two problems in this simple solution. First, it does not capture the fact that the global definitions are logically associated with *A*. Since *A* is the main function of the subsystem that contains *A*, *B* and *C*, and the global definitions are shared by all members of the subsystem, it is more reasonable to put the global definitions as an attribute of *A*. Second, since *A′* is directly accessible to other software units, the global definitions in *A′* may be modified by software units outside of the subsystem. This violates the principle of encapsulation and may cause problems in debugging and maintenance.

An alternative way to handle global definitions in our framework is to use the globals attributes and uses_globals links. As illustrated in Figure 4-13, we may handle the situation of Figure 4-12 by putting the global definitions as the globals attribute of *A* and setting uses_globals links from *B* and *C* to *A*.

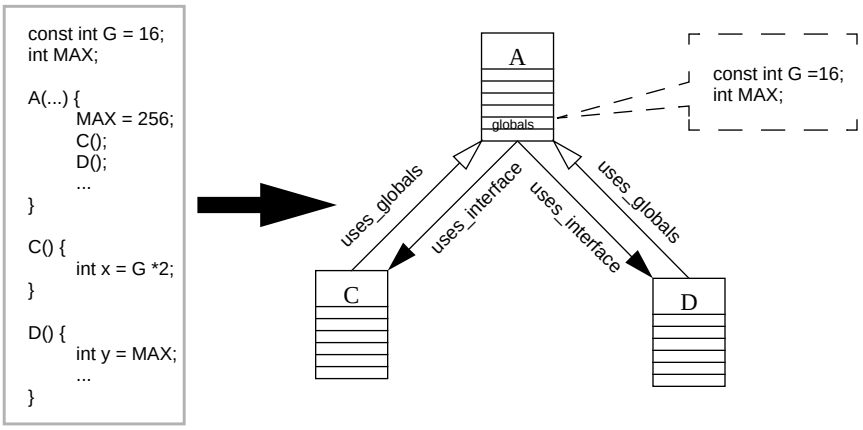


Figure 4-13: Globals and uses_globals links.

A uses_globals link imports the globals attribute from another software unit in the same way as a uses_interface link imports the interface attribute. If there is a uses_globals link from *X* to *Y*, then the globals of *Y* are included in the compilation of *X*.

A uses_globals link usually implies a uses_interface link running in the other direction: if there is a uses_globals link from software unit *X* to software unit *Y*, there should also be a uses_interface link or t_uses_interface link from *Y* to *X*. If there are no such links, then *Y* gets the globals from *X* without being used by *X*. This is usually considered bad design.

4.6.2 t_uses_globals Links

Since we have t_uses_interface links in addition to normal uses_interface links, it is a natural inference that we should have t_uses_globals in addition to the normal uses_globals links. A t_uses_globals link from *X* to *Y* implies that the globals of *X* depend on the globals of *Y*. Any software unit that uses the globals of *X* also gets the globals of *Y*.

With t_uses_globals links, we can simulate a nested scoping rule similar to that of PASCAL and ALGOL. Figure 4-14(b) shows an example with three software units *A*, *B*, and *C*. *C* points a t_uses_globals link to *B* and *B* points a t_uses_globals link to *A*. Since *B* can access the globals defined by *A* and *C* can access the globals defined by both *A* and *B*, this example correspond to the nested function declaration illustrated in Figure 4-14(a). Although no C or C++ compiler directly accepts a program written in this way, our framework can handle the subsystem in Figure 4-14(b) using existing C and C++ compilers.

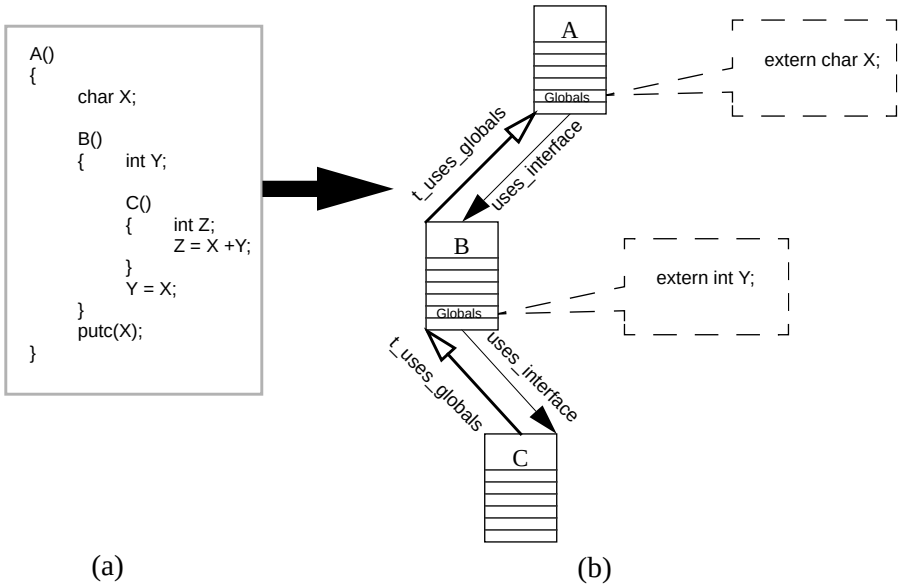


Figure 4-14: Globals and t_uses_globals links

4.6.3 Using uses_globals links and t_uses_globals links

The uses_globals links and t_uses_globals links can make the handling of globals definitions more natural, but they should be avoided whenever possible, since they may increase the interdependency between software units. A software unit with a uses_globals link or a t_uses_globals link represents a subsystem that needs information from its parent subsystem. Reusing this kind of subsystem is problematic. As illustrated in Figure 4-15, if we want to reuse a software unit *C* that has a uses_globals link pointing to another software unit, we should create an active copy of *C* and make its uses_globals link point to the new parent software unit. We also have to make sure that the reuser software unit provides appropriate global definitions to *C′*.

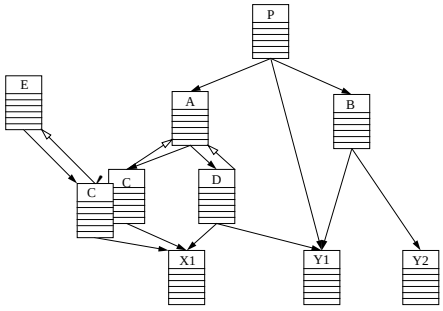


Figure 4-15: Reusing a software unit with uses_globals link

Nevertheless, reusing a subsystem with uses_globals links in our framework is still easier than reusing the same kind of subsystem in traditional environments. In Figure 4-15, although different derived objects are generated for *C* and *C′*, the reuse does not interfere the original project. To reuse the same kind of subsystems in traditional environments without overwriting the object files of the original project, we either have to copy all the files in the subsystem or use a complicated makefile that gets source file from the original directory but generates object files into a new directory.

4.6.4 Defining New Kinds of Links

POEM allows its users to define new kinds of links. But to keep our framework simple, we try to keep the number of kinds of links small. Besides, according to our current experience, the four kinds of links we have are enough to handle most, if not all, programming styles.

4.6.5 Variants and Conditional Links

Variants are alternative implementations of a logical module arising from the need for one software artifact to meet conflicting requirements [88]. For example, a program which is to run on several operating systems needs one version variant for each target operating system. Versions may also be created to meet varying user requirements, to fix bugs in released software, or to help testing and debugging [87].

Managing variants of software items that do not contain any versioned component is straightforward. In existing configuration management systems, variants of a file are usually implemented as version branches. In our framework, too, we can similarly implement variants of a software unit as its version branches.

Managing variants of software items that contain versioned components, however, is more difficult. Suppose we have a program *P* that is to run on three operating systems. Clearly it is impractical to create three variants of the whole program. since doing so would create three variants of every software unit in *P*, including those that are operating-systems independent. This would cause the *multiple maintenance* problem -- to change a software unit that is independent of the operating system, we must repeat the same modification at three places.

Ideally, we should be able to use the same versions of software units that are independent of the operating system, and at the same time select the appropriate version variants for software

units that depend on the operating system. Since version selection in our framework is carried out by setting the destination of links, achieving this goal depends on how we set up links between software units. In terms of variants, we can classify links in our framework into four categories. The first kind of link runs between variant-independent software units. The second kind of link runs between variant-dependent software units. The third kind of link points from variant-dependent software units to variant-independent ones. The last kind of link points from variant-independent software units to variant-dependent ones.

Handling the first three cases is straightforward. The first kind of link is simply the normal one. The second kind of link should run between matching variants. For example, a software unit variant for UNIX can point its links to other variants for UNIX, but not to variants for DOS. In the third case, although multiple variants of a logical software unit may point links to the same software unit, those links will not interfere with each other.

The problem arises in the last case -- a variant-independent software unit that points links to multiple variants of the same logical software unit. Let us take an example. Suppose a software unit *A* depends on another software unit *B*. *A* is independent of the operating systems and has only one version, but *B* depends on operating systems and has three variants, for DOS, VMS, and UNIX. If we point a link from A to each variant of *B*, then all three variants of *B* will be included in the subsystem *S(A)*. During the system building process, the object code generated by all three variants of *B* will be collected as the result of building *S(A)*. Clearly this is not what we want.

To solve this problem, we associate a predicate with each of the three links, as illustrated in Figure 4-16. These links are called *conditional links*. During the system building process, a build message is propagated along a conditional link if and only if the arguments of the build message satisfy the predicate associated with the conditional link. For example, if we send the message build(OS := “DOS”) to software unit *A* in Figure 4-16, then the message is propagated to variant 2 of *B* only. This has the effect of selecting variant 2 of *B*.

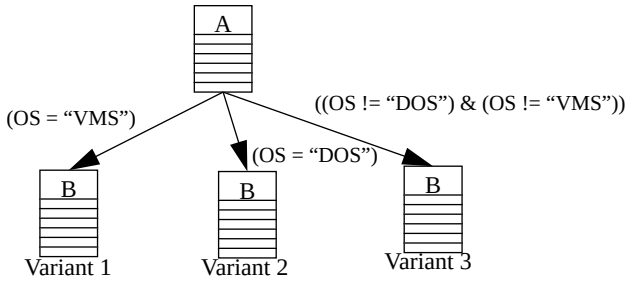


Figure 4-16: Variants and conditional links

Although conditional links should be treated differently in system building, they can be handled in the same way as regular links in version control operations. We can revise or snapshot a subsystem with conditional links without causing any problems.

The conditional links in our framework serve the same purpose as the *version selection rules* in DSEE and SHAPE. In DSEE and SHAPE, users can associate attributes with individual software elements and then select variants in the system building process via selection rules. An software artifact is selected if its attributes satisfy the selection rules. Our framework differs in that we associate predicates instead of attributes with software artifacts, and specify attribute values instead of predicates when selecting variants.

The advantage of our approach is that we can more easily insure that exactly one variant is selected in any selection. In Figure 4-16, since the disjunction of the three predicates, *(OS = “VMS”)* $\vee$ *(OS= “DOS”)* $\vee$ *((OS != “VMS”)* $\wedge$ *(OS != “DOS”))*, is a tautology, we know that at least one variant of *B* will be selected in any selection. On the other hand, since the conjunction of any two of those predicates is false, we also know that no more than one variant will be selected in any selection. We can always have this kind of properties in our framework by setting up *default variants*. For example, variant 3 is the default variant of *B* in Figure 4-16. The predicate associated with the link leading to variant 3 is the complement of the conjunction of the other two predicates.

In contrast, insuring that exactly one variant is selected in DSEE and SHAPE is more difficult. Since the same selection rules are used to select all the elements in a system building process, we have to consider the variants of all elements in a program when specifying selection rules. This is error-prone when the program is large and the variants arise in low-level modules. For example, using DSEE to handle the situation illustrated in Figure 4-16, we can label the second variant of *B* as “DOS”, and use the following selection rule to select the variant for DOS:

.../DOS -when_exist

But if there is another software element *C* that has two variants for DOS, one for DOS with graphical display and one for DOS without graphical display, then both variants of *C* will be selected by this selection rule. Although this specific problem can be solved with more complicated selection rules, we can see that as the variants get more complex, the specification of selection rules becomes more difficult.

Compared with DSEE and SHAPE, the major disadvantage of our approach is that we must specify predicates on every conditional link. The number of links pointing to version variants is usually greater than the number of version variants themselves. Besides, specifying predicates is more difficult than tagging labels.

Supporting conditional links might be expensive. The implementation will not be easy and they may slow down the whole system. We do not support conditional links in the current version of POEM.

4.6.6 Handling Unstructured Programming Styles

Our framework encourages a programming style with structural decomposition and a one-to-one mapping between interfaces and implementations. But it can also handle less organized programming styles where the mapping between interface and implementation is not one-to-one.

To demonstrate this ability, we present as Algorithm 4.1 a simple method that can directly transform an arbitrary C or C++ program from a traditional UNIX environment to our framework. The basic idea of this algorithm is to replace each ".h" file and ".c" file in the original program with a software unit, and replace each "#include" directive with a link in our framework. We also create a separate software unit *P* to represent the whole program. This software unit points a uses_interface link to every software unit that represents a ".c" file. During the system building process, the object code generated by all the software units is collected at *P* and linked into an executable program there. Since our version control operations are independent of the semantics of software units, we can use them to create and manage versions of software units and subsystems as usual.

Algorithm 4.1 Transforming an arbitrary C or C++ program into the POEM model

```
1: For each header file H in the original program:
2:       Create a new software unit with H as its interface;
3:       /* Leave its implementation empty. */
4: For each C or C++ source file S in the original program:
5:       Create a new software unit with S as its implementation;
6:       /* Leave its interface empty. */
7: For each header file H in the original program:
8:       For each "#include G" directive in H:
9:             Set a t_uses_interface link from the software unit representing H
10:            to the software unit representing G;
11:For each C or C++ source file S in the original program:
12:      For each "#include G" directive in S:
13:            Set a uses_interface link from the software unit representing S
14:            to the software unit representing G;
15:Create a software unit P to represent the whole program;
16:For each software unit S created at line 5:
17:      Set a uses_interface link from P to S;
```

Figure 4-17 shows a example of applying Algorithm 4.1 on a simple program. The structure of the transformed program is similar to the structure described by the "#include" directives in the original program. This kind of structure is usually specified as the dependencies in a makefile.

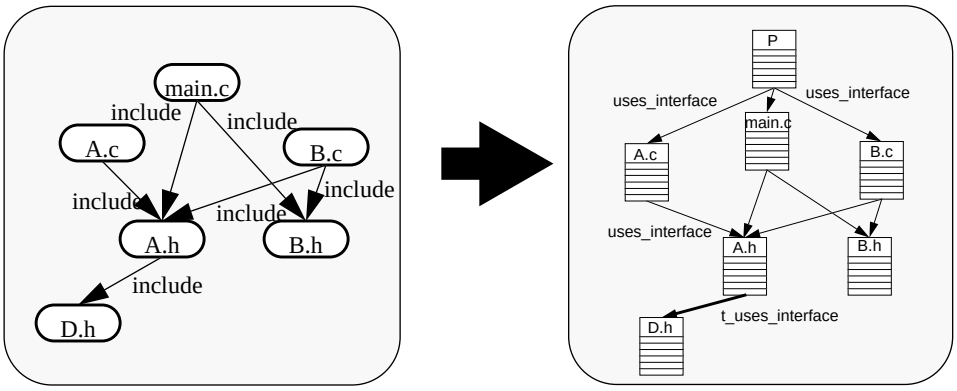


Figure 4-17: Transforming an arbitrary C or C++ program into software units

A program transformed by Algorithm 4.1, however, can seldom benefit from the advantages of our framework. Using software units without interface parts is against our principle of programming in terms of logical structures. Because the interface part represents the purpose of a software unit, a software unit without the interface part is one that does not correspond to any logical concept. Besides, links in the transformed program will not reflect the logical structure of our program faithfully. In Figure 4-17, the functions and classes declared in *B.h* are possibly the parent

modules of the functions and classes declared in *A.h*, because *B.c* has a link to *A.h*. But there is no link from *B.h* to *A.h* to indicate this relationship.

4.7 Comparison

In existing programming environments, configuration management is handled with tools that run *on top of* data repositories. As illustrated in Figure 4-18, traditional UNIX programming environments manage configuration with tools like MAKE and RCS that operate on files and directories. STONEMAN [18] envisioned configuration management in an Ada Programming Support Environment (APSE) as tools running on top of a database. In the ECMA [25] "Toaster Model," low-level configuration management is supported by a database, but higher-level activities are still carried out by separate tools.

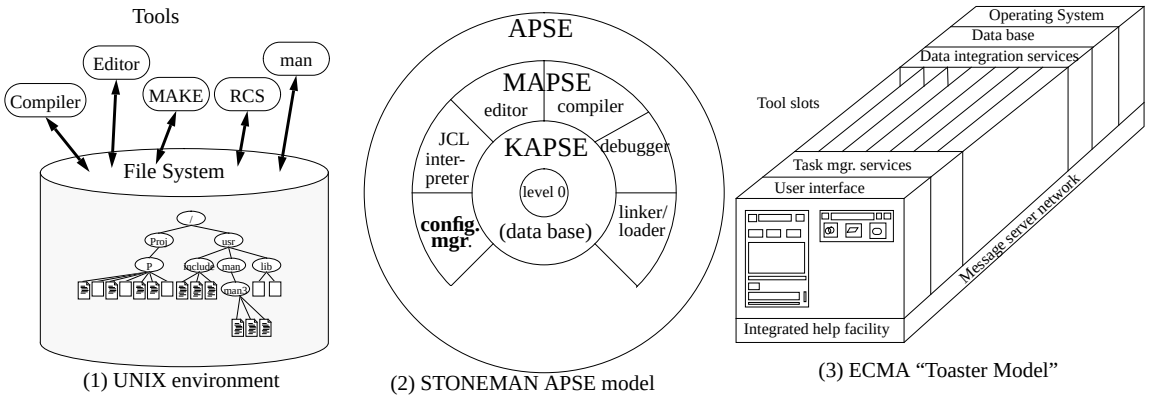


Figure 4-18: Configuration management in existing environments

In contrast, configuration management in our framework is carried out by the operations of software units, instead of tools that are separated from the data repository. A software unit encapsulates all the source code, derived objects, documentations, and building script of the corresponding function or class, and supplies a set of operations to programmers. Programmers develop software by invoking the operations of software units, not by using a set of tools.

We organize software artifacts at the *configuration management level* in parallel with the modularization at the *source code level*. A software unit plays two roles at the same time. It represents a function or a class in source code and it corresponds to a configuration in configuration management. Since we use the same decomposition structure, the gap between handling source code and managing configurations is narrowed. Additionally, since we represent the relations between modules explicitly as links, programmers can examine and traverse them directly without

looking into the source code. This helps programmers to understand the general structure of a program.

Our approach applies the principles of modularization and encapsulation in configuration management. It is well known that modularization and encapsulation are very useful in managing source programs. Similarly, they can help us greatly in handling configurations. As pointed out by Osterweil [66], software processes can be considered as special programs that are enacted by both human and computers. The data used by these special programs are software artifacts like source code, derived objects, system models, documentation, version history files, etc. By applying the principles of modularization and encapsulation to these software artifacts, we can greatly simplify configuration management. Modularization helps us decompose the configuration management tasks into manageable units. Encapsulation helps us hide the different configuration management requirements of individual modules.

Chapter 5

Formal Model

This chapter provides a formal model for the framework described in the previous chapters. We give more rigorous definitions of the mechanism used in our framework and derive a set of properties from these definitions. These properties tell us what is guaranteed in our model and what is not.

This formal model helps us in two ways. First, it gives us a deeper understanding of our framework. It tells us what will happen and what will not, and helps us in extracting the essential elements of our framework. Second, it helps us in finding loopholes in our framework. In the process of developing this formal model, we have noticed several problems that were overlooked when we implemented the first version of our prototype system. That led to several important modifications and gave us a clear idea about how to make improvements in the future.

The formal model described in this chapter is not tied to any specific implementation. But to give reader a more concrete feeling, we occasionally discuss how features in our model can be implemented on top of existing operating systems and database systems.

5.1 Basic Data Types

In our formal model we use five atomic data types and two data structures. The five atomic data types are *source code fragments*, *identities*, *derived objects*, *arguments*, and *Boolean values*. The two data structures are *sets* and *lists*.

A source code fragment is a piece of contiguous text. For example, a source code fragment may contain the definition of a C function or the declaration of a C++ class. In traditional file systems, a source code fragment may be represented as a file. But this is not necessarily the case. If we implement our model on top of an object-oriented database, then a source code fragment may

be represented as an object. Our only requirement is that we be able to test the equivalence of two source code fragments. The actual implementation of the equality test may be based on a simple lexical comparison or on a more complex algorithm that uses syntax and semantics information. It is up to the implementation to define the actual meaning of equality between source code fragments.

An identity is the property of an object that distinguishes it from all other objects [46]. The identity of an object should remain invariant across all possible modification of the object's value. In our model, the only major "object" to be discussed is the software unit, and we will be concerned with the identities of software units only. We distinguish software units by comparing their identities, and establish a link from a software unit *x* to another software unit *y* by storing the identity of *y* at *x*. The special identity value *NULL* is used to represent the absence of a software unit. For example, if the variable *l* is used to represent a link to a software unit and its current value is *NULL*, then no such link exists.

The actual representation of identities also depends on the operating systems and database systems used for implementation. If software units are represented as files in a conventional file system, then the file names can be used as their identities. If software units are represented as tuples in a relational database, then identities can be implemented as identifier keys. In ObjectStore, which is used for the current implementation of our model, identities can be represented as the address of persistent objects. However, all these approaches may have their own drawbacks, as discussed in [46].

Derived objects are the result of applying translators, like compilers and linkers, on other software objects. As with source code fragments, the only operation we need on derived objects is the testing of equivalence. Derived objects can be represented as files in file systems or objects in an object-oriented database. However, we want to make the distinction between derived objects and source code fragments, because a derived object cannot be used when a source code fragment is required, nor vice versa.

An *argument* is the option values for a compilation or linking command. In most existing systems, an argument is represented as a character string. For example, the argument "-g -p" means "provide debugging information and generate extra code for profiling" for the GNU C compiler. In our model, however, we do not need to know the format or meaning of the arguments. Again, the only operation we need on arguments is the testing of equivalence.

We use Boolean values and sets as they are conventionally defined. The union, intersec-

tion, membership, difference, and equivalence-test operations on sets are represented by "$\cup$", "$\cap$", "$\in$", "-", and "=" respectively. An empty set is represented as "{ }". In our model, we use sets of atomic types only. We do not need sets of sets or sets of lists. Also, we do not use sets with elements of different types.

A list is an ordered collection. We use square brackets to represent lists. For example, "[x, *y, z*]" is a list with three elements; "[ ]" is an empty list. The operations we need on lists are concatenation, membership, indexing, and testing of equivalence. We use the operator "+" for the concatenation of lists. For example, the result of "*[x, y, z] + [u, v]*" is "*[x, y, z, u, v]*." The membership of lists is represented by the "$\in$" operator. For example, "$x \in$ *[x, y, z]*" but "$x \notin$ *[u, v]*." The operator "*INDEX*" takes a list and a element and returns the position of the element in the list. For example, the result of *INDEX([x, y, z], y)* is 2. If the second argument is not in the first argument, than *INDEX* returns ∞. If an element appears more than once in a list, then *INDEX* returns the position of its first appearance. Two lists *x* and *y* are equal, denoted *x = y*, if and only if they have the same number of elements and all of their corresponding elements are equal. In our model, we use lists of atomic types only. We do not need lists of lists or lists of sets. Also, we do not use lists with elements that belong to different types.

In the following discussion, we denote the universe of source code fragments, that is, the set of all source code fragments, as Φ. We also denote the universe of identities as $\mathcal{I}$, the universe of derived objects as O, and the universe of arguments as $\mathcal{A}$.

5.2 Software Units

Like other object-oriented systems, our framework consists of a set of interconnected objects. The major objects in our framework are software units. Other major concepts in our framework, like workareas and software unit classes, can also be represented as objects. But doing so only increases the complexity of our model, and we found that it brings no more insights into our discussion.

A software unit has an unique identity and contains a set of attributes. For the purpose of our discussion here, we can represent a software unit as a tuple:

Definition 5.1 Software Unit

A software unit S is a tuple (ID, INF, IMP, G, UI, TUI, UG, TUG, VP, FX) where
- *ID is the identity of S,*
- *INF is a source code segment, called the* ***interface****,*
- *IMP is a source code segment, called the* ***implementation****,*
- *G is a source code segment, called the* ***globals****,*
- *NUI is a set of identities, called the (normal)* ***uses_interface*** *links,*
- *TUI is a set of identities, called the* ***t_uses_interface*** *links,*
- *NUG is a set of identities, called the (normal)* ***uses_globals*** *links,*
- *TUG is a set of identities, called the* ***t_uses_globals*** *links,*
- *VP is an identity, called the* ***version predecessor pointer*** *of S,*
- *FX is a Boolean value, called the* ***fixed flag****.*

In Definition 5.1, we do not include the operations of a software unit into the tuple. Again, we found that doing so increases the complexity of our discussion, but brings no more insights. Also, we list only the essential attributes of a software unit. A software unit in actual implementations may contain many more secondary data attributes. Our framework also allows users to define new classes of software units that contain additional attributes.

Among the members of the tuple, *ID* represents the identity of a software unit. *INF*, *IMP*, and *G* are source code segments. *NUI*, *TUI*, *NUG*, and *TUG* are links to other software units that can be directly modified by users. *VP* is a link used by version control operations. It is for internal use only and users cannot modify it directly. *FX* is a Boolean flag indicating the mutability of the containing software unit. If it is true, then the software unit is a immutable fixed version. If it is false, then the software unit is a writable active version.

We denote the member of a software unit by the dot notation. For instance, if we have a software unit *x*, we denote its interface by *x.INF* and its normal uses_interface links by *x.NUI*. We say that two software units *x* and *y* are equal and denote it as *x = y* if and only if they have the same value for each of their members.

Links between software units are represented by identities in our model. A link from a software unit *x* to another software unit *y* is represented by storing the identity of *y* in *x*. For example, if *y.ID* is a member of *x.TUI*, then there is a t_uses_interface link from *x* to *y*.

An important characteristic of object-oriented systems is the *complex states* of objects [93]. The state of an object depends not only on its own data attributes but also on the objects it refers to. It is usually meaningless to discuss the state of an isolated object, or to discuss the links of a single software unit without considering other software units.

For clarity, we always specify a set of software units as the context of our discussion. This set of software units corresponds to all the software units in our programming environment and is called a *software unit database*. A software unit database *D* is *consistent* if and only if each member of *D* has a unique identity and every link in *D* points to another software unit in *D*.

Definition 5.2 Consistent Software Unit Databases

A consistent software unit database D is a set of software units that satisfies the following properties:
- Unique identity:

$$\forall x, y \in D \qquad (x.\mathrm{ID} = y.\mathrm{ID}) \Rightarrow (x = y) \tag{5.1}$$

- No dangling links:

$$\forall x \in D \;\; \forall i \in I \qquad i \in (x.\mathrm{UI} \cup x.\mathrm{TUI} \cup x.\mathrm{UG} \cup x.\mathrm{TUG} \cup \{x.\mathrm{VP}\}) \Rightarrow \exists y \in D \;\; (y.\mathrm{ID} = i) \tag{5.2}$$

We denote the universe of software units as Σ and the universe of consistent software unit databases as Δ. Clearly, Δ is a subset of all possible sets of software units. That is:

$$\Delta \subset 2^{\Sigma} \tag{5.3}$$

5.2.1 Uses_interface and t_uses_interface Links

With software unit databases serving as the context of our discussion, we can find the sets of software units that are pointed to by the uses_interface links and the t_uses_interface links of a software unit:

Definition 5.3 NUsesInterface, TUsesInterface and UsesInterface

Given a software unit x in a consistent software unit database D, NUsesInterface(D, x) is the set of software units to which x points a uses_interface link:

$$NUsesInterface(D, x) \equiv \{y \in D | y.ID \in x.TUI\} \tag{5.4}$$

Given a software unit x in a consistent software unit database D, TUsesInterface(D, x) is the set of software units to which x points a t_uses_globals link:

$$TUsesInterface(D, x) \equiv \{y \in D | y.ID \in x.TUI\} \tag{5.5}$$

Given a software unit x in a consistent software unit database D, UsesInterface(D, x) is the set of software units whose globals are used by x:

$$UsesInterface(D, x) \equiv NUsesInterface(D, x) \cup TUsesInterface(D, x) \tag{5.6}$$

We can also calculate the reflexive transitive closures of *TUsesInterface* and *UsesInter-*

face, which are the set of software units that can be reached from *x* via paths of t_uses_interface links, and the set of software units that can be reached from *x* via paths of uses_interface links or t_uses_interface links respectively.

$$TUsesInterface^*(D,x) \equiv \{y \mid (y = x) \vee \exists z\,((y \in TUsesInterface(D,z)) \wedge (z \in TUsesInterface^*(D,x)))\} \quad (5.7)$$

$$UsesInterface^*(D,x) \equiv \{y \mid (y = x) \vee \exists z\,((y \in UsesInterface(D,z)) \wedge (z \in UsesInterface^*(D,x)))\} \quad (5.8)$$

The reflexive transitive closure of *NUsesInterface* can also be defined in the same way, but we do not need it in our model.

By definition, we know that the membership of *TUsesInterface** is a transitive relation:

$$\forall D \in \Delta \;\; \forall x,y,z \in D \quad x \in TUsesInterface^*(D,y) \wedge y \in TUsesInterface^*(D,z) \Rightarrow x \in TUsesInterface^*(D,z) \quad (5.9)$$

Similarly, the membership of *UsesInterface** is transitive:

$$\forall D \in \Delta \;\; \forall x,y,z \in D \quad x \in UsesInterface^*(D,y) \wedge y \in UsesInterface^*(D,z) \Rightarrow x \in UsesInterface^*(D,z) \quad (5.10)$$

As discussed in section 4.1, cycles of t_uses_interface links may cause problems. We define software unit databases free of t_uses_interface cycles as following:

Definition 5.4 Software Unit Databases Free of t_uses_interface Cycles

A consistent software unit database D is free of t_uses_interface cycles, denoted TUI_CYCLE_FREE(D), if and only if for any two software units x and y in D, there cannot be both a path of t_uses_interface links from x to y and a path of t_uses_interface links from y to x.

5.2.2 Uses_globals and t_uses_globals Links

The globals, uses_globals links and t_uses_globals links are basically symmetric to the interface, uses_interface links and t_uses_interface links of a software unit. We can define *NUses-Globals*, *TUsesGlobals* and *UsesGlobals* in the same way as we define *NUsesInterface*, *TUsesInterface* and *UsesInterface*:

Definition 5.5 NUsesGlobals, TUsesGlobals, and UsesGlobals

Given a software unit x in a consistent software unit database D, NUsesGlobals(D, x) is the set of all t software units to which x points a uses_globals link:

$$NUsesGlobals(D, x) \equiv \{y \in D | y \cdot ID \in x \cdot TUI\} \tag{5.11}$$

Given a software unit x in a consistent software unit database D, TUsesInterface(D, x) is the set of all software units to which x points a t_uses_globals link:

$$TUsesGlobals(D, x) \equiv \{y \in D | y \cdot ID \in x \cdot TUI\} \tag{5.12}$$

Given a software unit x in a consistent software unit database D, UsesInterface(D, x) is the set of all software units whose interface are used by x:

$$UsesGlobals(D, x) \equiv NUsesGlobals(D, x) \cup TUsesGlobals(D, x) \tag{5.13}$$

And we can calculate the set of software units that are reachable from a software unit via t_uses_globals links and the set of software units that are reachable from a software unit via either uses_globals links or t_uses_globals links:

$$\begin{aligned} &TUsesGlobals^*(D, x) \\ &\equiv \{y | (y = x) \vee \exists z ((y \in TUsesGlobals(D, z)) \wedge (z \in TUsesGlobals^*(D, x)))\} \end{aligned} \tag{5.14}$$

$$\begin{aligned} &UsesGlobals^*(D, x) \\ &\equiv \{y | (y = x) \vee \exists z ((y \in UsesGlobals(D, z)) \wedge (z \in UsesGlobals^*(D, x)))\} \end{aligned} \tag{5.15}$$

Again, it is not hard to see that the memberships of *TUsesGlobals*(D,x)* and *UsesGlobals*(D,x)* are both transitive relations.

$$\begin{aligned} &\forall D \in \Delta \ \ \forall x, y, z \in D \\ &\quad x \in TUsesGlobals^*(D, y) \wedge y \in TUsesGlobals^*(D, z) \Rightarrow x \in TUsesGlobals^*(D, z) \end{aligned} \tag{5.16}$$

$$\begin{aligned} &\forall D \in \Delta \ \ \forall x, y, z \in D \\ &\quad x \in UsesGlobals^*(D, y) \wedge y \in UsesGlobals^*(D, z) \Rightarrow x \in UsesGlobals^*(D, z) \end{aligned} \tag{5.17}$$

In the same way we define software units databases free of t_uses_interface cycles, we can define software unit database free of t_uses_globals cycles as following:

Definition 5.6 Software Unit Databases Free of t_uses_globals Cycles

A consistent software unit database D is free of t_uses_globals cycles, denoted TUG_CYCLE_FREE(D), if and only if for any two software units x and y in D, there cannot be both a path of t_uses_globals links from x to y and a path of t_uses_globals links from y to x.

5.2.3 Uses

Sometimes we simply want to know which software units are referred by a certain software unit, and there is no need to distinguish between kinds of links. We define *Uses(D,x)* for this purpose:

Definition 5.7 Uses

Given a software unit x in a software unit database D, Uses(D, x) is the set of software units to which x points a link:

$$Uses(D,x) \equiv UsesInterface(D,x) \cup UsesGlobals(D,x) \tag{5.18}$$

And we can define the reflexive transitive closure of *Uses(D, x)*:

$$Uses^*(D,x) \equiv \{y \mid (y = x) \vee \exists z\,((y \in Uses(D,z)) \wedge (z \in Uses^*(D,x)))\} \tag{5.19}$$

Definition 5.8 Version Predecessor

The version predecessor of a software unit x in a software unit database D, VPred(D, x), is the software unit in D whose identity matches x.VP. If there is no such software unit in D, then VPred(D, x) is NULL.

Because every software unit in a consistent software unit database has a unique identity, we know that at most one software unit will have the identity that matches *x.VP.*

5.3 Subsystems

During the system building process in our framework, a software unit *x* collects the object code generated by all the software units it can reach via uses_interface links and t_uses_interface links, and then pass them upward to the software units that point uses_interface or t_uses_interface links to *x*. The subsystem designated by *x* is the set of software units that contribute to the object code collected by *x*.

Definition 5.9 Subsystem

The subsystem designated by a software unit x in a software unit database D, denoted as S(D, x), is the set of software units that can be reached from x via uses_interface or t_uses_interface links:

$$S(D,x) \equiv \{y \mid y \in UsesInterface^*(D,x)\} \tag{5.20}$$

Among all subsystems, the ones that have no uses_globals or t_uses_globals links are

especially important because they can be reused more easily. We call them *autonomous subsystems*.

Definition 5.10 Autonomous Subsystem

A subsystem is autonomous if and only if it has no uses_globals or t_uses_globals links pointing out of it:

$$Autonomous(S(D,x)) \equiv [(y \in S(D,x)) \wedge (z \in UsesGlobals(D,y)) \Rightarrow z \in S(D,x)] \quad (5.21)$$

The semantics of a software unit depends not only on the software units in the subsystem it designates, but also on the software units it can reach via uses_globals or t_uses_globals links. Taking this into consideration, we define the *extended subsystem* designated by a software unit *x* to be the set of software units that can be reached from *x* via any kind of links:

Definition 5.11 Extended Subsystem

The extended subsystem designated by a software unit x in a software unit database D, designated XS(D, x), is the set of software units that can be reached from x via links:

$$XS(D,x) \equiv \{y | y \in Uses^*(D,x)\} \quad (5.22)$$

If a subsystem *S(D, x)* is autonomous, then there is no uses_globals link or t_uses_globals link pointing out of *S(D,X)*, and we know that *XS(D, x)* is the same as *S(D, x)*. On the other hand, if the subsystem and extended subsystem designated by *x* are the same, then we know that *S(D, x)* is autonomous because there will be no uses_globals link or t_uses_globals link pointing out of the subsystem:

$$Autonomous(S(D,x)) \Leftrightarrow S(D,x) = XS(D,x) \quad (5.23)$$

This implies that if a subsystem designated by *x* is autonomous, then *x* does not depend on any software unit outside of the subsystem.

5.4 System Building

When we compile a file *x* in a traditional C/C++ programming environment, what is fed to the compiler is not only the code contained in *x* but also the files included by *x*. Similarly, when we compile a software unit *x* in our framework, we should feed the compile with a list of source code segments that contains not only the implementation of *x* but also the interfaces and globals on which *x* depends. In this section, we discuss how this list of source code segments can be calcu-

lated. We also discuss how compilation and linking are handled in our formal model.

5.4.1 Effective Interface

In our framework, the interface of a software unit *x* may depend on the interface of another software unit *y*. For any compilation uses the interface of *x*, the interface of *y* should also be included. The syntax of C and C++ also require that the interface of *y* should appear before the interface of *x*.

The effective interface of a software unit *x*, denoted *EI(D, x)*, contains the interface of *x* itself and all the other interfaces that the interface of *x* depends on. We calculate the effective interface of a software unit by the following algorithm:

Algorithm 5.1 Effective Interface

```
1: FUNCTION EI(D: software unit database, x: software Unit): list of source code fragments;
2:     VAR
3:         visited : set of software unit;
4:         effective_interface : list of source code fragments;
5:
6:     PROCEDURE EITraversal(x: software Unit);
7:         VAR
8:             y: software unit;
9:         BEGIN
10:            visited := visited + {x.ID};
11:            for each y ∈ TUsesInterface(D, x)
12:                if (y.ID ∉ visited)
13:                    EITraversal(y);
14:            effective_interface := effective_interface + {x.INF};
15:        END
16:
17:    BEGIN
18:        IF (x ∉ D)
19:            RETURN( [ ] );
20:        visited := {};
21:        effective_interface := [];
22:        EITraversal(x);
23:        RETURN(effective_interface);
24:    END;
```

We can prove that there is no duplication in the list calculated by Algorithm 5.1:

Property 5.1 No duplication in effective interfaces

Assume that the interfaces of all software units are distinct. Then there is no duplication in the effective interface EI(D, x).

Proof:

Algorithm 5.1 keeps the software units it visited in the variable visited (line 10) and avoids visiting the same software unit twice (line 12). Since the interface of a software unit is inserted into effective_interface only when it is visited by EITraversal (line 14), we know that the interface of a software unit cannot be inserted into effective_interface more than once.

■

The effective interface of a software unit is determined by the t_uses_interface links. The interface of a software unit *y* appears in the effective interface of *x* if and only if there is a path of t_uses_interface links from *x* to *y*.

Property 5.2 Membership of interfaces in effective interfaces

The interface of y appears in the effective interface of x if and only if there is a path of t_uses_interface links from x to y.

$$\forall D \in \Delta \ \ \forall x, y \in D \qquad y.\mathrm{INF} \in EI(D, x) \Leftrightarrow y \in TUsesInterface^{*}(D, x) \tag{5.24}$$

Proof:

=>)

Suppose *y.INF* is in *EI(D,x)*. The only place *y.INF* may be appended to *EI(D,x)* is at line 14, which is executed whenever the procedure EITraversal is called. But we know that the procedure EITraversal performs a depth-first traversal from *x* by following the t_uses_interface links, and it visits a software unit only when there is a path of t_uses_interface links from *x*. Therefore, there must be a path of t_uses_interface links from *x* to *y*.

<=)

In Algorithm 5.1, line 12 and line 13 make sure that if a software unit *x* is visited by EITraversal, then all the software units *x* points t_uses_interface links to will also be visited. Therefore, if *x* is visited by EITraversal and there is a path of t_uses_interface links from *x* to *y*, then every software unit on the path, including *y*, will be visited by EITraversal. Since the interface of a software unit will be appended to *EI(D, x)* when it is visited by EITraversal (line 14), we know that *y.INF* must appear in *EI(D,x)*.

■

In our model, a t_uses_interface link from *x* to *y* implies that the interface of *x* depends on the interface of *y*. Algorithm 5.1 guarantees that whenever the interface of *x* is included in an effective interface, then the interface of *y* is also included.

Property 5.3 t_uses_interface links and membership of effective interface

If there is a path of t_uses_interface links from x to y, then y.INF appears in any EI(D, z) that contains x.INF:

$$\forall D \in \Delta \;\; \forall x, y, z \in D \qquad x.\mathrm{INF} \in EI(D, z) \wedge y \in TUsesInterface^*(D, x) \Rightarrow y.\mathrm{INF} \in EI(D, z) \tag{5.25}$$

Proof:

$y.\mathrm{INF} \in EI(z) \wedge x \in TUsesInterface^*(D, y)$

$\Rightarrow y \in TUsesInterface^*(D, z) \wedge x \in TUsesInterface^*(D, y)$ (Property 5.2)

$\Rightarrow x \in TUsesInterface^*(D, z)$ (Equation (5.9))

$\Rightarrow x.\mathrm{INF} \in EI(z)$ (Property 5.2)

■

The ordering of elements in the list calculated by Algorithm 5.1 is partly determined by the t_uses_interface links. Basically, if there is a t_uses_interface link from *x* to *y*, then the interface of *y* will appear before the interface of *x*. In other words, the interface of *x* will appear after the interface it depends on. This property, however, is not guaranteed if there are cycles of t_uses_interface links.

Property 5.4 Ordering of interfaces in effective interfaces

If there is a t_uses_interface path from x to y, but no path of t_uses_interface links from y back to x, then y.INF appears before x.INF in any EI(D, z) that contains x.INF:

$$\begin{aligned}&\forall D \in \Delta \;\; \forall x, y, z \in D \\ &\quad x.\mathrm{INF} \in EI(z) \wedge y \in TUsesInterface(D, x) \wedge x \notin TUsesInterface^*(D, y) \\ &\Rightarrow y.\mathrm{INF} \in EI(D, z) \wedge (INDEX(EI(D, z), y.\mathrm{INF}) < INDEX(EI(D, z), x.\mathrm{INF}))\end{aligned} \tag{5.26}$$

Proof:

The procedure EITraversal in Algorithm 5.1 performs a post-order, depth-first traversal on the directed graph formed by software units in *TUsesInterface*(D, x)* and their t_uses_interface links. The t_uses_interface links followed by EITraversal form a spanning tree in the graph. Relative to the spanning tree, the t_uses_interface links in the underlying graph can be classified into tree edges, forward edges, back edges, and cross edges [2]:

(i) If <*x*, *y*> is a tree edge, then *y* has not been visited when *x* is visited. EITraversal will visit *y* (line 13) and append *y.INF* into *EI(D,x)* before appending *x.INF* at line 14. Therefore, *y.INF* will appear before *x.INF*.

(ii) If <*x, y*> is a forward edge, then the visiting of *y* is already completed when *x* is visited. Therefore, *y.INF* is already in *EI(D, x)* when *x.INF* is appended to *EI(D, x)* at line 14.

(iii) If <*x, y*> is a back edge, then there is a path of t_uses_interface links from *y* to *x* that are used as tree edges in the traversal. But this is contradictory to our precondition that there is no path of t_uses_interface links from *y* to *x*. Therefore, <*x, y*> cannot be a back edge.

(iv) If <*x, y*> is a cross edge, then according to Lemma 5.6 of [2], we know that the visiting of y is already completed when *x* is visited. Therefore, *y.INF* is already in *EI(D, x)* when *x.INF* is appended to *EI(D, x)* at line 14.

■

5.4.2 Effective Globals

Similarly to the calculation of effective interface, we can calculate the effective globals of a software unit by Algorithm 5.2. The effective globals of a software unit *x* contains *x.G*, and all the globals *x.G* depends on. Any software unit that points a uses_globals link or a t_uses_globals link to *x* gets the effective interface of *x* for compilation.

Algorithm 5.2 Effective Globals

```
1: FUNCTION EG(D: software unit database, x: software unit): list of source code fragment;
2:     VAR
3:         visited : set of software unit;
4:         result : list of source code fragment;
5:
6:     PROCEDURE EGTraversal( x: software unit);
7:         VAR
8:             y: software unit;
9:         BEGIN
10:            visited := visited + {x.ID};
11:            for each y∈ TUsesGlobals(D, x)
12:                if (y.ID ∉ visited)
13:                    EGTraversal(y);
14:            result := result + {x.INF};
15:        END
16:
17:    BEGIN
18:        IF (x ∉ D)
19:            RETURN( [ ] );
20:        visited := {};
21:        result := [];
22:        EGTraversal(x);
23:        EG := result;
24:    END;
```

Since the definition of effective globals is symmetric to that of effective interface, we can derive properties that are similar to those of effective interface. These properties are listed below. We omit the proofs because they are similar to their counterparts for the effective interface.

Property 5.5 No duplication in effective globals

Assume that the interfaces of all software units are distinct. Then there is no duplication in the effective globals EG(D, x):

Property 5.6 Membership of globals in effective globals

The globals of y appears in the effective globals of x if and only if there is a path of t_uses_globals links from x to y:

$$\forall D \in \Delta \;\; \forall x, y \in D \qquad y.\mathrm{G} \in EG(D, x) \Leftrightarrow y \in TUsesGlobals^*(D, x) \tag{5.27}$$

Property 5.7 t_uses_globals links and membership of effective globals

If there is a path of t_uses_globals links from x to y, then y.G appears in any EG(z) that contains x.G:

$$\forall D \in \Delta \;\; \forall x, y, z \in D \qquad x.\mathrm{G} \in EG(D, z) \wedge y \in TUsesGlobals^*(D, x) \Rightarrow y.\mathrm{G} \in EG(D, z) \tag{5.28}$$

Property 5.8 Ordering of globals in effective globals

If there is a t_uses_globals link from x to y, but no path of t_uses_globals links from y back to x, then y.G appears before x.G in any EG(D, z) that contains x.INF:

$$\begin{aligned} &\forall D \in \Delta \;\; \forall x, y, z \in D \\ &\quad x.\mathrm{G} \in EG(z) \wedge y \in TUsesGlobals(D, x) \wedge x \notin TUsesGlobals^*(D, y) \\ &\Rightarrow y.\mathrm{G} \in EG(D, z) \wedge (INDEX(EG(D, z), y.\mathrm{G}) < INDEX(EG(D, z), x.\mathrm{G})) \end{aligned} \tag{5.29}$$

5.4.3 Effective Source

When compiling a software unit, we need not only its implementation but also the interfaces and globals it depends on. The *effective source* of a software unit *x* is the list that contains all the source code segments required by the compilation of *x*. It can be calculated by Algorithm 5.3:

Algorithm 5.3 Effective Source

```
1: FUNCTION ES(D: software unit database, x: software unit): list of source code fragment;
2:     VAR
3:         effective_source: list of source code fragment;
4:         s: software unit;
5:         t: source code fragment;
6:     BEGIN
7:         effective_source:= [];
8:         FOR EACH s ∈ UsesInterface(D, x) DO
9:             FOR EACH t ∈ EI(D, s) DO
10:                if (t ∉ effective_source)
11:                    effective_source:= effective_source + [t];
12:        FOR EACH s∈ UsesGlobals(D, x) DO
13:            FOR EACH t ∈ EG(D, s) DO
14:                if (t ∉ effective_source)
15:                    effective_source:= effective_source + [t];
16:        effective_source:= effective_source + [x.INF, x.IMP];
17:    END;
```

The effective source of a software unit *x* contains the implementation of *x*, a set of interfaces, and a set of globals. It is easy to show that all the elements in the list calculated by Algorithm 5.3 are distinct:

Property 5.9 No duplication in effective source

There is no duplication in the effective source ES(D, x).

Proof:

This is trivial, since lines 10-11 and 14-15 checked the redundancy before insertion.

■

The effective source of a software unit *x* should include the interface of *y* if and only if the implementation of *x* depends on the interface of *y*. If we assume that the implementation of a software unit always depends on its own interface, which is usually the case in C and C++, then this dependency occurs in the following four cases:

- x and y are the same software unit.
- The implementation of x directly depends on the interface of y. In this case, there will be a uses_interface link from x to y.
- The implementation of x depends on the interface of z, and the interface of z depends on the interface of y. In this case, there will be a uses_interface link from x to z, and a path of t_uses_interface links from z to y.

- The interface of x depends on the interface of y. In this case, there will be a path of t_uses_interface links from x to y.

These four cases can be summarized as Theorem 5.1:

Theorem 5.1 Membership of interfaces in effective sources

Assume that the interfaces of all software units are distinct. Then the interface of y is in the effective source of x if and only if x equals to y or x uses the interface of a software unit that has a path of t_uses_interface *links to y.*

$$\forall x, y \in D \quad y.\mathrm{INF} \in ES(D, x) \Leftrightarrow (x = y) \vee \exists z \in D\,[z \in UsesInterface(D, x) \wedge y \in TUsesInterface^*(D, z)] \tag{5.30}$$

Proof:

=>)

In Algorithm 5.3, the only places an interface may be inserted into *ES(D, x)* are lines 11 and 16. Line 16 inserts the interface of *x* itself. The condition for line 11 to be executed is that there is a software unit $z \in UsesInterface(D,x)$ (line 8) and a source code fragment $t \in EI(D, z)$ (line 9). Therefore, we know that

$$\forall x, y \in D \qquad t \in ES(D, x) \Rightarrow (t = x.\mathrm{INF}) \vee \exists z \in D\,[z \in UsesInterface(D, x) \wedge t \in EI(D, z)]$$

Replacing t with *y.INF*, we get:

$$\forall x, y \in D \quad y.\mathrm{INF} \in ES(D, x) \Rightarrow (y.\mathrm{INF} = x.\mathrm{INF}) \vee \exists z \in D\,[z \in UsesInterface(D, x) \wedge y.\mathrm{INF} \in EI(D, z)]$$

Since the interfaces of all software units are distinct, we have:

$$\forall x, y \in D \qquad y.\mathrm{INF} \in ES(D, x) \Rightarrow (y = x) \vee \exists z \in D\,[z \in UsesInterface(D, x) \wedge y.\mathrm{INF} \in EI(D, z)]$$

Then by Property 5.2 we know that:

$$\forall x, y \in D \quad y.\mathrm{INF} \in ES(D, x) \Rightarrow (x = y) \vee \exists z \in D\,[z \in UsesInterface(D, x) \wedge y \in TUsesInterface^*(D, z)]$$

<=)

The right-hand side of this theorem is a conjunction. To show that its first term $x = y$ implies the left-hand side of this theorem is trivial, since x.INF will be inserted at line 16 when $x = y$.

The second term of the right-hand side also implies the left-hand side. If there is a software unit $z \in UsesInterface(D,x)$ and a source code fragment $t \in EI(D, z)$ (line 8, 9), then *t* is either already in or will be inserted into *ES(D, x)* (lines 10-11). (*ES(D, x)* is kept in the variable *effective_source*.) Therefore we know:

$$\forall x, y \in D \qquad \exists z \in D\,[z \in UsesInterface(D, x) \wedge t \in EI(D, z)] \Rightarrow t \in ES(D, x)$$

Replacing t with *y.INF,* we get:

$$\forall x, y \in D \qquad \exists z \in D\,[\,z \in UsesInterface\,(D, x) \wedge y.\text{INF} \in EI\,(D, z)\,] \Rightarrow y.\text{INF} \in ES\,(D, x)$$

Then by Property 5.2 we know that

$$\forall x, y \in D \qquad \exists z \in D\,[\,z \in UsesInterface\,(D, x) \wedge y.\text{INF} \in UsesInterface^*\,(D, z)\,] \Rightarrow y.\text{INF} \in ES\,(D, x)$$

■

A t_uses_interface link from *x* to *y* implies that the interface of *x* depends on the interface of *y.* Therefore, if the effective source of some other software unit includes the interface of *x*, then the interface of *y* should also be included.

Lemma 5.1 t_uses_interface links and membership of effective source

If there is a path of t_uses_interface links from x to y, then x.INF appears in any ES(z) that contains y.INF.

$$\forall x, y, z \in D \qquad y.\text{INF} \in ES\,(z) \wedge x \in TUsesInterface^*\,(D, y) \Rightarrow x.\text{INF} \in ES\,(z) \tag{5.31}$$

Proof:

$y.\text{INF} \in ES\,(z) \wedge x \in TUsesInterface^*\,(D, y)$

$=> \ (\exists w \in D\,[\,w \in UsesInterface\,(D, x) \wedge y \in TUsesInterface^*\,(D, w)\,]\,) \wedge x \in TUsesInterface^*\,(D, y)$

(Theorem 5.1)

$=> \exists w \in D\,[\,w \in UsesInterface\,(D, x) \wedge (y \in TUsesInterface^*\,(D, w) \wedge x \in TUsesInterface^*\,(D, y))\,]$

(Reorganizing the equation)

$=> \exists w \in D\,[\,w \in UsesInterface\,(D, x) \wedge (x \in TUsesInterface^*\,(D, w))\,]$

(Equation (5.9))

$=> x.\text{INF} \in ES\,(z)$ (Theorem 5.1)

■

The ordering of elements in an effective source is partly determined by the t_uses_interface links. Basically, if there is a t_uses_interface link from *x* to *y* and the interface of *x* appears in some effective source *ES(z)*, then the interface of *y* will appear in front of the interface of *x* in *ES(z)*. This property is important because a t_uses_interface link from *x* to *y* implies that the interface of *x* depends on the interface of *y.* In this case, most programming languages require the interface of *y* appear before the interface of *x* in the text fed to the compiler.

Theorem 5.2 Ordering of interfaces in effective sources

If there is a t_uses_interface links from x to y, but not a path of t_uses_interface links from y back to x, then y.INF appears before x.INF in any ES(D, z) that contains x.INF.

$$\begin{aligned}&\forall D \in \Delta \;\; \forall x, y, z \in D \\ &\quad x.\text{INF} \in ES(D, z) \wedge y \in TUsesInterface(D, x) \wedge x \notin TUsesInterface^*(D, y) \\ &\Rightarrow y.\text{INF} \in ES(D, z) \wedge (INDEX(ES(D, z), y.\text{INF}) < INDEX(ES(D, z), x.\text{INF}))\end{aligned} \tag{5.32}$$

Proof:

In Algorithm 5.3, the only place *x.INF* can be inserted into *ES(D, z)* is at line 11. But for line 11 to be executed, *x.INF* must be in *EI(D, s)* for some *s* in *UsesInterface(D, z)* (lines 8 and 9). That is:

$$x.\text{INF} \in ES(D, z) \Rightarrow \exists s\,(s \in UsesInterface(D, z) \wedge x.\text{INF} \in EI(D, s))$$

Substituting this equation into the left-hand side of (5.32), we have

$$\exists s\,[s \in UsesInterface(D, z) \wedge x.\text{INF} \in EI(D, s)] \wedge y \in TUsesInterface(D, x) \wedge x \notin TUsesInterface^*(D, y)$$

=>

$$\exists s\,[s \in UsesInterface(D, z)] \wedge x.\text{INF} \in EI(D, s) \wedge y \in TUsesInterface(D, x) \wedge x \notin TUsesInterface^*(D, y)$$

=>

$$\exists s\,[s \in UsesInterface(D, z)] \wedge y.\text{INF} \in EI(D, s) \wedge (INDEX(EI(D, z), y.\text{INF}) < INDEX(EI(D, z), x.\text{INF}))$$

(Property 5.4)

This means *y.INF* appears before *x.INF* in *EI(D, s)* for some *s* in *UsesInterface(D, z)*. Consequently, line 10 will be executed with *t = y.INF* before *x.INF* is inserted to *ES(D, z)*. When line 10 is executed, if *y.INF* is already in *ES(D, z)*, then *y.INF* will appear before *x.INF* in *ES(D, z)*; if *y.INF* is not in *ES(D, z)*, then line 11 will be executed and *y.INF* will still be inserted before *x.INF* is.

■

Again, since definitions of globals, uses_globals links and t_uses_globals are symmetric to the definitions of interface, uses_interface, and t_uses_interface, we can derive similar properties for globals:

Theorem 5.3 Membership of globals in effective sources

The globals of y is in the effective source of x if and only if x uses the globals of a software unit that has a path of t_uses_globals links to y:

$$\begin{aligned}&\forall D \in \Delta \;\; \forall x, y \in D \\ &\quad y.\text{G} \in ES(D, x) \Leftrightarrow \exists z \in D\,[(z \in UsesGlobals(D, x)) \wedge (y \in TUsesGlobals^*(D, z))]\end{aligned} \tag{5.33}$$

Proof:

This theorem can be proved in the same way as Theorem 5.1.

■

Theorem 5.4 Ordering of globals in effective sources

If there is a path of t_uses_globals links from x to y, but not one from y to x, then y.G appears before x.G in any ES(D, z) that contains y.G

$$\begin{aligned} &\forall D \in \Delta \;\; \forall x, y, z \in D \\ &\quad x.\mathrm{G} \in ES(D, z) \wedge z \in TUsesGlobals(D, y) \wedge y \notin TUsesGlobals^*(D, x) \\ &\Rightarrow y.\mathrm{G} \in ES(D, z) \wedge (INDEXES(ES(D, z), y.\mathrm{G}) < INDEX(ES(D, z), x.\mathrm{G})) \end{aligned} \tag{5.34}$$

Proof:

This theorem can be proved in the same way as Theorem 5.2.

■

5.4.4 Compilation, Linking, and Derived Objects

Our model makes no assumptions about the formats of source code, arguments, and result of compilations in the system building process. The only requirement is that, given the same list of source code segments and arguments, the compilation should always return the same derived object. Therefore, we model the compilation of source code by a function called the *compilation function*. A compilation function takes a list of source code segments and a list of arguments and returns a derived object.

Definition 5.12 Compilation Function

The compilation function Compile: $S \times A \rightarrow O$ *is a function where:*

- *S is a list of source code fragment;*
- *A is a set of Arguments;*
- *O is a Derived Object.*

Note that if multiple compilers are available, the specific compiler being used in a compilation should be modeled as an argument in the argument list. We do not use separate compilation functions for different compilers.

We call the result of compiling the effective source of a software unit *x* the *local derived object* of *x*.

Definition 5.13 Local Derived Object

Given a list of arguments A, the local derived object of a software unit x in a software unit database D, denoted LocalDerived(D, x, A), is the result of compiling the effective source of x with arguments A:

$$LocalDerived(D, x, A) \equiv Compile(ES(D, x), A) \tag{5.35}$$

And the set of derived objects generated by a subsystem is called the *derived objects* of the subsystem:

Definition 5.14 Derived Objects

Given a list of arguments A, the derived objects of a subsystem S(D, x), denoted as Derived(D, x, A), is the set of derived objects generated by the members of S(D, x).

$$Derived(D, x, A) \equiv \bigcup_{y \in UsesInterface(D, x)} LocalDerived(D, y, A) \tag{5.36}$$

Similarly to the definition of the compilation function, we can define a linking function to model the linking of derived objects in the system building process. Given a set of derived objects and a list of arguments, the linking function returns another derived object:

Definition 5.15 Linking Function

The linking function Link: $O \times A \rightarrow O'$ *is a function where:*

- *O is a set of derived objects;*
- *A is a set of arguments;*
- O' *is a derived object.*

Again, we make no assumptions about the format of derived objects and arguments used in a linking function. If multiple linkers are available, the specific linker used in a compilation should be specified as an argument to the linking function.

The executable of a software unit *x* is the result of linking the derived objects of the subsystem designated by *x*:

Definition 5.16 Executable

Given a list of arguments A, the executable of a subsystem S(D, x) a, denoted Exe(D, x, A), is the result of linking the derived objects of S(D, x) with A:

$$Exe(D, x, A) \equiv Link(Derived(D, x, A), A) \tag{5.37}$$

5.5 Atomic Operations

Our framework has five basic operations that can be used to change the status of a software unit database. They are *new*, *edit*, *delete*, *snapshot* and *revise*. These basic operations are the transactions of the software unit databases. Each of them is atomic; that is, they either occur in their entirety or do not occur at all, and if they do occur, they do not interfere with each other. We model each of these operations as a function that takes a software unit database as its major input and returns another software unit database as its result.

5.5.1 New

The first atomic operation in our model is New. It adds a new active software unit to an software unit database. The new software unit contains empty interface, implementation, and globals, and has no links to other software units. It is has an identity that is different from any existing software unit in the software unit database.

In order to choose a unique identity for the new software unit, we assume that we have a function, *NewID:* $2^S \to I$, that takes a software unit database *D* and generates an identity that is different from the identity of any software unit in *D*. In other words, NewID satisfies the following proposition:

$$\forall D \in 2^{\Sigma} \quad \forall x \in D \qquad NewID(D) \neq x.\mathrm{ID} \tag{5.38}$$

Implementing such a function should not be a problem in existing operating systems or databases. In ObjectStore, which is used to implement our prototype system POEM, each object in a database is assigned a unique identity when created.

Algorithm 5.4 New: $2^{\Sigma} \to 2^{\Sigma}$

```
1: FUNCTION New(D: software unit database): software unit database;
2:     VAR
3:         new_su: software unit;
4:     BEGIN
5:         new_su:= (NewID(D), NULL, NULL, NULL, { }, { }, { }, { }, NULL, FALSE);
6:         RETURN(D U {new_su} );
7:     END
```

Applying the New operation on a consistent software unit database yields another consistent software unit database:

Property 5.10

If D is a consistent software unit database, then New(D) is also a consistent software unit database.

$$\forall D \in \Delta \qquad New(D) \in \Delta \tag{5.39}$$

Proof:

To prove that *New(D)* is consistent, we have to show that all the software units it contains have unique identity and there is no dangling link.

(i) Unique identity.

This is trivial because the identity of the new software unit is generated by NewID(D), which by definition generates an identity that is different from any existing software unit in *D*.

(ii) No dangling links.

This is also trivial because New sets the links of the new software unit to be empty sets, and does not modify existing software units.

■

It is obvious that the New operation does not modify any fixed software unit in a software unit database.

Property 5.11 New does not change fixed software units

For any fixed software unit x in a software unit database D, x remains unchanged in New(D).

$$\forall D \in \Delta \;\; \forall x \in D \qquad (x.\mathrm{FX} = TRUE) \Rightarrow \exists x' \in New(D)\; [x' = x] \tag{5.40}$$

Proof:

This is trivial because New does not modify any existing software unit in *D*.

■

And it is not hard to show that the New operation does not create any cycle of t_uses_interface links or t_uses_globals links:

Property 5.12 New does not create t_uses_interface cycles or t_uses_globals cycles

If a consistent software unit database D is free of t_uses_interface cycles, then New(D) is also free of t_uses_interface cycles

$$\forall D \in \Delta \qquad TUI\_CYCLE\_FREE(D) \Rightarrow TUI\_CYCLE\_FREE(New(D)) \tag{5.41}$$

If a consistent software unit database D is free of t_uses_globals cycles, then New(D) is also free of t_uses_globals cycles

$$\forall D \in \Delta \qquad TUG\_CYCLE\_FREE(D) \Rightarrow TUG\_CYCLE\_FREE(New(D)) \tag{5.42}$$

Proof:

This is trivial because New does not add any t_uses_interface link or t_uses_globals link to *D*.

■

5.5.2 Edit

The Edit operation modifies the content of an existing software unit. It can change the interface, implementation, globals, uses_interface links, t_uses_interface links, uses_globals links, or t_uses_globals links of a software unit. The other attributes of a software unit – the identity, version predecessor pointer and fixed flag – cannot be changed by the Edit operation.

We model the Edit operation as a function that takes a software unit database *D* and two software units *x* and *x′*. The software unit *x* should be a member of *D*, and *x′* is the new value for *x*. Edit replace *x* with *x′* if all the following conditions are satisfied:

- *x* is in *D*.
- The identity of *x′* is the same as the identity of *x*.
- The version predecessor of x′ is the same as that of *x*.
- *x* is not fixed.
- *x′* is not fixed.

If any one of these conditions is not true, then Edit becomes a null operation and returns *D* without any modification.

Algorithm 5.5 Edit: $2^{\Sigma} \times \Sigma \times \Sigma \rightarrow 2^{\Sigma}$

```
1: FUNCTION Edit(D: software unit database, x: software unit, x′: software unit): software unit database;
2:     BEGIN
3:         IF (x ∉ D) ∨ (x.FX = TRUE) ∨ (x′.FX = TRUE) ∨ (x.ID ≠ x′.ID) ∨ (x.VP ≠ x′.VP)
4:             RETURN(D);
5:         ELSE IF (there is an identity in (x'.NUI ∪ x'.TUI ∪ x'.NUG ∪ x'.TUG) that does not correspond to
6:                 the identity of any software unit in D)
7:             RETURN(D);
8:         ELSE
9:             RETURN(D - {x′} U {x});
10:    END
```

If the Edit operation operates on a consistent software unit database, then the resulting software unit database is still consistent:

Property 5.13

If D is a consistent software unit database, then Edit(D, x, x′) is also a consistent software unit database:

$$\forall D \in \Delta \;\; \forall x, x' \in \Sigma \qquad Edit(D, x, x') \in \Delta \tag{5.43}$$

Proof:

To prove that *Edit(D, x, x′)* is consistent, we have to show that all the software units it contains have unique identity and there is no dangling link.

(i) Unique identity.

Since *x* is replaced by *x′* only if they have the same identity (line 3), the software units in *Edit(D, x, x′)* will still have unique identities.

(ii) No dangling link.

This is trivial because lines 5 through 7 make an explicit check to ensure this property.

■

The Edit operation does not modify any fixed software unit in a software unit database.

Property 5.14

For any fixed software unit y in a software unit database D, y remains unchanged in Edit(D, x, x′):

$$\forall D \in \Delta \;\; \forall y \in D \;\; \forall x, x' \in \Sigma \qquad (y.\mathrm{FX} = TRUE) \Rightarrow \exists y' \in Edit(D, x, x') \; [y' = y] \tag{5.44}$$

Proof:

This is trivial because if *x* is fixed, then *Edit(D, x, x′)* is the same as *D* (lines 3 and 4).

■

5.5.3 Delete

The Delete operation is used to remove an active software unit from a software unit database. If the software unit being removed is referred to by other software units, then the Delete operation becomes a null operation.

We model the Delete operation as a function that takes a software unit database *D* and a software unit *x* as its argument. It removes *x* from *D* if *x* is an active software unit in *D* and *x* is not referred to by any software unit in *D*.

Algorithm 5.6 Delete: $2^{\Sigma} \times \Sigma \rightarrow 2^{\Sigma}$

```
1: FUNCTION Delete(D: software unit database, x: software unit): software unit database;
2:     BEGIN
3:         IF (x ∉ D) ∨(x.FX ≠ FALSE)
4:             RETURN(D);
5:         ELSE IF there is a y in D - {x} such that x ∈ Uses(D,y)
6:             RETURN(D);
7:         ELSE
8:             RETURN(D - {x});
9:     END
```

Applying the Delete operation on a consistent software database unit yields another consistent software unit database:

Property 5.15

If D is a consistent software unit database, then Delete(D, x) is also a consistent software unit database:

$$\forall D \in \Delta \quad \forall x \in \Sigma \qquad Delete\,(D, x) \in \Delta \tag{5.45}$$

Proof:

To prove that *Delete(D, x)* is a consistent software unit database, we have to show that all the software units it contains have unique identity and there is no dangling link.

(i) Unique identity.

This is trivial because Delete does not modify or create a new identity.

(ii) No dangling link.

This is also trivial because Delete does not modify or create new links, and it explicitly checks

if the software unit being deleted is being used by other software units (line 5 and line 6).

■

The Delete operation does not modify any fixed software unit:

Property 5.16 Delete does not modify any fixed software unit

For any fixed software unit y in a software unit database D, y remains unchanged in Delete(D, x).

$$\forall D \in \Delta \;\; \forall y \in D \;\; \forall x \in \Sigma \qquad (y.\mathrm{FX} = TRUE) \Rightarrow \exists y' \in Delete(D, x)\ [y' = y] \tag{5.46}$$

Proof:

This is trivial because line 3 explicitly insures that the software unit being deleted is not fixed.

■

And the Delete operation does not create any cycle of t_uses_interface links or t_uses_globals links:

Property 5.17 Delete does not create t_uses_interface cycles or t_uses_globals cycles

If a consistent software unit database D is free of t_uses_interface cycles, then Delete(D, x) is also free of t_uses_interface cycles

$$\forall D \in \Delta \;\; \forall x \in \Sigma \qquad TUI\_CYCLE\_FREE(D) \Rightarrow TUI\_CYCLE\_FREE(Delete(D, x)) \tag{5.47}$$

If a consistent software unit database D is free of t_uses_globals cycles, then Delete(D, x) is also free of t_uses_globals cycles

$$\forall D \in \Delta \;\; \forall x \in \Sigma \qquad TUG\_CYCLE\_FREE(D) \Rightarrow TUG\_CYCLE\_FREE(Delete(D, x)) \tag{5.48}$$

Proof:

This is trivial because New does not add any t_uses_interface link or t_uses_globals link to *D*.

■

5.5.4 Snapshot

The Snapshot operation creates a fixed version of an extended subsystem as the version predecessor of an existing active extended subsystem. We model it as a function that takes a software unit database *D* and a software unit *x* as its input, and returns a software unit database containing new software units that represent the snapshot of *XS(D, x)*.

Algorithm 5.7 Snapshot: $2^{\Sigma} \times \Sigma \rightarrow 2^{\Sigma}$

```
1: FUNCTION Snapshot(D: software unit database, x: software unit): software unit database;
2:     VAR
3:         D′:           software unit database;
4:         mod_set:      SET of software units;/* Software units that are modified. */
5:         mod_xs_rts:   SET of software units;/* Root software units of modified extended subsystems. */
6:         y, new_su:    software unit;
7:     BEGIN
8:         IF (x.FX ∨ (x ∉ D))
9:             RETURN(D);
10:        D′ := D;
11:        mod_set := { };
12:        FOR EACH y ∈ XS(D, x)
13:            IF Modified(D,y)          /* The definition of Modified will be discussed later. */
14:                mod_set := mod_set ∪ {y};
15:        mod_xs_rts := { };
16:        FOR EACH y ∈ XS(D, x)
17:            if (XS(D, y) ∩ mod_set ≠ { })
18:                mod_xs_rts := mod_xs_rts + {y};
19:        FOR EACH y ∈ mod_xs_rts    /* Create a snapshot for each software unit in mod_xs_rts. */
20:            BEGIN
21:                new_su := (NewID(D′), y.INF, y.IMP, y.G, { }, { }, { }, { }, y.VP, TRUE);
22:                D′ := D′ + {new_su};
23:                y.VP := new_su.ID;
24:            END
25:        FOR EACH y ∈ mod_xs_rts
26:            BEGIN                     /* The definition of ConnectSnapshot will be discussed later. */
27:                VPred(D,y).NUI :=  ConnectSnapshot(D, y, NUsesInterface(D, y));
28:                VPred(D, y).TUI := ConnectSnapshot(D, y, TUsesInterface(D, y));
29:                VPred(D, y).NUG := ConnectSnapshot(D, y, NUsesGlobals(D, y));
30:                VPred(D, y).TUG := ConnectSnapshot(D, y, TUsesGlobals(D, y));
31:            END
32:        RETURN(D′);
33:    END;
```

Algorithm 5.7 can be divided into four parts. In the first part, lines 8 and 9, it tests the validity of its arguments. If *x* is fixed or if *x* is not a member *D*, then this is an invalid Snapshot operation. Algorithm 5.7 returns *D* without making any modification.

In the second part, lines 10 through 18, Algorithm 5.7 finds all the software units in *XS(D, x)* that should be considered as modified and puts them in the set mod_xs_rts. Basically, a software unit *y* is considered modified if *XS(D, y)* contains a modified software unit (lines 15–18). At line 13 of Algorithm 5.7, we use the function Modified to test if a software unit is different from its version predecessor. This function is defined below.

(Algorithm 5.7 continued)

```
34:FUNCTION Modified(D: software unit database, x: software unit): BOOLEAN;
35:     BEGIN
36:          IF (x.FX = TRUE)
37:               RETURN FALSE;
38:          ELSE IF (x.VP = NULL)/* x is considered as modified if it has no version predecessor. */
39:               return TRUE;
40:          ELSE IF (x.INF ≠ VPred(D, x).INF) ∨ (x.IMP ≠ VPred(D, x).IMP) ∨ (x.G ≠ VPred(D, x).G)
41:                    ∨ (x.NUI ≠ VPred(D, x).NUI) ∨ (x.TUI ≠ VPred(D, x).TUI)
42:                    ∨(x.TUG ≠ VPred(D, x).TUG) ∨ (x.TUG ≠ VPred(D,x).TUG))
43:               RETURN TRUE;
44:          ELSE
45:               RETURN FALSE;
46:     END
```

In the third part, from line 19 to line 24, Algorithm 5.7 creates a new software unit for each software unit in mod_xs_rts. The newly created software units are snapshots of the ones in mod_xs_rts. They inherit their interface, implementation, and globals from the software units they are created from, but have no links pointing to other software units.

In the last part, from line 25 to line 31, Algorithm 5.7 sets the links of the software units created in the third part. This is done by calling the function ConnectSnapshots, which is defined as follows:

(Algorithm 5.7 continued)

```
47:FUNCTION ConnectSnapshots(D: software unit database, y: software unit,
48:                                   L: SET of software units): SET of identities;
49:     VAR
50:          result : SET of identities;
51:          z: software unit;
52:     BEGIN
53:          result := { };
54:          FOR EACH z∈ L
55:               IF (z.FX = TRUE)
56:                    result := result ∪ {z.ID}
57:               ELSE
58:                    result := result ∪ {VPred(D, z).ID}
59:     END
```

The last part of Algorithm 5.7 is a little complicated, but the basic idea is simple. Suppose we have a software unit x' that is created as the snapshot of x. If x has a link pointing to y and there is also a newly created snapshot of y, then we set up a corresponding link from x to the snapshot of y. On the other hand, if there is no newly created snapshot of y, then we set up a link from x to y.

The Snapshot operation does not change the consistency of a software unit database.

Applying the Snapshot operation on a consistent software unit database yields another consistent software unit database.

Property 5.18 A snapshot of a consistent software unit database is still consistent.

If D is a consistent software unit database, then Snapshot(D, x) is also a consistent software unit database:

$$\forall D \in \Delta \;\; \forall x \in \Sigma \qquad Snapshot\,(D, x) \in \Delta \tag{5.49}$$

Proof:

(i) Unique identity.

This is trivial because the identities of new software units are generated by *NewID(D′)* at line 21.

(ii) No dangling link.

Since Algorithm 5.7 does not delete any software unit, all the links that are not modified remain valid in *Snapshot(D, x)*. There are two places at which Algorithm 5.7 sets links of software units. The first is at line 23, where *y.VP* is set to the identity of the newly created software unit new_su. Since new_su is inserted into *D′* at line 22, we know that line 23 will not cause dangling links.

The other place where Algorithm 5.7 sets links of software units is from line 27 to line 30, where the function ConnectSnapshots is called to connect the software units created in the loop of line 19. In this function, the new links are set at lines 56 and 58. Line 56 cannot cause dangling links because *z* is a software unit in *D*, which will still exist in *D′*. Line 58 cannot cause dangling links either. The value of *VPred(D′, z)* is always a valid software unit because if *z* does not have a version predecessor in the original software unit database *D*, then *z* will be considered as modified (line 38) and a new software unit will be created as its version predecessor at line 21.

■

We also observe that the Snapshot operation does not modify any fixed software unit in a software unit database.

Property 5.19

For any fixed software unit y in a software unit database D, y remains unchanged in Snapshot(D, x):

$$\forall D \in \Delta \;\; \forall y \in D \;\forall x \in \Sigma \qquad (y.\mathrm{FX} = TRUE) \Rightarrow \exists y' \in Snapshot\,(D, x)\;\; [y' = y] \tag{5.50}$$

Proof:

In Algorithm 5.7, Snapshot modifies software units at two places. The first place is at line 23, where the version predecessor of *y* is set. Since *y* is in mod_xs_rts and we know that all members of mod_xs_rts are active software units, line 23 does not modify any fixed software units.

The other place Algorithm 5.7 modifies software units is between lines 27 and 30, where the links of the version predecessor of *y* are set. Since *y* is in mod_xs_rts and the version predecessor of every element in mod_xs_rts is set to a newly created software unit at line 23, we know that the version predecessor of *y* is always a newly created software unit. Therefore, lines 27 through 30 do not modify any existing software unit.

■

And the Snapshot operation does not create any cycle of t_uses_interface links or t_uses_globals links:

Property 5.20 Snapshot does not create t_uses_interface cycles or t_uses_globals cycles

If a consistent software unit database D is free of t_uses_interface cycles, then Snapshot(D, x) is also free of t_uses_interface cycles

$$\forall D \in \Delta \quad \forall x \in \Sigma \qquad TUI\_CYCLE\_FREE(D) \Rightarrow TUI\_CYCLE\_FREE(Snapshot(D, x)) \tag{5.51}$$

If a consistent software unit database D is free of t_uses_globals cycles, then Snapshot(D, x) is also free of t_uses_globals cycles

$$\forall D \in \Delta \quad \forall x \in \Sigma \qquad TUG\_CYCLE\_FREE(D) \Rightarrow TUG\_CYCLE\_FREE(Snapshot(D, x)) \tag{5.52}$$

Proof:

The only place Algorithm 5.7 may add t_uses_interface links to *D* is at line 28, where the function ConnectSnapshots is called to establish t_uses_interface links among newly created software units. That is also the only place Algorithm 5.7 can possibly create cycles of t_uses_interface links.

In ConnectSnapshots, line 56 and line 58 are the only two places links are added. Line 56 create a link to a fixed software unit that existed in the original *D*. Since Algorithm 5.7 does not modify or add t_uses_interface link of any software unit that is not created by it, the destination software unit of the link create at line 56 cannot have any path of links leading back to the source software unit of the link, which is newly created by Algorithm 5.7. Therefore, we know line 56 cannot create any cycle of t_uses_interface links.

Line 58 cannot create cycles of t_uses_interface links either. It creates a link from *VPred(D, y)*

to *VPred(D, z)* if and only if there is a link from *y* to *z* in the original software unit database D. Therefore, if line 58 creates a cycle of t_uses_interface links, then there must be a cycle of t_uses_interface links in *D*. Since this contradicts with the fact that there are no cycles of t_uses_interface links in *D*, we know that line 58 cannot create cycles of t_uses_interface links alone. Line 58 cannot create cycles of t_uses_interface links with existing t_uses_interface links in *D* or the links created by line 56 either: Line 58 sets links from software units that are newly created by Algorithm 5.7, but *D* cannot contain any link to these newly created software units, and line 56 does not create any such link either.

We can also prove that Snapshot does not create any cycle of t_uses_globals links with the same argument.

■

The necessary and sufficient condition for Algorithm 5.7 to create a snapshot for a software unit *y* is that *y* be a member of the extended subsystem that is being operated on, and that some members of the subsystem designated by *y* are modified.

Property 5.21

In Algorithm 5.7, a snapshot is created for a software unit y if and only if y is a member of XS(D, x) and some members of XS(D, y) are modified.

$$\forall D \in \Delta \;\; \forall x, y, z \in D \quad y \in XS(D, x) \wedge Modified(D, y) \Leftrightarrow \exists y' \in Snapshot(D, x) \; [\, (y'.\mathrm{ID} = y.\mathrm{ID}) \wedge (y'.\mathrm{VP} = y.\mathrm{VP}) \,] \tag{5.53}$$

Proof:

This is trivial because *x* is inserted in mod_xs_rts if and only if some members of *XS(D, x)* are modified (lines 17 and 18), and a snapshot is generated for *x* if and only if *x* is in mod_xs_rts (lines 19 and 21).

■

5.5.5 Revise

The Revise operation creates an active version of a extended subsystem as the version successor of an existing fixed extended subsystem. We model it as a function. Like the Snapshot operation, this function takes a software unit database *D* and a software unit *x*. *D* is the original software unit on which the Revise operation operates; *x* is the root software unit of the extended subsystem that is being revised.

The Revise operation, however, needs an additional argument to model the workarea that contains *x*. As discussed in section 4.3, the Revise operation is not propagated across the boundaries of workareas, and it creates new active versions only for software units in the local workarea. The additional argument for the containing workarea can be represented as a set of software units in our model.

Algorithm 5.8 Revise: $2^{\Sigma} \times \Sigma \times 2^{\Sigma} \rightarrow 2^{\Sigma}$

```
1: FUNCTION Revise(D: software unit database, x: software unit, W: set of software units)
2:                  : software unit database;
3:      VAR
4:          D′:             software unit database;
5:          y, new_su:      software unit;
6:          revise_set, new_su_set: set of software units;
7:      BEGIN
8:          IF (¬x.FX ∨ (x ∉ D))
9:              RETURN(D);
10:         D′ := D;
11:         revise_set := XS(D, x) ∩ W;
12:         new_su_set := { };
13:         FOR EACH y ∈ revise_set
14:             BEGIN
15:                 new_su := (NewID(D′), y.INF, y.IMP, y.G, { }, { }, { }, { }, y.ID, FALSE)
16:                 new_su_set := new_su_set + {new_su};
17:                 D′ := D′ + {new_su};
18:             END
19:         FOR EACH y ∈ new_su_set;
20:             BEGIN
21:                 y.NUI := ConnectRevisions(D, y, NUsesInterface(D, y), new_su_set);
22:                 y.TUI := ConnectRevisions(D, y, TUsesInterface(D, y), new_su_set);
23:                 y.NUG := ConnectRevisions(D, y, NUsesGlobals(D, y), new_su_set);
24:                 y.TUG := ConnectRevisions(D, y, TUsesGlobals(D, y), new_su_set);
25:             END
26:         RETURN(D′);
27:     END;
```

Algorithm 5.8 can also be divided into four parts. The first part, lines 8 and 9, tests the validity of the arguments. If *x* is not fixed or if *x* is not a member of *D*, then the function returns *D* immediately without making any modification.

The second part, line 11, finds the set of software units that should be revised. This is simply the intersection of the extended subsystem designated by *x* and the containing workarea W.

The third part, lines 12 through 18, creates a new active version for each software unit found in the second part. The new active software units are created with their interface, implementation, and globals copied from the software units from which they are created.

The last part, lines 19 through 27, connects the newly created software units. It calls the function ConnectRevisions defined below. Again, the algorithm for this part is a little complicated, but the basic idea is rather simple. For each new software unit x' created from an existing software unit x, if there is a link from x to y and a new version of y is created, then we set up a link from x' to the new version of y. If no new version is created for y, then we set up a link from x' to y. In the first case, y is a software unit in the local workarea; in the latter case, y is a software unit residing in some remote workarea.

```
28:FUNCTION ConnectRevisions(D: software unit database, x: software unit,
29:                                  L: SET of software units, N: SET of software units): SET of identities;
30:    VAR
31:         result : SET of identities;
32:         w, y: software unit;
33:    BEGIN
34:         result := { };
35:         FOR EACH w ∈ L
36:              IF (there is a z in N such that VPred(z) = w)
37:                   result := result ∪ {z.ID};
38:              ELSE
39:                   result := result ∪ {w.ID};
40:    END
```

The Revise operation does not change the consistency of a software unit database. Applying the Revise operation on a consistent software unit database yields another consistent software unit database.

Property 5.22 A revision of a software unit database is still a software unit database.

If D is a software unit database, then Revise(D,x,B) is also a software unit database:

$$\forall D \in \Delta \;\; \forall x \in D \;\; \forall B \in 2^{\Sigma} \qquad Revise\,(D, x, B) \in \Delta \tag{5.54}$$

Proof:

To prove that *Revise(D, x, W)* is a consistent software unit database, we have to show that all the software units it contains have unique identity and there is no dangling link.

(i) Unique identity.

The only place Algorithm 5.8 adds new software units into a software unit database is between lines 15 and 17. Since the identities of the new software units are generated by calling the function NewID, we know that all the new software units have unique identity.

(ii) No dangling link.

Since Algorithm 5.8 does not delete any software unit, all the links that are not modified remain valid in *Revise(D, x)*. Therefore, we only have to check the validity of newly created links. In Algorithm 5.8, links of newly created software units are established between lines 21 and 24 by calling the function ConnectRevisions, which in turn sets up links at lines 37 and 39. Line 37 cannot cause dangling links, because line 36 explicitly checks the validity of the link's destination. Line 39 cannot cause dangling links either. Since *w* is a software unit in *D* and Algorithm 5.8 does not delete any software unit, *w* will still exist in *Revise(D, x)*.

■

It is not hard to observe that the Revise operation does not modify any fixed software unit in a software unit database.

Property 5.23

For any fixed software unit x in a software unit database D, x remains unchanged in Revise(D, z, B):

$$\forall D \in \Delta \;\; \forall y \in D \;\; \forall x \in \Sigma \;\; \forall B \in 2^{\Sigma} \qquad (y.\mathrm{FX} = TRUE) \Rightarrow \exists y' \in Revise(D, x, B)\ [y' = y] \tag{5.55}$$

Proof:

In Algorithm 5.8, the only place Revise modifies software units is between lines 21 and 24, where the links of *y* are set. Since *y* is in new_su_set and all the elements of new_su_set are newly created (lines 15 and 16), we know that Revise does not modify any existing software unit.

■

And the Revise operation does not create any cycle of t_uses_interface links or t_uses_globals links:

Property 5.24 Revise does not create t_uses_interface cycles or t_uses_globals cycles

If a consistent software unit database D is free of t_uses_interface cycles, then Revise(D, x) is also free of t_uses_interface cycles

$$\forall D \in \Delta \;\; \forall x \in \Sigma \;\; \forall B \in 2^{\Sigma} \qquad TUI\_CYCLE\_FREE(D) \Rightarrow TUI\_CYCLE\_FREE(Revise(D, x, B)) \tag{5.56}$$

If a consistent software unit database D is free of t_uses_globals cycles, then Snapshot(D, x) is also free of t_uses_globals cycles

$$\forall D \in \Delta \;\; \forall x \in \Sigma \;\; \forall B \in 2^{\Sigma} \qquad TUG\_CYCLE\_FREE(D) \Rightarrow TUG\_CYCLE\_FREE(Revise(D, x, B)) \tag{5.57}$$

Proof:

The only place Algorithm 5.8 may add t_uses_interface links to *D* is at line 22, where the function ConnectRevisions is called to establish t_uses_interface links among newly created software units. That is thus the only place Algorithm 5.8 can possibly create cycles of t_uses_interface links.

In ConnectRevisions, line 37 and line 39 are the only two places links are added. Line 39 create a link to a software unit that exists in the original software unit database *D*. Since Algorithm 5.8 does not modify or add t_uses_interface link of any software unit that is not created by it, the destination software unit of the link create at line 39 cannot have any path of links leading back to the source software unit of the link, which is newly created by Algorithm 5.8. Therefore, we know that line 39 cannot create any cycle of t_uses_interface links.

Line 37 cannot create cycles of t_uses_interface links either. It creates a link from *y* to *z* if and only if there is a link from *VPred(y)* to *VPred(z)* in the original software unit database D. Therefore, if line 37 creates a cycle of t_uses_interface links, then there must be a cycle of t_uses_interface links in *D*. Since this contradicts with the fact that there are no cycles of t_uses_interface links in *D*, we know that line 37 cannot create cycles of t_uses_interface links alone. Line 37 cannot create cycles of t_uses_interface links with existing t_uses_interface links in *D* or the links created by line 39 either: Line 39 sets links from software units that are newly created by Algorithm 5.8, but *D* cannot contain any link to these newly created software units, and line 39 does not create any such link either.

We can also prove that Revise does not create any cycle of t_uses_globals links with the same argument.

■

5.5.6 Discussion

The five operations discussed in this section are the only basic operations that can be used to modify software unit databases in our framework. In actual implementations, there might be variants of these operations. For example, our prototype environment POEM supplies different interfaces for editing source code and editing links. They look very different to end users, but map to the same Edit operation in our formal model.

5.6 Successors of a Software Unit Database

If we apply any of the five operations discussed in the previous section to a software unit database *D*, we get another software unit database *D′*. We call *D′* an *immediate successor* of *D* and define *ISuccessors(D)* to be the set of all immediate successors of *D*.

Definition 5.17 Immediate successors set of a software unit database

The immediate successor set of a software unit database D, denoted ISuccessors(D), is the set of software unit databases that can be generated by applying New, Edit, Delete, Snapshot, or Revise on D:

$$ISuccessors(D) \equiv \{D' \mid \exists x, y \in \Sigma \ \exists B \in 2^{\Sigma} \quad (D' = New(D, x)) \vee (D' = Edit(D, x, y)) \vee (D' = Delete(D, x)) \vee (D' = Snapshot(D, x)) \vee (D' = Revise(D, x, B))\} \tag{5.58}$$

From the properties derived in the previous sections, we know that the immediate successor of a consistent software unit database is still a consistent software unit database:

Lemma 5.2 Immediate successors of a consistent software unit database are consistent.

The immediate successors of a consistent software unit database are consistent software unit databases:

$$\forall D \in \Delta \quad \forall D' \in ISuccessors(D) \qquad D' \in \Delta \tag{5.59}$$

Proof:

By definition, members of *ISuccessors(D)* are generated by applying one of the New, Edit, Delete, Snapshot, and Revise operations on *D*. From Property 5.10, Property 5.13, Property 5.15, Property 5.18 and Property 5.22, we know that applying these operations on a consistent software unit database yields another consistent software unit database. Therefore, all members of *ISuccessors(D)* are consistent software unit databases.

■

If we repeatedly apply operations on a software unit database *D*, we will still get a software unit base. We define *Successors(D)* to be the set of software unit databases that can be generated by repeatedly applying operations on *D*:

Definition 5.18 Successors of a software unit database

The successors set of a software unit database D, denoted Successors(D), is the set of software unit databases that can be generated by repeatedly applying New, Edit, Delete, Snapshot, or Revise on D:

$$Successors(D) \equiv \{X \mid (X = D) \vee \exists Y (X \in ISucessors(Y) \wedge (Y \in Successors(D)))\} \quad (5.60)$$

From Lemma 5.2, we can easily infer that if *D* is a consistent software unit database, then all members of *Successors(D)* are also consistent.

Theorem 5.5 Successors of a consistent software unit database are consistent.

The successors of a consistent software unit database are consistent software unit databases:

$$\forall D \in \Delta \quad \forall D' \in Successors(D) \qquad D' \in \Delta \quad (5.61)$$

Proof:

By induction, we can show that repeatedly applying New, Edit, Delete, Snapshot, and Revise on a consistent software unit database *D* yields another consistent software unit database:

Induction Base:

D is a consistent software unit database.

Induction Hypothesis:

Let D' be a member of $\in$ *Successors(D)*, which is generated by repeatedly applying New, Edit, Delete, Snapshot, and Revise on *D*. Assume that D' is consistent.

Induction Step:

From Lemma 5.2, we know that if D' is consistent, then any immediate successor of D', which is generated by applying New, Edit, Delete, Snapshot, or Revise on D', is also consistent.

■

If we start with an empty software unit database in our programming environment and use only the five basic operation discussed in the previous section, we will always get a consistent software unit database:

Property 5.25

All the successors of the empty software unit database {} are consistent:

$$\forall D' \in Successors(\{\}) \qquad D' \in \Delta \quad (5.62)$$

Proof:

According to Definition 5.2, a software unit database is consistent if each of its member has a unique identity and it contains no dangling links. Since the empty set {} satisfies both those conditions. we know it is a consistent software unit database.

According to Lemma 5.2, the successors of a consistent software unit database are also consistent. Therefore, we know that all the successors of {} are consistent.

■

From Property 5.23 and equation (5.3), we know that

$$Successors(\{\}) \subset \Delta \subset 2^{\Sigma} \tag{5.63}$$

Software unit databases in *Successors({})* are of our special interest, because they represent all the software unit databases that can be created by our basic operations. Members of *Successors({})* have several useful properties. Theorem 5.6 states that if a software unit database *D* is a member of *Successors({})*, then a software unit in *D* is fixed if it is used by another fixed software unit or if it is the version predecessor of another software unit.

Theorem 5.6

For any software unit database D in Successors({}), the following properties hold:
1. If a software unit x is the version predecessor of another software unit y, then x is fixed.

$$\forall D \in Successors(\{\}) \quad \forall x, y \in D \qquad (y.\mathrm{VP} = x.\mathrm{ID}) \Rightarrow (x.\mathrm{FX} = TRUE) \tag{5.64}$$

2. If a software unit x is used by a fixed software unit y, then x is also fixed.

$$\forall D \in Successors(\{\}) \quad \forall x, y \in D \qquad x \in Uses(D, y) \wedge (y.\mathrm{FX} = TRUE) \Rightarrow (x.\mathrm{FX} = TRUE) \tag{5.65}$$

Proof:

We prove this theorem by induction on *D*:

Induction Base:

When *D* = {}, clearly both properties hold.

Induction Hypothesis:

Let *D1* be any software unit database in Successors({}). Assume that both properties hold in *D1*.

Induction Step:

When *D* = *ISuccessors(D1)*, *D* is the result of applying one of the New, Edit, Delete, Snapshot, and Revise operations on *D1*. Therefore, we need to show that all those five operations retain

the two properties of this theorem.

(1) New:

This is trivial. Since New does not set the version predecessor or the fixed flag of any software unit, it retains the first property (5.64). Since New does not set any link either, it also retains the second property (5.65).

(2) Edit:

Edit retains the first property because it does not set the version predecessor or the fixed flag of any software unit. Edit retains the second property because it does not modify any fixed software unit and does not modify the fixed flags of any software unit.

(3) Delete:

Since Delete does not modify the fixed flag or the version predecessor of any fixed software unit, it retains the first property. Since Delete does not modify the fixed flag or the links of any fixed software unit, it also retains the second property.

(4) Snapshot:

In Algorithm 5.7, the only place Snapshot modifies the version predecessors of software units is at line 23, where the version predecessor of *y* is set to new_su. Since new_su is created as a fixed software unit (line 21), we know that the first property is retained.

In Algorithm 5.7, the only place Snapshot modifies the links of software units is in the function ConnectSnapshots. ConnectSnapshots retains the second property because if *y* is not fixed, then the version predecessor of *y* will be used instead of *y*. By the induction hypothesis, we know that the version predecessor of *y* is fixed.

(5) Revise:

To show that Revise retains the first property, we need to show that all members of *XS(D, x)* are fixed if *x* is fixed. By the induction hypothesis, we know that all the software units used by *x* are fixed, and then all the software units used by the software units used by *x* are fixed. With a simple induction, we can show that all the software units that can be reached from *x*, which is exactly the definition of *XS(D, x)*, are fixed.

In Algorithm 5.8, the only place Revise modifies the version predecessor of software units is at line 15, where the version predecessor of new_su is set to *y*. Since *y* is a member of revise_set and revise_set is a subset of *XS(D, x)*, *y* is a member of *XS(D, x)*. Since line 15 is executed only if *x* is fixed (lines 8 and 9), *y* must be fixed and the first property is retained.

In Algorithm 5.8, Revise modifies the links of software units at two places. The first place is at

line 15, where the links of new_su are set to the empty sets. Clearly this does not violate the second property. The second place is lines 21 through 24, where the links of *y* are set to the return value of ConnectRevisions. Since *y* is a member of new_su_sets (line 19) and all software units in new_su_set are active (lines 15 and 16), *y* must be active. Clearly setting the links of an active software unit will not violate the second property.

■

5.7 Properties of Fixed Software Units

Once created, a fixed software unit should not be changed, and the derived objects and executable it generates should remain constant over time. In this section we give the proofs of these properties.

Lemma 5.3 Fixed software units are constant.

For any fixed software unit x in a software unit database D ∈ Successors({}), x remains unchanged in the immediate successors of D.

$$\forall D \in Successors(\{\}) \;\; \forall x \in D \;\; D' \in ISuccessors(D) \\ (x.\mathrm{FX} = TRUE) \Rightarrow \exists x' \in D' \, [x' = x] \tag{5.66}$$

Proof:

By definition, *ISuccessors(D)* is the set of all software unit databases that can be generated by applying New, Edit, Delete, Snapshot, or Revise on *D*. From Property 5.11, Property 5.14, Property 5.16, Property 5.19 and Property 5.23, we know that a fixed software unit in *D* will not be modified in these operations. Therefore, *x* remains unchanged in *Successors(D)*.

■

With Lemma 5.3, we can show that the effective source of a fixed software unit remains constant in the successors of *{}*:

Lemma 5.4 The effective source of a fixed software unit is constant.

For any fixed software unit x in a software unit database D ∈ Successors({}), ES(D, x) remains unchanged in the immediate successors of D:

$$\forall D \in Successors(\{\}) \;\; \forall x \in D \;\; \forall D' \in ISuccessors(D) \\ (x.\mathrm{FX} = TRUE) \Rightarrow \exists x' \in D' \, [\, (x' = x) \wedge (ES(D', x') = ES(D, x)) \,] \tag{5.67}$$

Proof:

According to Algorithm 5.3, *ES(D, x)* contains a set of interfaces, a set of globals, and *x.IMP*.

From Theorem 5.1, we know that the interface of *y* is in *ES(D, x)* only if *y* can be reached from *x* via a path of uses_interface and t_uses_interface links. From Theorem 5.3, we know that the globals of *y* is in *ES(D, x)* only if *y* can be reached from *x* via a path of uses_globals and t_uses_globals links. Therefore, all the elements of *ES(D, x)* belongs to software units that can be reached from *x* via links (including *x* itself).

From the second property of Theorem 5.6, we know that if *x* is fixed and there is a link from *x* to *y*, then *y* must be fixed too. It is not hard to infer that if *x* is fixed and there is a path of links from *x* to *y*, then *y* must be fixed too.

Combining the above two facts, we know that all the elements of *ES(D, x)* belong to fixed software units. From Lemma 5.3, we know that all the fixed software units in *D* remain unchanged in the successors of *D*. Therefore, *ES(D, x)* remains unchanged in the successors of *D*.

■

And from Lemma 5.3 we can infer that the local derived objects of a fixed software unit remain constant in the successors of *{}*:

Lemma 5.5 The local derived objects of a fixed software unit are constant.

For any software unit x in a software unit database D ∈ Successors({}), the local derived objects of x remain unchanged in the immediate successors of D.

$$\forall D \in SuccessorSet(\{\}) \quad \forall x \in D \quad \forall D' \in ISuccessorSet(D) \quad \forall a \in \mathcal{A} \quad (x.\mathrm{FX} = TRUE) \Rightarrow \exists x' \in D' [\, (x' = x) \wedge LocalDerived(D', x', a) = LocalDerived(D, x, a) \,] \tag{5.68}$$

Proof:

$x.\mathrm{FX} = TRUE$

$\Rightarrow \exists x' \in D' [\, (x' = x) \wedge (ES(D', x') = ES(D, x)) \,]$ (Lemma 5.3)

$\Rightarrow \exists x' \in D' [\, (x' = x) \wedge (Compile(ES(D', x'), a) = Compile(ES(D, x), a)) \,]$

(since *Compile* is a function)

$\Rightarrow \exists x' \in D' [\, (x' = x) \wedge LocalDerived(D', x', a) = LocalDerived(D, x, a) \,]$

(Definition 5.13)

■

Theorem 5.7 The derived objects of a fixed software unit are constant.

For any software unit x in a software unit database D ∈ Successors({}), the derived objects of x remain unchanged in the immediate successors of D:

$$\forall D \in SucessorSet(\{\})\ \ \forall x \in D\ \ \forall D' \in ISuccessorSet(D)\ \ \forall a \in \mathcal{A} \quad (5.69)$$
$$(x.\mathrm{FX} = TRUE) \Rightarrow \exists x' \in D'\ [\ (x' = x) \wedge (Derived(D', x', a) = Derived(D, x, a))\ ]$$

Proof:

According to Definition 5.14, *Derived(D, x, a)* is the union of the locally derived objects of all the software units that can be reached from *x* via uses_interface and t_uses_interface links. From the second property of Theorem 5.6, we can infer that all the software units that can be reached from a fixed software unit via links are fixed too. From Theorem 5.7, we know that the local derived objects of a fixed software unit are constant. Combining all these facts, we know that the derived objects of a fixed software unit are constant.

■

Theorem 5.8 The executable of a fixed software unit is constant.

For any fixed software unit x in a software unit database D ∈ Successors({}), the executable of x remains unchanged in the immediate successors of D:

$$\forall D \in SucessorSet(\{\})\ \ \forall x \in D\ \ D' \in ISucessorSet(D)\ \ \forall a \in \mathcal{A} \quad (5.70)$$
$$(x.\mathrm{FX} = TRUE) \Rightarrow \exists x' \in D'\ [Exe(D, x, a) = Exe(D', x, a)\ ]$$

Proof:

$(x.\mathrm{FX} = TRUE) \wedge D' \in ISucessorSet(D)$

$=> \ Derived(D, x, a) = Derived(D', x, a)$ (Theorem 5.7)

$=> Link(Derived(D, x, a)) = Link(Derived(D', x, a))$ (since *Link* is a function)

$=> Exe(D, x, a) = Exe(D', x, a)$ (Definition 5.16)

■

5.8 Versional Consistency

In a programming environment, there may be multiple versions of the same source code segment. But when building the executable of a program, we should use at most one version of each source code segment. If multiple versions of the same source code segment are used, the result is usually erroneous. Either the linker will complain about the redundancy, or a version of

the source code segment is selected randomly.

We say that the configuration of a program is *versionally consistent* if it contains at most one version of each source code segment. In this section, we discuss how versional consistency can be defined in our model and how it is affected by different operations.

We start by defining the basic terminology. A software unit *x* is a *version ancestor* of another software unit *y* if there is a path of version predecessor links from *y* to *x*. The version predecessor links are set up by the Snapshot operations and the Revise operations. If there is a version predecessor pointer from *y* to *x*, then *y* is logically derived from *x*.

Definition 5.19 Version Ancestors

In any software unit database D, the version ancestors of a software unit x, VA(D, x), are software units that x can reach via a path of version predecessor links:

$$VA(D, y) \equiv \{ y \mid (VPred(D, x) = y) \vee \exists z \in D [(VPred(D, x) = z) \wedge y \in VA(D, z)] \} \quad (5.71)$$

Two software units are of the same version family if they can be traced back to a common version ancestor. They are both versions of the same "logical software unit."

Definition 5.20 Same Version Family

Two software units x and y are of the same version family, denoted as SVF(x, y), if and only if x is equal to y or x and y have a common version ancestor:

$$SVF(x, y) \equiv (x = y) \vee \exists z [z \in VA(D, x) \wedge z \in VA(D, y)] \quad (5.72)$$

Version family is a notion in our model, not a unit of management. In some configuration management systems, like Adele, system models are specified in terms of version families, and version families are first-class entities that can be directly operated by users. We do not support operations to handle version families because our system models are effectively specified by the links between individual versions of software units.

In a software unit database that belongs to *Successors({})*, software units of the same version family form a tree with their version predecessor pointers.

Property 5.26 Software units and their version predecessor pointer form a forest

In any software unit database D ∈ Successors({}), software units and their version predecessor pointers form a forest (i.e. a set of trees).

Proof:

To prove this property, we first show that there can be no cycles of version predecessor pointers. Clearly the New, Edit and Delete operations cannot cause any cycle of version predecessor pointers, because they do not create or modify any version predecessor pointer.

The Revise operation cannot form any cycle because it only sets the version predecessor pointers of active software units. By the first property of Theorem 5.6, we know that no version predecessor pointer leads to an active software unit.

The Snapshot operation sets the version predecessors of the fixed software units it creates, but this does not cause any cycle either. Because the only version predecessor pointers leading to these newly created software units are those of active software units, and no version predecessor pointer leads to those active software units, we know there are no cycles of version predecessor pointers.

Since each node (i.e. a software unit) has at most one parent (i.e. its version predecessor pointer), we know the graph is a set of trees.

■

Clearly, in any software unit database that belongs to *Successors({ })*, the predicate *SVF* defines an equivalence relation among software units:

Property 5.27 Same Version Family defines an equivalence relation

In any software unit database $D \in Successors(\{\})$, the relation defined by the Same Version Family predicate is an equivalence relation.

Proof:

To show that the relation is an equivalence relation, we have to show that it is reflexive, symmetric, and transitive.

Reflexive:

$$SVF(x, x) = (x = x) \vee \exists z\,[z \in VA(D, x) \vee z \in VA(D, x)] = TRUE$$

Symmetric:

$$SVF(x, y) = (x = y) \vee \exists z\,[z \in VA(D, x) \vee z \in VA(D, y)] = (y = x) \vee \exists z\,[z \in VA(D, y) \vee z \in VA(D, x)] = SVF(y, x)$$

Transitive:

If *SVF(x, y)* and *SVF(y, z)* are true, then *x* and *y* have a common version ancestor and *y* and *z* also have a common version ancestor (Definition 5.20). Therefore, *x*, *y*, and *z* must belong to the same connected component. From Property 5.26, we know that the connected component must be a tree and that the root of the tree is a common ancestor of *x* and *z*. Then since *x* and *z* have a com-

mon version ancestor, *SVF(x, z)* is true (Definition 5.20).

■

Since system building in our framework is carried out in terms of subsystems and extended subsystems, we define version consistency in our model in those terms.

Definition 5.21 Versional Consistency

*An subsystem S(D, x) is **versionally consistent** if and only if every element of S(D, x) belongs to a different version family:*

$$VConsistent(S(D,x)) \equiv \forall x, y \in S(D,x) \quad [(x = y) \vee \neg SVF(x,y)] \tag{5.73}$$

*Similarly, an extended subsystem XS(D, x) is **versionally consistent** if and only if every element of XS(D, x) belongs to a different version family:*

$$VConsistent(ES(D,x)) \equiv \forall x, y \in S(D,x) \quad [(x = y) \vee \neg SVF(x,y)] \tag{5.74}$$

When discussing versional consistency, we do not distinguish uses_globals links from uses_interface links, and there is no need to treat subsystems and extended subsystems separately. In the remaining of this section we discuss only the versional consistency of subsystems. Similar properties for extended subsystems can always be derived in the same way.

Some operations may change the versional consistency of subsystems, and some others do not. The Edit operation may change the versional consistency of a subsystem, because it allows us to set the links of active software units arbitrarily. For example, we can create a versionally inconsistent subsystem as illustrated in Figure 5-1 by Edit operations.

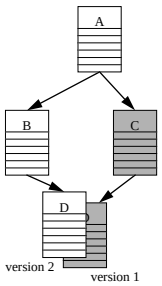


Figure 5-1: A versionally inconsistent subsystem.

One way to avoid this problem is to modify Edit so that it disallows any action that results in versionally inconsistent subsystems. But implementing this constraint will be expensive in our framework. For example, to detect that adding the link from *B* to version 2 of *D* in Figure 5-1 causes an versionally inconsistent subsystem, we have to know that *B* is used by *A*. But since links

in our framework are directional, we have to check all software units in a software unit database to know which ones use *B*.

It is not hard to see that the New operation and the Delete operation cannot change the versional consistency of a subsystem, because the software units they create or delete cannot be members of other existing subsystems. The Snapshot operation cannot change the versional consistency of a subsystem either. But the proof is more complicated:

Theorem 5.9 The Snapshot operation retains versional consistency.

In any software unit database $D \in Successors(\{\})$, if a subsystem is versionally consistent, then its snapshot is also versionally consistent.

$$\forall D \in Sucessor(\{\}) \quad \forall x \in D \\ VConsistent(XS(D, x)) \wedge (D' = Snapshot(D, x)) \Rightarrow VConsistent(XS(D', Pred(D', x))) \tag{5.75}$$

Proof:

Let $D' = Snapshot(D, x)$. The software units in D' can be partitioned into four sets (see Figure 5-2):

Set 1: Fixed software units in *XS(D, x)*.

Set 2: Active software units in *XS(D, x)*.

Set 3: Fixed software units newly created by the Snapshot operation.

Set 4: Software units in *D* but not in *XS(D, x)*.

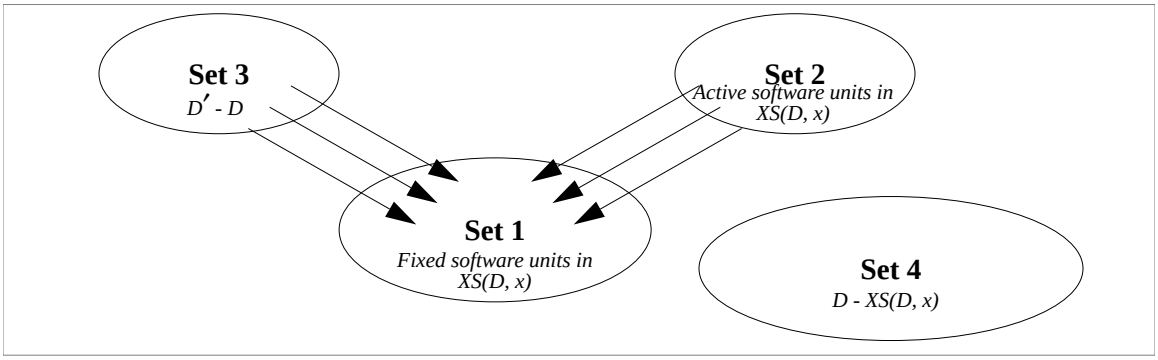


Figure 5-2: Software units in the *Snapshot* operation

According to Definition 5.11, *XS(D, x)* is the set of software units that are reachable from *x* via links. Therefore, all the software units in sets 1 and 2 are reachable from *x*, and there can be no link from set 1 or set 2 to set 4. Also, since there cannot be any link from fixed software units to active software units (Theorem 5.6), we know that there will be no links from set 1 to set 2.

In Algorithm 5.7, Snapshot creates a new fixed software unit for each software unit in set 2. The result is the software units in Set 3. Since all the links created by Snapshot are either from set 3 to set 1 (line 56) or inside set 3 (line 58), we know that set 3 cannot reach set 2 or set 4. Then, since the snapshot of *x, VPred(D′, x)*, will be in set 3, we know that *XS(D′, VPred(D′ x))* will contain the software units in sets 3 and 1 only.

The software units in sets 3 and 2 have a one-to-one correspondence. For each software unit *y* in set 2, Algorithm 5.7 creates a software unit in set 3 as *VPred(D′, y)*. Since the version predecessor of *VPred(D′, y)* is set to the original version predecessor of *y* (line 21), *VPred(D′, y)* belongs to the version family to which *y* originally belonged. Therefore, since every software unit in set 2 and set 1 belongs to a different version family, every software unit in set 3 and set 1 also belongs a different version family.

■

The Revise operation may create versions of subsystems that are not versionally consistent. Figure 5-3 shows an example.

Here, although the subsystem S(A) is versionally consistent, the new version of *S(A)* generated by *Revise(D, x, {A, B})* is not. The reason for this problem is that the Revise operation does not create new versions for software units residing in remote workareas. Although *C* is a member of the subsystem designated by *A*, no new version is created for it.

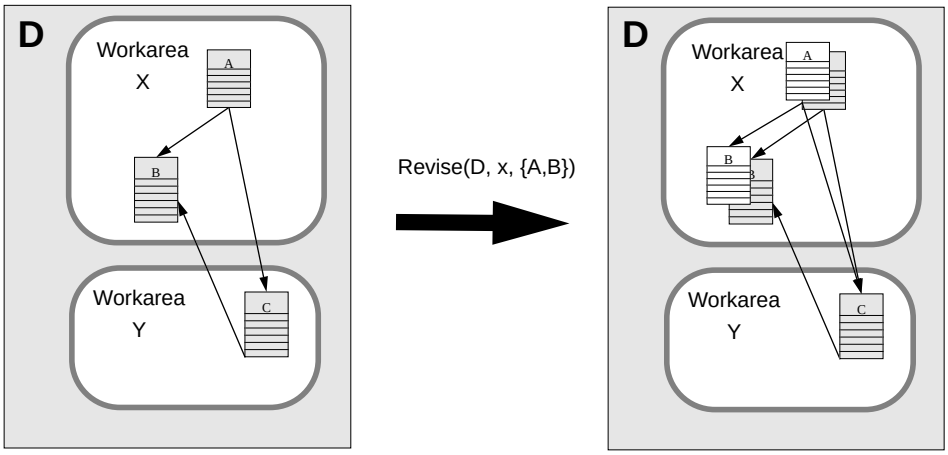


Figure 5-3: A Revise operation can cause versional inconsistency.

In fact, applying a Revise operation on a subsystem does not always cause a versional consistency problem even when the subsystem spans several workareas. We observe that the prob-

lem occurs only when there are paths of links leading out and back to the same workarea. For example, in Figure 5-3 the link from *A* to *C* leads out of workarea *X*, then the link from *C* to *B* leads back to workarea *X* again.

More precisely, the operation *Revise(D, x, W)* will not cause versional consistency problem if the intersection of *S(D, x)* and *W* is *endpoint enforced*, which is defined as following:

Definition 5.22 Endpoint Enforced

Suppose that we have a set of software units S in a software unit database D. S is ***endpoint enforced*** *if and only if the following is true:*

$$\forall D \in \Delta \;\; \forall x, y, z \in D\,[\,(x \in S) \wedge (y \in S) \wedge Uses^*(x, z) \wedge Uses^*(z, y) \Rightarrow z \in S] \tag{5.76}$$

We conclude this section by proving this property.

Theorem 5.10 Revise retains versional consistency under certain circumstances.

If an extended subsystem S(D, x) is versionally consistent and S(D, x) ∩ W is endpoint enforced, then the new revision of S(D, x) in Revise(D, x, W) is also versionally consistent.

Proof:

Let $D' = Revise(D, x, W)$, the software units in D' can be partitioned into four sets (See Figure 5-4):

Set 1: Software units in *S(D, x) ∩ W.*

Set 2: Software units in *S(D, x) - W.*

Set 3: Software units newly created by *Revise(D, x, W)*.

Set 4: Software units in *D* but not in *S(D, x)*.

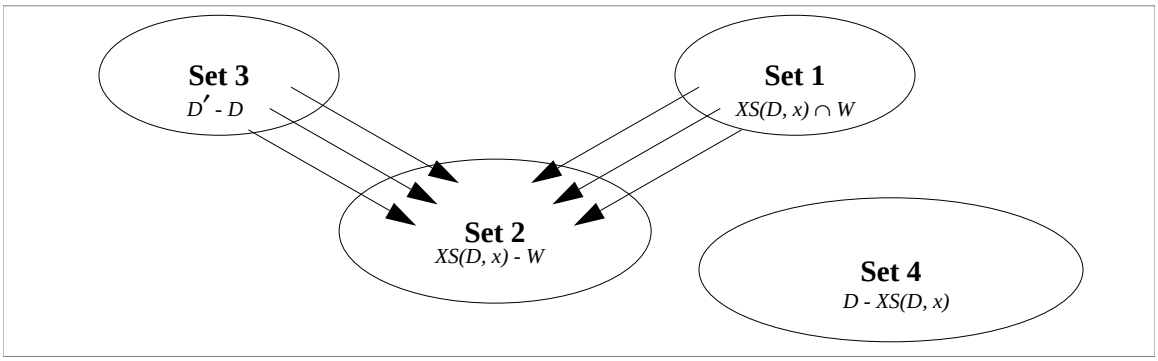


Figure 5-4: Software units in the *Revise* operation

According to Definition 5.11, *S(D, x)* is the set of software units that are reachable from *x* via links.

Therefore, all the software units in sets 1 and 2 are reachable from *x*, and there can be no link from set 1 or set 2 to set 4. Also, since $S(D, x) \cap W$ is endpoint enforced, there can be no link from set 2 to set 1.

In Algorithm 5.8, Revise creates a new active software unit for each software unit in set 1. The result is the software units in Set 3. Since all the links created by Revise are either from set 3 to set 2 (line 39) or inside set 3 (line 37), we know that set 3 cannot reach set 1 or set 4. Then, since the newly created revision of *x* will be in set 3, we know that the new revision of $S(D', x)$ will contain the software units in set 3 and set 2 only.

The software units in set 3 and the software units in set 1 have a one-to-one correspondence. For each software unit *y* in set 1, Algorithm 5.8 creates a software unit in set 3. Since the version predecessor of the new software unit is set to *y* (line 15), the new software unit will belong to the version family to which *y* belongs. Therefore, since every software unit in set 2 and set 1 belongs to a different version family, every software unit in set 3 and set 2 also belongs a different version family.

■

Chapter 6

Implementation Issues

In this chapter, we discuss some important issues we encountered in the implementation of POEM. We will discuss the overall structure of POEM, the implementation of the basic concepts, the interpretive language that let users customize POEM, and a special technique to improve the version control of objects. We do not intend, however, to cover all the aspects of POEM. Issues like the implementation of the user interface and the parsing of the script language are omitted from this thesis because they are rather straightforward.

6.1 Overall Structure

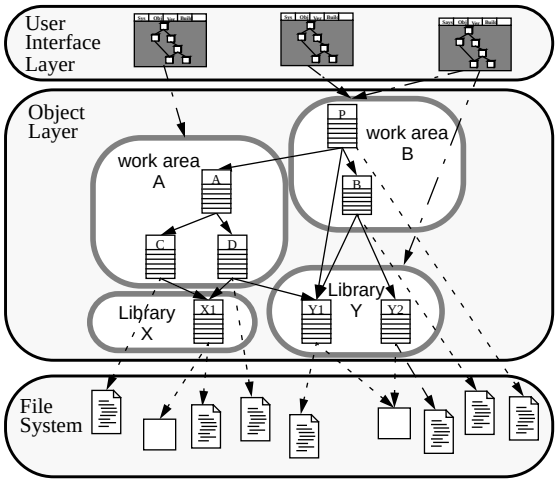


Figure 6-1: Architecture of POEM

As illustrated in Figure 6-1, the architecture of POEM consists of a *user interface layer* and an *object layer* on top of the file system. The object layer is where the software units are stored, and is basically an object-oriented database. Currently we are using an object-oriented

database system called ObjectStore [49] to implement this layer. ObjectStore is basically an extension of the C++ language that supports persistent objects, allows multiple clients to work on the same database simultaneously in a distributed environment, and supports transaction facilities to serialize the operations on objects.

To increase the flexibility of this environment, we supply an interpretive language that lets users define new classes of software units or change the definition of individual software units, thus tailoring the environment to their needs. Compatibility among user-defined software units is enforced by requiring that all new classes of software units inherit existing classes.

In POEM, users do not directly access files and directories of the underlying file system. But instead of storing everything in the object layer, POEM still stores large software artifacts like source code, object code, and documents as files in the underlying file system. Doing so makes POEM more open to the outside world. Existing tools like editors, compilers, and debuggers can be invoked by the operations of software units to process these files. We can also utilize existing code and documentation by creating software units that reference existing files.

The purpose of the user interface layer is to let programmers access software units in the object layer. This layer is fairly simple because most functions of POEM are already supported by software units in the object layer. Its major responsibilities are to display the contents of software units, to let users invoke operations of software units, and to show the relationship among software units. A graphical user interface can easily be built, since software units and their links map naturally to icons and edges. Operations of a software unit can also be represented as items in a pop-up menu.

6.2 Implementing the Basic Concepts

The four major concepts in POEM are software units, subsystems, workareas and classes of software units. In this section we discuss the implementation of these basic concepts.

6.2.1 Software Unit

We implement software units as persistent objects in ObjectStore. Software units have two kinds of attributes – *data attributes* and *operations*. Data attributes are represented as the data members of persistent objects in ObjectStore. Currently we support four atomic types and two data structures for data attributes. The four atomic types are *integers*, *strings*, *Boolean values* and *links*; the two data structures are *sets* and *lists*. The implementation of integer, strings, and Boolean val-

ues is straightforward; they are represented by the corresponding data types in ObjectStore. Links between objects are implemented by *ObjectStore references*. ObjectStore references are an extension to C++ pointers. They can be used as pointers in most cases, and remain valid between processes. The two data structures, sets and lists, are implemented on top of the corresponding data structures supplied by ObjectStore.

The operations of software units are defined in an interpretive language called VERSE, which is discussed in section 6.3. These operation definitions are translated into abstract syntax trees before being stored as attributes of software units. At run time, the execution of an operation is carried out by evaluating the corresponding abstract syntax tree.

6.2.2 Workareas

We implement each workarea as an ObjectStore database file. A workarea contains a set of software units and some display information, including the location and visibility of each member software unit. Users can change the display information interactively in the POEM workarea windows.

6.2.3 Subsystems

In the POEM model, a subsystem is a set of software units connected by links. Although subsystems play an important role in the conceptual model, we do not need extra mechanisms to implement them. The only potential problem is that a subsystem may span multiple workareas, and we may need special mechanisms to handle the links that point to software units in remote workareas. Fortunately, ObjectStore references, which are used to implement the links in POEM, handles this problem quite smoothly. They can point to remote objects in the same way as local objects. If the remote database is not open, ObjectStore opens it automatically.

6.2.4 Classes of Software Units

In POEM, we use the same kind of objects to implement classes of software units and normal software units. The inheritance relations between classes and the instantiation relations between classes and software units are both simulated by the *delegation* [53][77][84] mechanism.

The delegation mechanism is proposed as an alternative to the mechanism of classes and inheritance. The basic idea is that while each object defines its own attributes, it can also delegate

to other objects the responsibility of responding messages that do not match any locally defined attributes. As pointed out by Lieberman [53], the delegation mechanism is powerful enough to simulate the classes and inheritance that are used in most object-oriented systems. The extra power of delegation is that an instance may share data and behavior with the object it delegates to while having its own local modifications.

Delegation gives us great flexibility. It enables incremental modifications at the object level, allows dependencies between objects, and decreases the stored information by sharing. As will be discussed in section 6.4, we use a new technique based on this flexibility to improve the version control of objects. According to our experience, that new technique is rather simple and efficient.

Delegation, however, usually results in a chaotic and inefficient environment. First, because instances can redefine the attributes of the prototypes, there is no guarantee of *upward compatibility:* what is true of the prototypes may not be true of the instances. Second, the semantic complexity of the system increases greatly. Because all objects can serve as prototypes, and the prototypes have no knowledge of the existence of their instances, modification of any existing object can cause unpredictable changes in the behavior of other objects. Knowing where the value of a certain attribute is stored becomes important, but this is hard because it might be embedded in arbitrarily deep levels of delegation. Third, a delegation system will be slower than an inheritance system, because the stored values of attributes must be found at run time and there can be multiple levels of delegations between an object and the stored values.

In contrast, the class-inheritance model gives users more manageability at the expense of flexibility. It discerns the roles of classes and of objects. Classes separate structure and behavior concerns from those of run-time computation. All objects of the class are guaranteed to have the same behavior pattern. However, even with the class variables supported by some inheritance systems, the sharing of data at object level is quite limited. Without using class variables, sharing of data between two objects has to be implemented by creating a separate object that stores the common values; using class variables, all objects of the same class have to share a single value. The inheritance model does not support symmetric sharing either.

From the above observations, we decide to use a combination of inheritance and delegation in the implementation of POEM. For end users, we supply a user interface that behaves like a class-inheritance system. Users can define new classes that inherit from existing ones, and instantiate software units from classes. But internally, we use the same kind of objects to implement

classes and software units, and describe all the attribute-sharing relations between classes and software units by delegation.

Figure 6-2 illustrates the relations between classes and software units. The round-corner boxes above the dotted line are classes; those below the dotted line are software units. Every software unit and class, except the root class SoftwareUnit, has a delegation link pointing to another class. A class points its delegation link to its parent class; a software unit points its delegation link to the class to which it belongs.

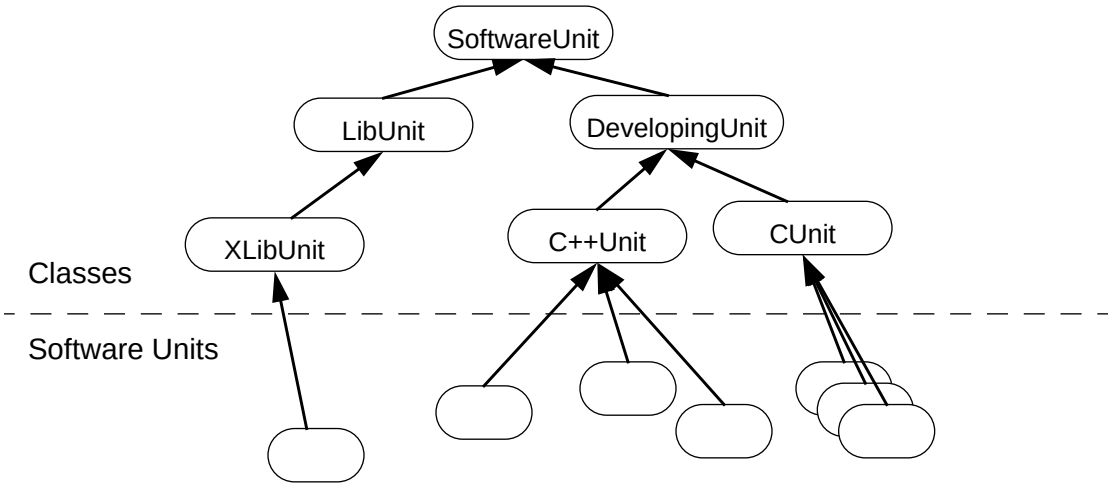


Figure 6-2: Classes and software units

When a software unit receives a message, POEM first searches for a matching attribute on the receiver software unit. If a matching attribute is found, then it is evaluated as the return value of the message. If the attribute is not found, then POEM follows the delegation link to a class and search for a matching attribute there. The search continues along the delegation links until a matching attribute is found or until the root class is reached and there are no more delegation links to follow. In the latter case, an error message is generated.

If the matching attribute is an operation found on a class, and if the operation itself uses other attributes, then the search for those attributes will start from the original receiver software unit instead of the class where the operation is defined. This is similar to the handling of the SELF variable in the SELF language [84].

An advantage of using delegation is that we can save space by sharing data attributes between classes and software units. The data attributes defined on classes serve as the default values for those on the software units. If a software unit *X* does not change the default value of a data

attribute defined in its class *C*, then there is no need to copy the value into *X*. The delegation mechanism will use the values defined in *C* as they are actually copied into *X*. In configuration management, this is especially useful because most system building processes have a lot of default parameters that are seldom changed.

Another advantage of delegation is that we can define new operations or redefine existing operations on individual software units. This may increase the flexibility of a system. However, we do not encourage users to do this because it may make the semantics of a system complicated. When special needs arise, users are advised to create a new class as the sub-class of existing ones.

6.3 VERSE

To increase the flexibility of POEM, we supply an interpretive language called VERSE to let programmers define new classes and redefine existing ones. Since most activities in POEM are carried out by operations defined in classes, changing class definitions has the effect of customizing the whole environment.

In fact, we do not hard wire any class definitions into the implementation of POEM. All the classes used in POEM, including the very basic ones, are defined in VERSE. POEM only supplies a translator to transform the class definitions into intermediate representations and an interpreter to execute the intermediate representations at run time. Although there is some overhead to using an interpretive language, the performance of POEM has been satisfactory because the time it spends on executing VERSE code is relatively short compared with the time it spends on basic tasks like compilation and archiving source documents.

The design of VERSE has to meet several requirements. First, since we are using an object-oriented model, the language must supply the basic mechanisms required in an object-oriented system, including the ability to access data attributes of an object and the ability to send messages to other objects. C++ is an example of languages with these abilities. Second, since we have to synthesize and issue many commands to the underlying operating system, the ability to concatenate strings into commands is important. UNIX shell and MAKE are examples of languages with this ability. Third, since we use sets and lists as the basic data structures in our model, the language should make the handling of sets and lists easy. Data structures like arrays and records are of secondary importance, although they are commonly used in conventional languages. Fourth, since we use the delegation mechanism, we need the ability to handle the SELF variable, which is fundamental in the delegation model. Lastly, since supporting C and C++ programming is

our major goal, it will be helpful if the syntax of VERSE is similar to that of C and C++.

We found that no existing language satisfied all the above requirements, and so we designed the syntax of VERSE. From the first and last requirement listed above, we decide to have the syntax of VERSE similar to that of C++. We also adopted features from other languages to enhance VERSE's ability to handle strings, sets, lists, and delegation.

Figure 6-3 shows an operation defined in VERSE. This operation belongs to the class C++Unit. It generates the compilation command for a C++ software unit according to the attributes of the software unit. It also takes three parameters, which are listed in the operation header at line 2.

```
1: /* C++Unit:: comiple_cmd : String */
2: operation (tfile, ofile, lib_hdr_path)
3: {
4:       [cmd, i]
5:
6:       if (@compiler == "g++")
7:            return(@gcc_compile_cmd($tfile, $ofile, $lib_hdr_path));
8:       if (@compiler == "Cfront")
9:            return(@cfront_compile_cmd($tfile, $ofile, $lib_hdr_path));
10:      if (@compiler == "SUN_CC_2.0")
11:           $cmd = "/opt/SUNWspro2.0/bin/CC -c -o " + $ofile;
12:      else
13:           $cmd = "/opt/SUNWspro/bin/CC -c -o " + $ofile;
14:      if (@build_flags.member("Debug"))
15:           $cmd = $cmd + " -g";
16:      if (@build_flags.member("Profile"))
17:           $cmd = $cmd + " -pg";
18:      if (@build_flags.member("Optimize1"))
19:           $cmd = $cmd + " -O1";
20:      if (@build_flags.member("Optimize2"))
21:           $cmd = $cmd + " -O2";
22:
23:      foreach ($i, $lib_hdr_path)
24:           $cmd = $cmd + " -I" + $i;
25:
26:      return($cmd + " "+ $tfile);
27:}
```

Figure 6-3: An operation defined by VERSE

The variables of VERSE are not typed. In Figure 6-3, all the identifiers prefixed with the character '@' reference the data attributes of the containing software unit; all the identifiers prefixed with the character '$' are parameters or local variables of the operation. The parameters are declared in the parentheses at line 2. The local variables are declared in the list at line 4.

VERSE supports five atomic data types and two structural data types. The four atomic

data types are *integers*, *Boolean values*, *strings*, and *links*. The two structural data types are *sets* and *lists*.

Using VERSE, we can concatenate strings directly with the "+" operator. For example, at line 11 of Figure 6-3 we concatenate two strings and assign the result to the variable `$cmd`. To issue this string as a command to the underlying operating system, we can use the built-in function *Sys*, as in

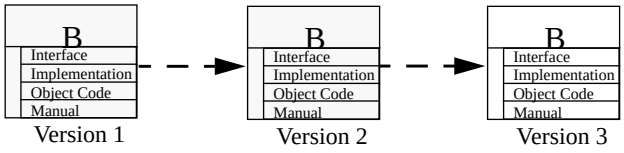
```
Sys($cmd);
```

For more information about VERSE, please see Appendix B and Appendix C. Appendix B explains the syntax of VERSE, Appendix C gives an example of defining a class with VERSE. A more detailed explanation of VERSE is also available as [55].

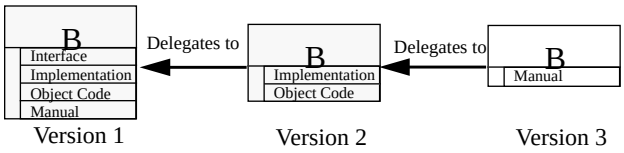
6.4 Versioning by Delegation

In POEM, we use a new technique based on delegation to handle the versions of software units. Similar to other existing version control systems, we save space by storing only the differences between versions. But additionally, our technique allows old versions to be accessed directly without check-in and check-out operations. This technique is rather simple and can be generalized to be used in any object-oriented system.

This technique is illustrated in Figure 6-4. Suppose we have a software unit with four data attributes – interface, implementation, object code, and manual. In the second version of this software unit, we modified the implementation and rebuilt its object code. Then in the third version, we modified its manual.



(a) Logical View



(b) Physical Representation

Figure 6-4: Versioning by Delegation

Logically, this results in three versions of the software unit, each containing all four data attributes, as shown in Figure 6-4(a). Users can access any attribute of the three versions. But internally, we store only the modified attributes in each versions, and have each version point a delegation link to its version predecessor. This is shown in Figure 6-4(b).

Suppose we want to access the interface of the second version. Since the second version does not have this data attribute, POEM follows the delegation link to version 1 and retrieves the interface there. The retrieved interface is returned as it is locally defined at version 2. Similarly, if we try to access the interface of version 3, POEM follows the delegation links and retrieves the interface from version 1.

Here we take advantage of the fact that all old versions in our framework are fixed (Theorem 5.6). If this property does not hold and modification of old versions is allowed, then any change made on an old version may also change the behavior of its version successors. This is seldom what we want and handling this problem requires extra mechanisms.

6.4.1 Revise and Snapshot

The two basic version control operations in POEM are revise and snapshot. When we apply the revise operation on a software unit, POEM creates a new version of the software unit and has it point a delegation link to the original version. The new version is created without any attributes. Logically, this makes the new version exactly the same as the original version. When we read an attribute, its value is retrieved from the original version. But when we try to write to an attribute, the new value is stored in the new version. This is similar to the copy-on-write technique

used in operating systems.

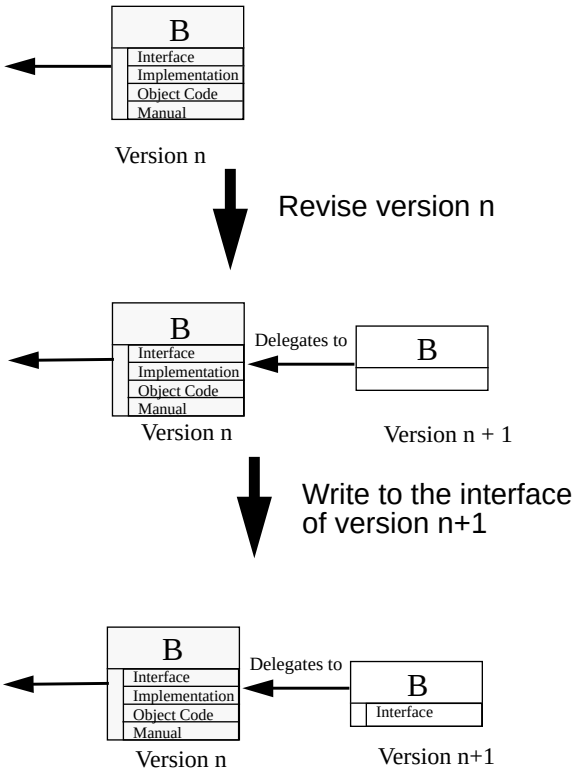


Figure 6-5: Versioning by delegation – Revise

The handling of the snapshot operation is more complicated. As illustrated in Figure 6-6, when we apply the snapshot operation on an active software unit, a new fixed version is created as the version predecessor of the original version. All the attributes of the current version are also moved to the newly created fixed version. Comparing Figure 6-6 with Figure 6-5, we notice that the snapshot operation is very similar to the revise operation. However, there are two major differences. First, we start with a active version instead of a fixed version in the snapshot operation. Second, we associate the identity of the original version with the version successor instead of the predecessor. This is important because if there are some links pointing to the active version before the snapshot operation, those links will still point to an active version after the snapshot operation.

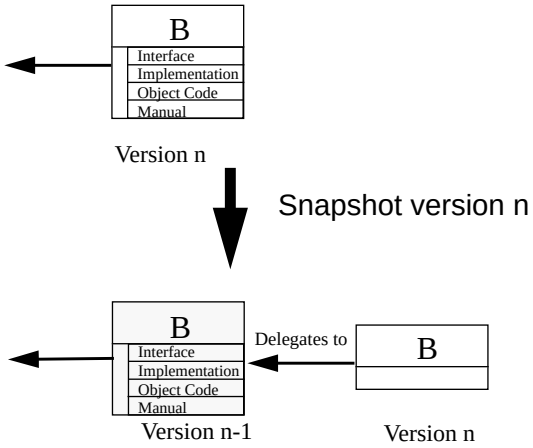


Figure 6-6: Versioning by delegation – Snapshot

6.4.2 Interaction with Inheritance

The major challenge we found in implementing our version control mechanisms by delegation is their interaction with the inheritance mechanisms. As discussed in Section 6.2.4, we also use the delegation mechanism to implement the inheritance relations between classes and software units. Therefore, except for the first versions of each software unit, every software unit has two delegation links, one to its class and the other to its version predecessor. This is illustrated in Figure 6-7. To make things simple, we currently assume that all versions of a software unit belong to the same class.

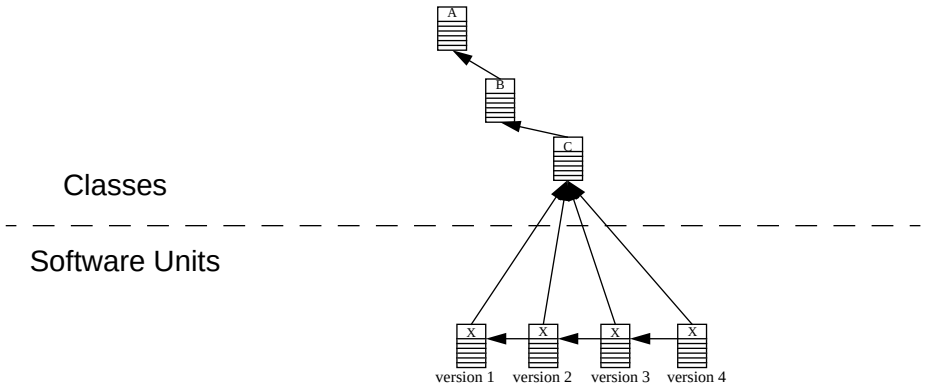


Figure 6-7: Inheritance and versioning by delegation

Since each software unit has two delegation links, we have to decide which one to follow

first in the search of attributes. If we follow the inheritance link first, then we may miss attribute values defined in earlier versions. Let us take Figure 6-7 as an example. Suppose class *C* defines the default value of an attribute α, and this default value is overwritten at version 3 of *X*. Since version 4 is supposed to start with the same contents as version 3, the system should get the value of α from version 3 when we try to access α in version 4. However, if the system follows the inheritance link first, *C* is visited before version 3 of *X* and the default value of α defined at *C* will be returned.

On the other hand, if we follow the version predecessor link first, then the semantics will be correct but it will take longer to find an attribute defined in the classes. Every time we access an attribute defined in the classes, the system must search all the earlier versions before looking into the classes. In the example of Figure 6-7, if we want to use an operation defined in class *C* in version 4 of *X*, then all the earlier versions of *X* will be visited before *C*.

Weighing the pros and cons of these two approaches, we decide to opt for the second one. It is obvious that being slow is better than being incorrect. And fortunately, the time required to traverse delegation links is relatively short compared with other configuration management activities like compilation and archiving files. In our experience, searching for attributes has not been a major factor in the performance of POEM.

Another benefit of following the version predecessor link first is that we can omit the inheritance links of later versions, as illustrated in Figure 6-8. Since the inheritance links of latter versions lead to the same classes, there is no need to keep them around.

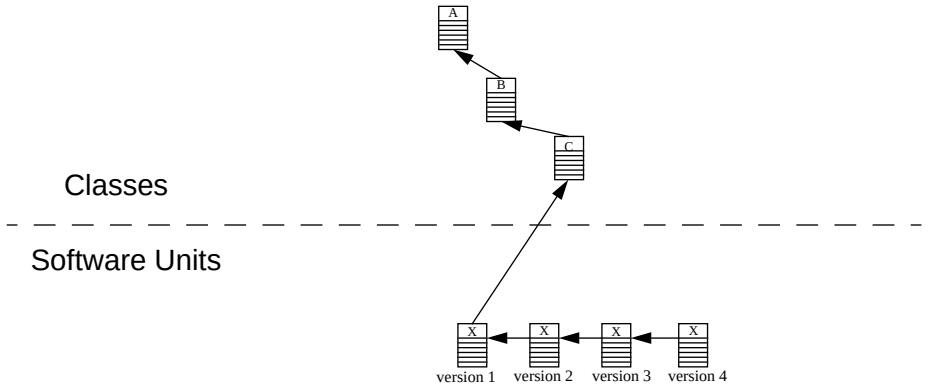


Figure 6-8: Using a single delegation link for versioning and inheritance

In case the traversing of delegation links does become a performance concern, we can

speed up the searching by short-circuiting the delegation links. When there are many versions of a software unit, we can copy all the attribute values of earlier versions into a new version, and make the delegation link of the new version pointing directly to its class. This is illustrated in Figure 6-9. In this example, we copy the attribute values defined in version 1 through version 6 into version 7 and short-circuit the delegation link of version 7 to the class *C*. This operation does not modify the semantics of any version, but speeds up the searching of attributes in version 7 and its version successors.

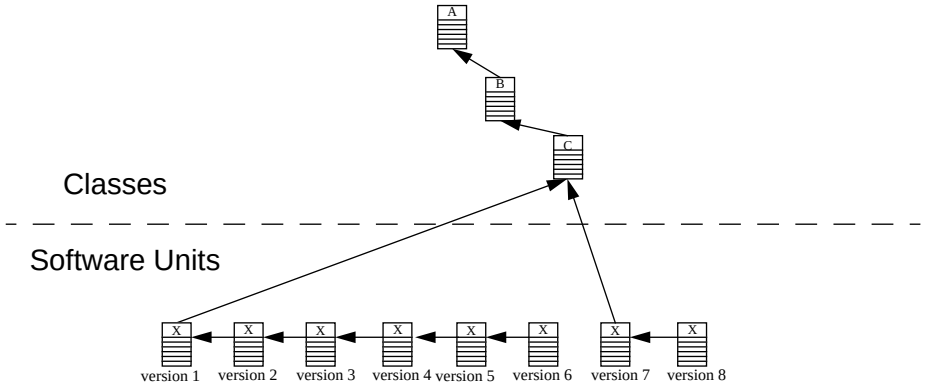


Figure 6-9: Short-circuiting a delegation link to improve performance.

6.4.3 Discussion

The fact that version control mechanisms can be implemented by delegation is not just a coincidence. We find many similarities between version control and delegation. The concepts of version predecessors and version successors in version control can be mapped to that of prototypes and instances in the delegation model. While a version successor is derived from its predecessor, an instance is derived from its prototype. While a new version is created to refine its predecessor, an instance is created to extend its prototype. While new versions are created with the same contents as their version predecessors, instances in delegation are created with the same contents as their prototypes.

The two major benefits of versioning by delegation are that all versions of an object can be accessed directly without check-in and check-out operations, and multiple versions of the same object can exist simultaneously in the same workarea. In contrast, most existing version control systems require check-out operations before using an old version and check-in operations before

we can make an object into history. Most existing version control systems also permit only one version of an object to exist in each workarea. If we want to compare versions of the same object, we have to rename the versions or use special tools to access the version repositories directly.

Chapter 7

Conclusion

The major goal of this research is to find a framework that makes configuration management simple yet powerful. We have shown that it is possible to provide a programming environment that handles system building and version control according to the logical structure of source code. This makes programming easier because programmers no longer have to maintain the mapping between the structure of source code and that of the physical software artifacts. Our framework also handles derived objects automatically and allows programmers to handle large subsystems as a single unit.

Our framework draws its power from two principles. First, we organize software artifacts at the *configuration management leve*l in parallel with the modularization at the *source code level*. A software unit plays two roles at the same time: it represents a function or a class in source code and it corresponds to a configuration in configuration management. Since we use the same decomposition structure, the gap between handling source code and managing configurations is narrowed. Additionally, since we represent the relations between modules explicitly as links, programmers can examine and traverse them directly without looking into the source code. This helps programmers to understand the general structure of a program.

Second, our approach applies the principles of modularization and encapsulation to configuration management. It is well known that modularization and encapsulation are very useful in managing source programs. Similarly, they can help greatly in handling configuration management. As pointed out by Osterweil [66], software processes can be considered special programs that are enacted by both human and computers. The data used by these special programs are software artifacts like source code, derived objects, system models, documentation, version history files, etc. By applying the principles of modularization and encapsulation on these software arti-

facts, we can simplify configuration management greatly. Modularization helps us to decompose the configuration management tasks into manageable units. Encapsulation helps us to hide the different configuration management requirements of individual subsystems.

7.1 Summary of Contributions

The major contribution of this thesis is a new framework for designing configuration management systems. We showed that programming environments designed with this framework will be easier to use. We showed that we can formalize our framework and derive various useful properties from the formalization. We also showed that our framework is practical enough to be implemented and used for real projects.

Easier to Use

In Chapter 3, we showed that POEM, a prototype programming environment based on our framework, supports mechanisms that let its users handle configuration management directly in terms of functions and classes. We argued that these mechanisms make configuration management easier. We compared POEM with other existing environments and discussed why the difference is important in version control, system building, cooperative programming, software reuse, and software maintenance.

A Unified Model

In Chapter 4, we described the basic mechanisms of our framework and discussed their roles in system building and version control. In Chapter 5, we formalized our framework and proved several properties that we claimed in Chapter 4. With the properties we derived in this chapter, we know what can happen in our environments and what cannot.

Practical

We demonstrated that our framework is practical by developing a prototype environment called POEM and then using POEM to develop several programs. The use of POEM was discussed in Chapter 3 and the implementation issues were discussed in Chapter 6. We showed that although POEM supports a rich set of functionalities, it can be implemented rather easily. We also showed that our framework is quite flexible and open. The current implementation of POEM uses many existing tools. In addition, users can customize their environments or incorporate new tools by designing their own classes of software units.

7.2 Limitations

In this section we describes the limitation and drawbacks of our framework and discuss some possible solutions to these problems. The problems discussed here are not tied to a certain implementation, but are intrinsic to our framework.

Most of the problems discussed here can be worked around by augmenting our framework with additional mechanisms. But solving each problem by introducing a new mechanism will make the framework more complicated, and thus harder to understand and implement. Besides, new mechanisms may cause new problems and may interfere with each other.

7.2.1 Managing Legacy Code

Our framework is aimed at supporting the development of new programs. It cannot directly manage software in traditional environments. The major reason for this difficulty comes from the difference in the units of operations for configuration management. In traditional programming environments, files and directories are the two major tools for organizing software; in our framework, software is organized by software units and workareas. Although a directory may be mapped to a workarea, a file cannot be mapped to a software unit directly. In addition, although our framework uses links to replace the "#include" directives in traditional environments, an "#include" directive cannot be cannot be used in place of a link. A link is version-sensitive. It points to a certain version of a software unit. In contrast, a "#include" directive can be used to include any version of a file.

A possible solution to this problem is to develop a tool that automatically transforms existing code into our framework. Indeed, we discussed in Section 4.6.6 a algorithm that can perform this kind of transformation. Programs transformed by that algorithm, however, cannot take full advantage of our framework. On the one hand, functions and classes are not separated into individual software units. One the other hand, interface and implementation of the same module are not combined into the same software unit. If we want to have a better transformation tool, we will need a more complicated algorithm that analyzes the syntax and semantics of the existing code.

7.2.2 Handling General Documentations

Since our framework is based on the modularization of programs, it cannot properly handle documents generated before a project is decomposed into functions and classes. Our frame-

work is designed to support configuration management activities in software design, implementation, and maintenance, but not activities like requirement specifications and feasibility studies.

Our framework cannot properly manage users' manuals and tutorials either. Those documentations are written from an end user's point of view, not the software developer's. The way in which they are organized is usually very different from the way programs are decomposed into modules.

A possible solution to this problem is to manage that documentation separately and then associate them with software units in our framework by hyper-links. We could define new classes of objects to store parts of documentations, and then add new types of links to our software units so they can point to objects in the corresponding documentation.

Using a hyper-link approach, however, may result in unmanageably complex systems. As discussed in Section 2.3, a system with too many unrestricted hyper-links is like a program with many goto statements. They may cause cognitive overhead and navigation problems.

7.2.3 Dependence on Graphical User Interfaces

Although we can define our framework independent of any user interface, as shown in Chapter 5, a proper graphical user interface is necessary if we want to make the implementation of our framework practical and easier to use. Without a graphical user interface, it is not easy to specify and manage links between software units, or to understand the relations between software units.

Graphical displays have become a common feature of modern workstations and personal computers. But they are still more expensive than textual ones. Some programmers also resent the use of graphical user interfaces because they are usually slower than textual ones, and because programmers have to move their hands between keyboards and mice. This may distract programmers and slow down their work.

7.2.4 Managing Toolkits

Sometimes a set of functions and classes are logically related but do not use one another's interfaces. This kind of situation happens frequently in the so-called tool kits. For example, the Xt library of the X window system contains many functions that can be called by application pro-

grams, but those functions seldom depend on each other. We illustrate the relations between software units of a toolkit in Figure 7-1.

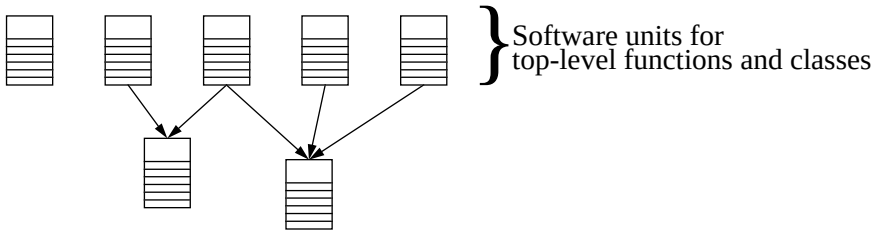


Figure 7-1: A toolkit

Handling toolkits may be a problem in our framework because there is no single class or function that is the common parent of all the members of a toolkits. Without a common parent, we cannot create and manage versions of the toolkit as a single unit, or build object code of the toolkit with a single build operation.

A possible solution to this problem is to create a dummy software unit to act as the common parent of members of toolkits, as illustrated in Figure 7-2. To build object code for the whole toolkit, we invoke the build operation of the dummy software unit; if we want to keep the current stage of the toolkit, we invoke the snapshot operation of the dummy software unit.

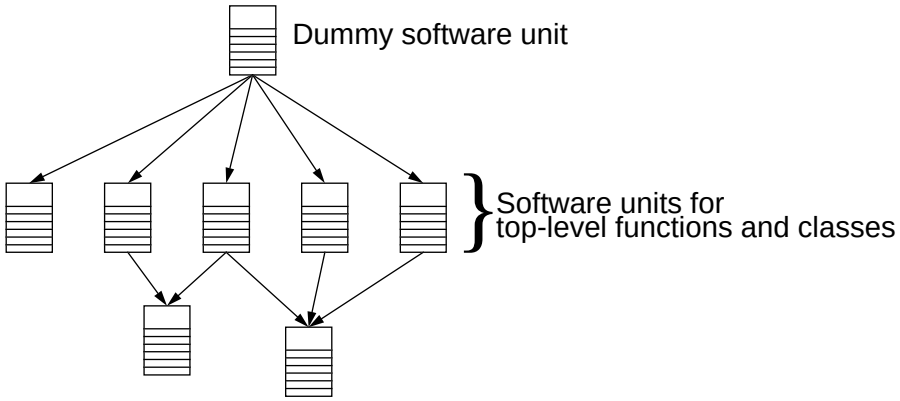


Figure 7-2: Using a dummy software unit to manage a toolkit

Another possible solution to this problem is to define operations on workareas. For example, we can store all members of a toolkit in a workarea and build the object of the whole toolkit by using a *workarea-build* operation that send build messages to all software units in the workarea. This approach, however, requires some modification of our framework.

7.2.5 Changing Versions of Low-level Components

Sometimes a software unit is used by many other software units. If we want to replace the shared software unit with a new version, then we may have to modify many links. Figure 7-3 illustrates this kind of situation. Changing many links using a graphical user interface might be very cumbersome.

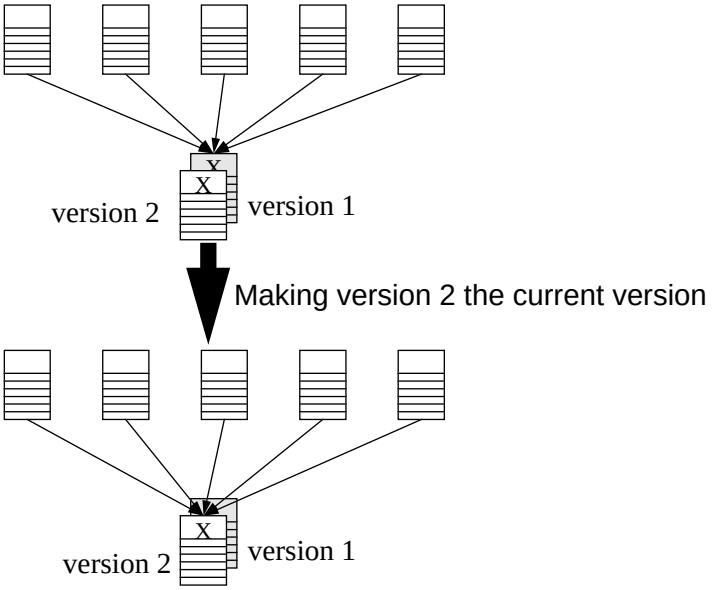


Figure 7-3: Changing versions of a shared software unit

This problem happens in our framework because the destinations of our links are fixed and we do not support the notion of "current version," unlike most other version control systems. In traditional C/C++ programming environments, a "#include" directive or a make rule does not specify the version of the files it refers to. If we want to use a different version of a file, we can simply bring that version into the directory that keeps all the current versions.

A simple solution to this problem is to supply a command in the user interface that automatically replaces links for users. For example, we can supply a command that takes two software units *x* and *y* as its arguments and then replaces all links to *x* by links to *y*. If such a mechanism is available, we can perform the task illustrated in Figure 7-3 with a single operation.

The drawback of this approach is that if the users of the shared software unit are scattered in several workareas, then the link replacing mechanism must make modifications in all those workareas. This may conflict with other programmers' activities. Moreover, finding all the workareas that contain users of a shared software unit is not trivial in a large system.

A second possible solution to this problem is to create a *broker* for the shared software unit. As illustrated in Figure 7-4, we can create an empty software unit to serve as the broker of the shared software unit. All the software units that want to use the shared software unit point their links to the broker, and the broker points a t_uses_interface link to the shared software unit. With the t_uses_interface link, the broker has the same effective interface as the shared software unit. To replace the shared software unit with a different version, we simply change the destination of the t_uses_interface link.

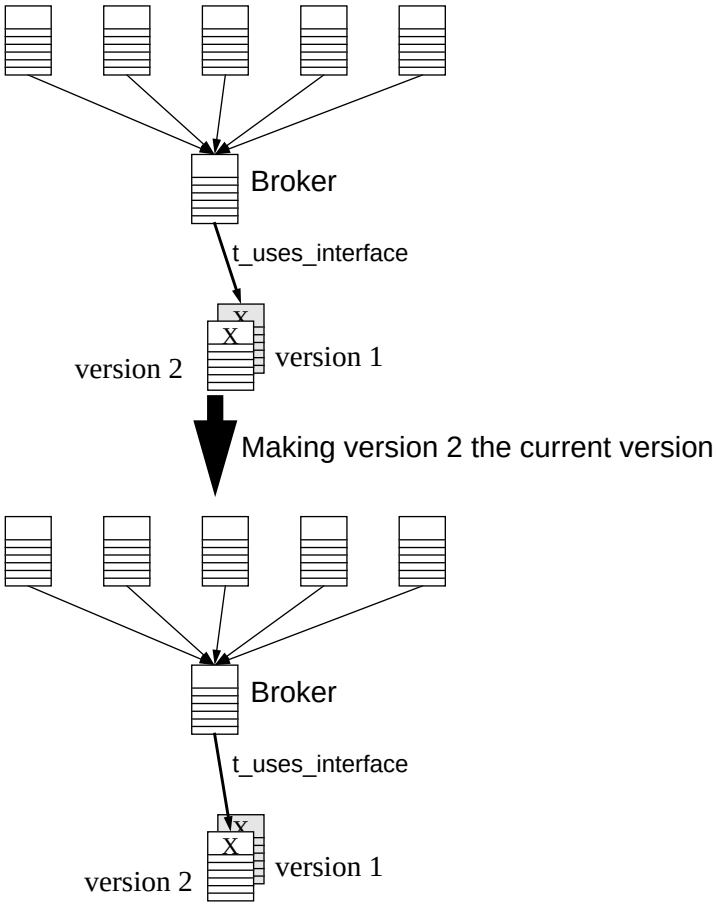


Figure 7-4: Changing version of a shared software unit with a broker

The drawback of this approach is that we have to create a broker for each shared software unit. This is not natural and may make the structure of a program more complicated.

A third solution to this problem is to use special *version-selecting links*. For example, we can have a special link that always points to the latest fixed version of a certain software unit. This kind of link is different from the conditional links discussed in section 4.6.5. While a conditional

link always points to the same software unit, the destination of a version selecting link is calculated dynamically. The behavior of version selecting links is similar to that of the version selection rules in DSEE and SHAPE.

There are two major drawbacks of this approach. First, implementing version selecting links might be expensive. Evaluating the destinations of links at run time will degrade the performance of our system. Second, setting rules associated with the version-selecting links might be difficult, especially when there are multiple version branches of the shared software unit. If we do not set the rules properly, then the destination of a version selecting link may jump from one version branch to another version branch. This is usually not desired.

7.3 Future Work

While the basis of our framework is established in this dissertation, there are still many problems to be solved. Below we discuss some possible research directions of this research.

7.3.1 Field Test

We need more field testing of our ideas. Until now, our experience with POEM has been more than satisfactory. But, limited by the resources in a university, we have not used POEM on real-life projects. The largest program we have tried to develop under POEM is POEM itself, which contains only several thousand lines of source code. We are especially interested in seeing how the working patterns using workareas in POEM differ from those using workspaces in existing programming environments.

7.3.2 Merging

As discussed in Chapter 4, our framework uses workareas instead of workspaces. Components of a program are partitioned into workareas instead of duplicated in each workspace. Consequently, our framework does not depend on the merging mechanism as much as some commercial systems, like TeamWork and ClearCase, do. The merging of version branches, however, is still an important problem in our approach.

Most existing version control systems, like RCS and SCCS, support some line-based merging tools. Those tools are rather simple and rely on users to solve most of the conflictions they find. There are also some efforts to develop syntax-based or semantics-based merging tools [42][86][89][90][91]. Potentially, those tools can generate more accurate results and reduce the

number of false alarms about conflicts. These techniques, however, rely on very complicated algorithms and are not yet mature enough to apply to real-life programs.

Merging tools, whether line-based, syntax-based, or semantics-based, can be incorporated into our framework to merge individual software units. We can define an merge operation that merges two fixed software units into a new active software unit, or merge a fixed software unit into an existing active software unit. Although a merged version may have more than one version predecessors, no major modification of our framework should be required because the concept of "version predecessor" is not vital to our model.

A more interesting but also more difficult problem is the merging of subsystems. According to the philosophy of our approach, operations should be able to operate on a function or a class without considering the functions and classes it uses. In the case of merging, we should be able to merge two versions of a software unit without considering the software units they use. This is equivalent to merging two subsystems. Since each subsystem in our framework is a graph of software units and links, and each version of a subsystem may contain different software units, the merging of two subsystems is similar to merging two rooted graphs. To our knowledge, there is no significant research on this problem.

7.3.3 Supporting Debugging and Run-time Analysis

Our prototype environment, POEM, supports some functionalities, like source code editing and graphical layout of software structures, that are not traditionally supported by configuration management systems. But POEM still does not constitute a full-fledged programming environment. In particular, it does not have enough support for debugging and run-time analysis.

A possible solution this problem is to integrate POEM with FIELD [59][69][70]. FIELD is another programming environment developed at Brown University. It has a rich set of textual and graphical tools for run-time analysis, and supports several visualization tools that enable users to analyze the same program with different views. All these functionalities are lacking in POEM. POEM, on the other hand, can complement FIELD with its configuration management abilities. Although FIELD also supports some configuration management tools, they are based on RCS and MAKE and inherit from them some major limitations. For example, it is difficult to handle related makefiles at the same time with the system building tool of FIELD.

Integrating POEM with FIELD is expected to be easier than integrating POEM with other programming environments. The philosophy of FIELD is very similar to that of POEM in that

both environments encourage programmers to think and work in terms of the logical structures of software. The message-broadcasting service of FIELD can also make integration easier. We should be able to integrate the two environments without modifying FIELD's source code.

7.3.4 Supporting Geographically Distributed Development

As software systems become larger and network communication becomes cheaper and faster, more and more projects are developed by programmers located at different development sites. For a software configuration management system to support large-scale development efforts, it must address the problem of geographically-distributed development.

Supporting geographically distributed configuration management is difficult because it is usually impractical to use the same copy of software over a wide-area network. Some components of a project have to be duplicated. A configuration management system has to decide what to duplicate and how the modifications at different sites are to be synchronized.

A possible way to support geographically distributed configuration management using our framework is to duplicate all the fixed software units at every site and keep active software units local to their owner sites. We would not duplicate the active software units because they represent the ongoing work of each programmer, and programmers usually would not be interested in the intermediate results from programmers at remote sites. Duplicating fixed software units would not be a problem because they are not modified after being created.

7.3.5 Change Control

Change control is concerned with managing the *process* of developing and maintaining software artifacts, rather than managing the software artifacts being developed and maintained. It controls the request, evaluation, approval, and scheduling of changes, and is a high-level activity based on configuration management mechanisms.

Since the configuration management mechanisms supported in our framework are rather different from those in existing systems, we expect that change control based on our environments will be rather different too. We are especially interested in seeing how change control based on workareas in POEM will differ from that based on workspaces in existing systems. In systems that use workspaces, programmers simultaneously work on different parts of the same version of a program. Then every so often everyone would temporarily stop their work for a complete build of the

whole program. In contrast, using workareas in POEM would be similar to working in a assembly line. Programmers working on low-level modules will continuously generate new versions of their functions and classes and then pass these new versions to programmers working on high-level modules.

7.3.6 Transforming Existing Code

As discussed in section 7.2.1, our framework cannot directly manage code generated by traditional environments. And although we can transform existing code into our environments with a simple algorithm as discussed in section 4.6.6, the result of this direct transformation is seldom satisfactory.

A possible solution to this problem is to develop a tool that can extract the *source model* of an existing program. The source model of a program includes information like call graphs, class diagrams, and file dependencies. Using this information, we can reorganize an existing software system into software units that represent individual functions and classes.

There are already some tools that can extract the source model rather efficiently. For example, FIELD extract this information by using language-specific parsers. Rigi [62] uses spatial and visual information inherent in graphical representation of software systems to extract system abstractions and design information. Murphy and Notkin described in [63] a lightweight source model extractor based on lexical analysis.

Appendix A

Makefiles for the Example in Chapter 3

In chapter 3 we showed an example of developing POEM under POEM itself. In this appendix, we list the makefiles required if we were to develop that example using MAKE instead. The makefiles shown here actually come from the current implementation of POEM. But since the example in chapter 3 implemented only parts of POEM, we also simplify the makefiles here, so that they correspond faithfully to that example.

A.1 Directory Structure

As illustrated in Figure A-1, the files of POEM are divided into several directories. Each directory under the root directory <poem> represents a major module of POEM. The directory <poem/poem> represents the main module of POEM. The directory <poem/poemui> represents the user interface module of POEM. <poem/data> keeps the information that is common to all modules. There are also directories for modules that are not discussed here.

We keep source files and object files of the same module in different directories. For example, we keep the source files of the user interface module under <poem/poemui/src>. The corresponding object files are kept under <poem/poemui/bin>. We separate them because we may want to build different object files from the same copy of source files. For example, we can create a directory <poem/poemui/bin.HP> to store the object files generated for HP workstations.

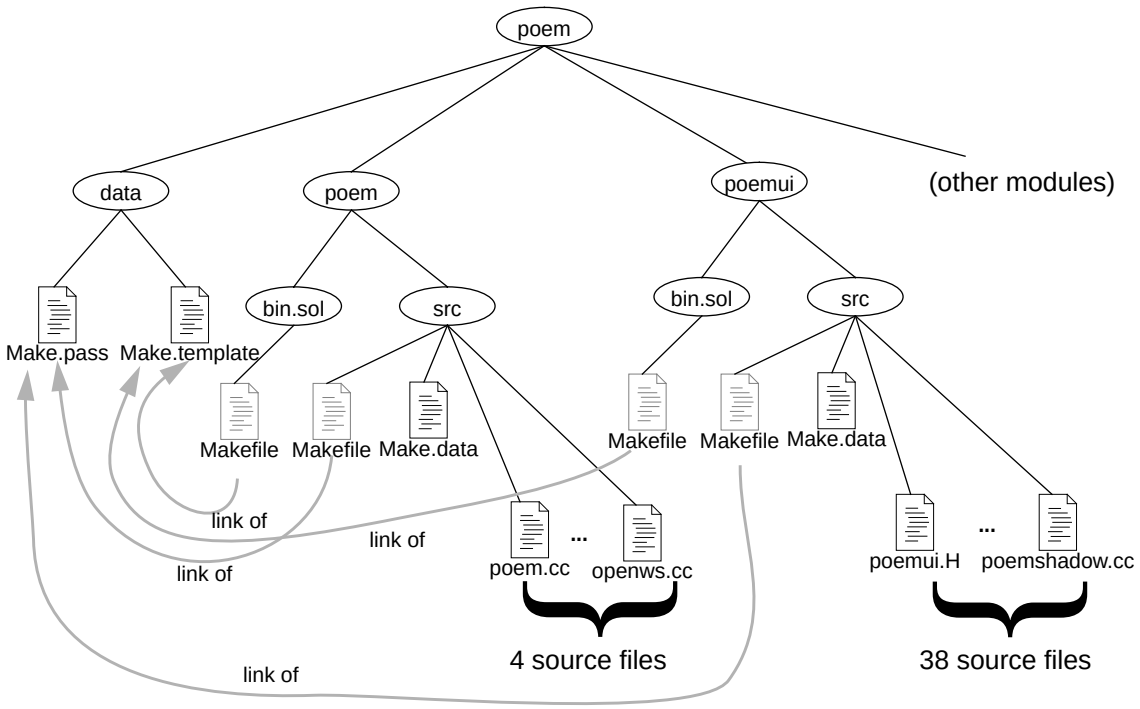


Figure A-1: Directory structure of POEM

Since the makefiles used for different modules are rather similar, we extract their common parts into two makefiles, “Make.pass” and “Make.template,” and store them in the directory under <poem/data>. We make a symbolic link to “Make.pass” from the source directory of each module, and a symbolic link to “Make.template” from the binary directories of each module. The major function of “Make.pass” is to pass its control over to the corresponding “Make.template” under the binary directory. It allows us to start the making process directly without changing directory to the corresponding binary directory. The actual building is carried out by “Make.template.” It includes “Make.data” under the corresponding source directory. “Make.data” contains information that is unique to each module.

A.2 Make.template under poem/data

```
# The following should be defined in the Make.data file:
#    PACKAGE -- package name
#    SFILES  -- source files
#    OFILES  -- object files
# The following are optional in the Make.data file:
#    USERFLAGS -- additional flags used in compilation
#    LPATH   -- library search paths
#    LFLAGS -- libraries needed
```

```
CCFLAGS = -g -I/pro/poem/include -I../src $(USERFLAGS)
LDFLAGS = -g -L/pro/poem/lib $(LPATH) $(LFLAGS)

VPATH= ../src
SHELL= /bin/sh

CC = /cs/bin/CC

LIB = /pro/poem/lib/lib$(PACKAGE).a

ifndef SFILES
SFILES= $(wildcard ../src/*.[Cc]?)
endif

%.cc: %.ll
	lex $<
	mv lex.yy.c $@

%.cc: %.yy
	yacc $<
	mv y.tab.c $@

%.o: %.C
	$(CC) $(CCFLAGS) -c $<

include ../src/Make.data

############################
# Rules
############################

lib: $(LIB)

$(LIB) lib: $(OFILES)
	ar rvu $(LIB) $(OFILES)
	ranlib $(LIB)

clean:
	rm -f *.o

############################
# makedepend
############################

depend:
	(cd ../src; makedepend -f .depend -I/opt/SUNWspro/SC3.0/include/CC $(CCFLAGS) $(SFILES))

include ../src/.depend
```

A.3 Make.pass under poem/data

```
#
#           Make.pass
#
#     This file is the default makefile for source directorys.  It causes
#     the make to actually be done in the proper binary directory.  Each
#     source directory should link the name `Makefile` to this file.
#

#ifndef ARCH
ARCH= `arch`
#endif

.SUFFIXES:

all .DEFAULT:
	(cd ../bin.$(ARCH); gmake $(PARM) $@)

create:
	- if [ ! -d ../bin.$(ARCH) ] ; then mkdir ../bin.$(ARCH); else true; fi
	- if [ ! -d ../lib ] ; then mkdir  ../lib; else true; fi
	rm -f Makefile
	ln -s ../../data/Make.pass Makefile
	rm -f ../bin.$(ARCH)/Makefile
	ln -s ../../data/Make.template ../bin.$(ARCH)/Makefile
	touch .depend
```

A.4 Make.data under poem/poem/src

```
# POEM -- Top-level functions of POEM
#

PACKAGE= poem

USERFLAGS =  -I/cs/include/motif -I/pro/poem/include

LLPATH = -L/cs/lib/motif -L/user/lib/X11 \
        -L/pro/poem/poemui/bin  -L/usr/openwin/lib -L/opt/SUNWspro3.0/lib

LDLIBS  = $(LLPATH) -lXm -lXt -lX11 -lgen -lm

OFILES= poemmain.o openws.o
SFILES= poemmain.cc openws.cc
all: poem

poem: $(OFILES) /pro/poem/poemui/bin/libpoemui.a
	$(LINK.cc) -o poem $(OFILES) $(LDLIBS)
```

A.5 Make.data under poem/poemui/src

```
#
# POEM -- The user interface of POEM
#
```

```
PACKAGE= poemui

USERFLAGS =  -I/cs/include/motif -I/pro/poem/include

LLPATH = -L/cs/lib/motif -L/user/lib/X11 \
        -L/pro/poem/lib  -L/usr/openwin/lib -L/opt/SUNWspro3.0/lib

LDLIBS  = $(LLPATH) -lXm -lXt -lX11 -lgen -lm

OFILES= poemui.o poemmenu.o wscb.o sucb.o \
        selectcb.o poembtn.o poemexpose.o classcb.o poemerr.o sutbl.o \
        invokeverse.o getsu.o edge.o node.o vccb.o poemvc.o \
        focus.o buildcb.o shadowcb.o poemshadow.o

SFILES= ooemui.cc poemmenu.cc wscb.cc sucb.cc \
        selectcb.cc poembtn.cc poemexpose.cc classcb.cc poemerr.cc \
        sutbl.cc invokeverse.cc getsu.cc edge.cc node.cc vccb.cc poemvc.cc \
        focus.cc buildcb.cc shadowcb.cc poemshadow.cc

all: lib
```

Appendix B

VERSE

In this appendix we explain the syntax of VERSE, the interpretive language used to define classes and operations in POEM. The syntax of VERSE is based on C, but also borrows some features from SELF [1][84] and the DML of ObjectStore [49]. There are also some features that are unique to VERSE.

B.1 Data Types

VERSE supports four atomic data types and two structured data types. The four atomic types are integers, boolean values, strings, and links. The two structured types are lists and sets. Integer values can be used in places where boolean values are expected. All non-zero integer values are treated as TRUE; zero is treated as FALSE.

B.1.1 Atomic Data Types

- Integers

  Examples:

  5
  -3

- Boolean Values

  Examples:

  TRUE
  FALSE

- Strings

  Examples:

  “<string>”
  ““ // empty string
  “This is a string”
  “This symbol: \” is a double quote.”
  “This symbol: \n is a new line character.”

- Links

Format 1:

:<identifier>

Example:

```
:foo                    // a software unit in the current workspace;
```

Format 2:

<dir_name>:<identifier>

Example:

```
/u/yjl/poem:root_obj    // a software unit in the workspace /u/yjl/poem;
```

Format 3: (Not implemented)

:<identifier>(<boolean_expression>)

Examples:

```
:foo(@version == 3.2)
:foo(@fixed == TRUE && @owner == "yjl")
```

Format 4:

<dir_name>:<identifier>(<boolean_expression)

Examples:

```
"/u/yjl/poem":dtor(@version == 1.2)
```

Format 5:

NULL

Example:

```
NULL                    // null value.
```

Format 6:

SELF

Example:

```
SELF                    // self.
```

B.1.2 Structured Data Types

- Lists

Format:

[<value 1>, <value 2>, ..., <value n]

Example:

```
[7, 1, 1, 2, 6, 7, 8]
[ ]                     // Empty list.
```

- Sets

Format:

[: <value 1>, <value 2>, ..., <value n> :]

Example:

```
[:7, 1, 2, 6, 7, 8:]
[::]                    // Empty set.
```

B.1.3 Undefined Attribute

If an attribute is undefined, the value "UNDEF" is returned.

Example:

```
if (@x == UNDEF)
    ...;
```

B.2 Storage Classes

- Local Variable

  Format:

  $<identifier>

  Example:

  ```
  $foo                    // A local variable called “foo.”
  ```

- Attribute

  Format 1:

  self.<identifier>

  Example:

  ```
  $su.foo                 // attribute “foo” of variable $su.
  SELF.foo                // attribute “foo” of SELF.
  ```

  Format 2:

  @<identifier>

  Example:

  ```
  @foo                    // same as SELF.foo.
  ```

B.3 Operators and Expressions

B.3.4 Boolean Operators

- Negation

  Format:

  !<expression>

  Example:

  ```
  !@debug                 // The negation of the attribute “debug.”
  ```

- Or

  Format:

  <expression> || *<expression>*

  Example:

  ```
  @debug || @profile
  ```

- And

  Format:

  <expression> && <expression>

  Example:

  ```
  @debug && $trace
  ```

B.4 Integer Operators

- Arithmetic Operators

  Examples:

  ```
  3 + 4                   // Addition
  3 - 4                   // Substraction
  3 * 4                   // Multiplication
  3 / 4                   // Division
  3 % 4                   // Modulus
  ```

- Comparison

  Examples:

  ```
  $i == $j
  $i != $j
  $i > $j
  $i >= $j
  $i < $j
  $i <= $j
  ```

B.4.5 String Operators

- String Concatenation

  Format:

  <string> + <string>

  Example:

  ```
  “abc” + “cde”                    // The result is “abccde”
  ```

B.4.6 List Operators

- Selection

  Format:

  <list>[<integer>]

  Examples:

  ```
  [2, 3, 9][2]                     // Return the second element, 3.
  [2, 3, 9][-1]                    // Return the last element, 9.
  [ 2][3]                          // Return UNDEF.
  [2, 3, 9][0]                     // Return UNDEF.
  ```

- Append

  Format:

  <list> + <list>

  Example:

  ```
  [2, 3, 9] + [3, 0, 3]            // The result is [2, 3, 9, 3, 0, 3].
  ```

- Removal

  Format:

  ```
  <list1> - <list2>                // Elements in list2 may not appear in list1;
  ```

  Example:

  ```
  [2, 3, 9] - [3]                  // The result is [2, 9].
  ```

- Membership

  Format:

  ```
  <list>.member(<element>)              // Return TRUE if <element> is a member of <list>.
  ```

B.4.7 Set Operators

- Union

  Format:

  <set> + <set>

Example:

```
[:2, 3, 5:] + [:4, 5:]          // The result is [:2, 3, 5, 4:]
[:4, 5, 2:] + [:9:]             // The result is [: 4, 5, 2, 9:]
```

- Intersection

Format:

*<set> * <set>*

Example:

```
[:3, 4, 5:] * [:2, 0, 6:]       // The result is [: :].
[:4, 5, 2:] *[:2:]              // The result is [:2:]
```

- Set Difference

Format:

<set> - <set>

Example:

```
[:4, 5, 2:] - [:2:]             // The result is [:4, 5:]
```

- Membership

Format:

```
<set>.member(<element>)         // Return TRUE if <element> is a member of <set>.
```

B.4.8 Message Sending

- Attributes of software unit

Format:

<software_unit>.<identifier>(<expression_list>)

Example:

```
:foo.build("UNIX", "DEBUG")     // Send the build message to software unit foo with
                                // arguments "UNIX" and "DEBUG."
```

- Re-send a message to an overwritten attribute.

Format:

RESEND.<identifier>(<expression_list>)

Example:

```
RESEND.initialize()             // Resend the initialize message to the parent class.
```

B.4.9 In-place Operations on Sets and Lists

These operations are different from the ones discussed in sections B.4.6 and B.4.7 in that they directly modify the sets or lists instead of creating new values.

- Insert

Format:

```
<list>.insert(<element>)        // Insert new element at the end of list.
```

Example:

```
$list.insert(3)
```

Format:

```
<set>.insert(<element>)         // Insert new element into a set.
```

Example

```
$set.insert(4)
```

• Remove

Format:

<list>.remove(<element>) // Remove first appearance of element.

Format:

<set>.remove(<element>) // Remove element if it is a member of set.

B.4.10 Function Call

Format:

<function-name> (<arg-list>)

Example:

FileSum(“/pro/poem/foo.C”)

B.5 Statements

• Compound Statement

Format:

{

<statement>

...

}

• Assignment

Format:

<local variable> = <expression>; *// Set local variable*

Example:

$I = 4;

Format:

<attribute> = <expression>; *// Set attribute*

Examples:

@attr = “good”;

@attr = UNDEF; // Unset an attribute

@parent = <prototype>; // Set parent

• Foreach

/* 2nd argument will be evaluated only once. */

Format:

foreach(<id>, <integer>)

<statement>

Example:

```
foreach ($I, 30)
    Print($I);
```

Format:

foreach(<id>, <list>)

<statement>

Example:

```
foreach ($I, [3, 4, 5])
    Print($I);
```

Format:

foreach(<id>, <set>)

<statement>

Example:

```
foreach ($I, {3, 4, 5})
    Print($I);
```

- While

Format:

while (<boolean expression>)
    <statement>

- If then else

Format 1:

if (<boolean expression>)
    <statement>
else
    <statement>

Example:

```
if ($i > 0)
    Print("Positive\n");
else
    Print("Negative\n");
```

Format 2:

if (<boolean expression>)
    <statement>

Example:

```
if ($i > 0)
    Print("Positive\n");
```

- Return value for an operation

Format:

return(<expression>);

Example:

```
return($i * 12);
```

- Message sending statement

Format:

<message sending expression>;

- Function call statement

Format:

<function call expression>;

B.6 Operation Definitions

The current implementation of VERSE requires that each operation be defined in a separate file. Operations do not have their own names and are referenced by the files that contain them.

Format:

operation(<argument1>, <argument 2>, ..., <argument n>)
{
    [<local_variable 1>, <local_variable 2>, ..., <local_variable n>]
    <statement 1>
    <statement 2>
    ...
    <statement n>
}

```
Example:
    operation(x, y)
    {
        [z]
        $z = $x + $y;
        return($z);
        }
```

B.7 Grammar

We summarize the syntax of VERSE by the following grammar, which can be directly used as the input of YACC.

```
attribute       : operation
                | expression
                ;

operation   : LX_OPERATION '(' optional_id_list ')'
                '{' local_vars stmt_list '}'
                ;
optional_id_list : /* empty */
                | id_list
                ;
local_vars  : /* empty */
                | '[' id_list ']'
                ;
id_list         : LX_ID
                | id_list ',' LX_ID
                ;
statement       : compound_stmt
                | assignment
                | set_attr
                | unset_attr
                | attr_delegation
                | foreach_stmt
                | while_stmt
                | if_stmt
                | func_stmt
                | msg_stmt
                | break_stmt
                | return_stmt
                ;
compound_stmt : '{' stmt_list '}'
                ;
stmt_list       : statement
                | stmt_list statement
                ;
assignment  : '$' LX_ID '=' expression ';'
                ;
set_attr        : '@' LX_ID '=' expression ';'
                ;
unset_attr      : '@' LX_ID '=' LX_UNDEF ';'
```

```
                ;
attr_delegation : '@' LX_ID LX_DELEGATE expression ';'
                ;
foreach_stmt : LX_FOREACH '(' '$' LX_ID ',' expression ')' statement
                ;
while_stmt    : LX_WHILE '(' expression ')' statement
                ;
if_stmt       : LX_IF '(' expression ')' statement LX_ELSE statement
                | LX_IF '(' expression ')' statement
                ;
func_stmt     : function_call ';'
                ;
msg_stmt      : msg_sending ';'
                |  msg_resending ';'
                ;
break_stmt    : LX_BREAK ';'
                ;
return_stmt   : LX_RETURN '(' ')' ';'
                | LX_RETURN '(' expression ')' ';'
                ;
expression_list :       /* empty */
                | expression
                | expression_list ',' expression
                ;
expression    : simple_expression
                | simple_expression LX_RELOP simple_expression
                | software_unit
                | LX_SELF
                ;
simple_expression : term
                | '-' term
                | simple_expression '+' simple_expression
                | simple_expression '-' simple_expression
                | simple_expression LX_OR simple_expression
                ;
term          : factor
                | term '*' term
                | term '/' term
                | term '%' term
                | term LX_AND term
                ;
factor        : variable
                | function_call
                | msg_sending
                | msg_resending
                | '[' expression_list ']'
                | LX_LSET expression_list LX_RSET
                | LX_INTEGER
                | LX_TRUE
                | LX_FALSE
                | LX_NULL
                | LX_STRING
```

```
                | '(' expression ')'
                | '!' factor
                ;
function_call : LX_ID '(' expression_list ')'
                ;
msg_sending : expression '.' LX_ID '(' expression_list ')'
                |  expression '.' LX_ID
                | '@' LX_ID '(' expression_list ')'
                | '@' LX_ID
                ;
msg_resending:  LX_RESEND '.' LX_ID '(' expression_list ')'
                | LX_RESEND '.' LX_ID
                ;
variable        : '$' LX_ID
                | variable '[' expression ']'
                ;
software_unit : ':' LX_ID
                | LX_STRING ':' LX_ID
                | ':' LX_ID '(' expression ')'
                | LX_STRING ':' LX_ID '(' expression ')'
                ;
```

B.8 Lexical Rules

```
LX_ID           : [a-zA-Z_][a-zA-Z0-9_]*
LX_INTEGER      : [0-9]+
LX_STRING       : \"([^\n"\\]|(\\.)|(\\\n))*\"
LX_OR           : "||"
LX_AND          : "&&"
LX_DELEGATE : ":="
LX_LSET         : "[:"
LX_RSET         : ":]"
LX_RELOP        : "=="
                | "!="
                | ">="
                | "<="
                | ">"
                | "<"
LX_OPERATION : "operation"
LX_IF           : "if"
LX_ELSE         : "else"
LX_WHILE        : "while"
LX_FOREACH      : "foreach"
LX_RETURN       : "return"
LX_BREAK        : "break"
LX_SELF         : "SELF"
LX_RESEND       : "RESEND"
LX_TRUE         : "TRUE"
LX_FALSE        : "FALSE"
LX_UNDEF        : "UNDEF"
LX_NULL         : "NULL"
```

B.9 Comments

```
WHITE       : [\ \t\n]+
CMMT        : "/*"(((“*”+[^*/])|[^*])*)(“*”+”/”)
```

B.10 Built-in Functions

- FileExist(String: <file_name>): Boolean;
  Return true if <filename> exists.
- FileMTime(String: <file_name>): Integer;
  Return time of last data modification on <file_name>.
- FileSum(String: <file_name>): Integer;
  Return the circular checksum of <file_name>.
- NewSU(SoftwareUnit: <class_name>, SoftwareUnit: <vpred>): SoftwareUnit;
  Return a new software unit that belongs to the class <class_name> and with <vpred> as its version predecessor.
- Print(Any: <data1>, Any <data 2>, ..., Any <data n>): Void;
  Print <data1> through <data n> to the standard output device. Each data item may be an integer, a string, a software unit, a set, or list.
- RcsCheckin(String: <file_name>, String: <rcs_file_name>): String;
  Check in <file_name> to the RCS archive file <rcs_file_name>. Return version number of the new version.
- RcsCheckout(String:<file_name>, String:<version>, String:<rcs_file_name>): Boolean;
  Check out version <version> of <file_name> from the RCS archive file <rcs_file_name>. Return TRUE if the operation is successful.
- StringSum(String: <string>): Integer;
  Return the circular check-sum of <string>;
- Sys(String: <string>): Integer;
  Issue <string> as a shell command. Return the exit status of the command.

Appendix C

An Example of VERSE

All the classes of software units in POEM are defined in the VERSE language, which is discussed in Chapter 6 and Appendix B. In this appendix, we show an example of defining a class with VERSE.

The example we choose is the system-defined class DevelopingUnit. It represents all the software units that are still under development. It is the contrary of LibUnit, which represents all the software units that serve as the interfaces to existing library modules. As illustrated in Figure C-1, DevelopingUnit is the subclass of the root class SoftwareUnit and has two system-defined classes CUnit and C++Unit.

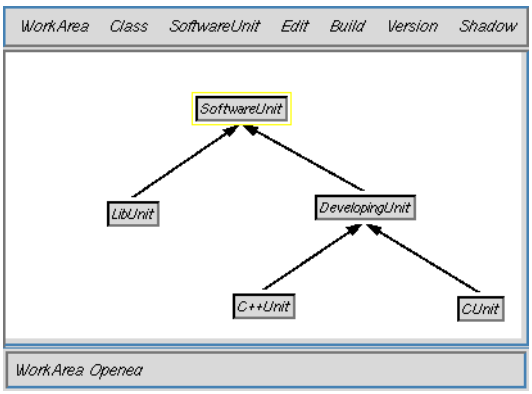


Figure C-1: System-defined classes

C.1 List of Attributes

We keep all the files related to the definition of DevelopingUnit under a separate directory. There are VERSE files that define individual operations and a file called “Attributes.lst” that lists all the attributes of DevelopingUnit. After we prepare all these files, we select New under the Class

menu to create a new class button, and then select Load_Definition under the Class menu to load the files from this directory.

The content of “Attributes.lst” is listed below:

```
name                    S   “DevelopingUnit”
is_pure_prototype       S   TRUE
fixed                   S   FALSE
build_paras             S   [:”Debug”,”Profile”,”Optimize1”,”Optimize2”,”StaticLinking”:]
available_compilers     S   [:”SUN_CC”,”gcc”:]
compiler                S    “SUN_CC”
build_flags             S   [:”Debug”:]
compile_cmd_sum         S   -1
old_compile_cmd_sumS    -1
link_cmd_sum            S   -1
tfile_sum               S   -1
old_tfile_sum           S   -1
dfile_sum               S   -1
exe_args                S   ““
dfile                   S   ““
Init                    F   Init.vrs
EditSource              F   EditSource.vrs
SetName                 F   SetName.vrs
Interface               F   Interface.vrs
GlobalDef               F   GlobalDef.vrs
Compile                 F   Compile.vrs
Build                   F   Build.vrs
BuildExe                F   BuildExe.vrs
RunExe                  F   RunExe.vrs
derived                 F   derived.vrs
fixed_derived           F   fixed_derived.vrs
modified                F   modified.vrs
mderived                F   mderived.vrs
Clean                   F   Clean.vrs
clean                   F   clean.vrs
libraries               F   libraries.vrs
SetSUFlags              F   SetSUFlags.vrs
SetSubsystemFlags       F   SetSubsystemFlags.vrs
set_flags               F   set_flags.vrs
SetSUCompiler           F   SetSUCompiler.vrs
SetSubsystemCompilerF   SetSubsystemCompiler.vrs
set_compiler            F   set_compiler.vrs
SetExeArgs              F   SetExeArgs.vrs
```

Each line in this file defines an attribute of DevelopingUnit, and each line contains three fields separated by spaces. The first field is name of the attribute. If the second field is ‘S,’ then the third field gives the definition of the attribute right away. If the second field is ‘F,’ then the third field specifies the file where the definition of the attribute is stored.

C.2 Operation Definitions

C.2.2 Init.vrs

```
/* DevelopingUnit::Init() */
operation()
{
	RESEND.Init();
	@dfile = @dir + @name + “.doc”;
	Sys(“touch “ + @dfile);
	@dfile_sum = -1;
	Print(“Initialize DevelopingUnit...\n”);
}
```

C.2.3 EditSource.vrs

```
/* DevelopingUnit::EditSource() */
operation()
{
	@checkout_src();
	Sys(“gnuclient -q “ + @ifile + “ &”);
	Sys(“gnuclient -q -p 22493 “ + @sfile + “ &”); /*22493 is the gnuserv port # used by the impl. editor*/
	Sys(“gnuclient -q -p 22494 “ + @gfile + “ &”); /*22494 is the gnuserv port # used by the globals editor*/
	Sys(“gnuclient -q -p 22495 “ + @dfile + “ &”); /*22495 is the gnuserv port # used by the doc editor*/
}
```

C.2.4 SetName.vrs

```
/* DevelopingUnit::SetName() */
operation(new_name)
{
	Sys(“gnudoit ‘(kill-buffer \”” + @ifile + “\”)’”);
	Sys(“gnudoit -p 22494 ‘(kill-buffer \”” + @gfile + “\”)’”);
	Sys(“mv “ + @ifile + “ “ + @dir + $new_name + “.h”);
	Sys(“mv “ + @gfile + “ “ + @dir + $new_name + “_g.h”);
	Sys(“mv “ + @dir + @name + “.o “ + @dir + $new_name + “.o”);
	Sys(“mv “ + @dir + @name + “ “ + @dir + $new_name);
	@name = $new_name;
	@ifile = @dir + @name + “.h”;
	@gfile = @dir + @name + “_g.h”;
}
```

C.2.5 Interface.vrs

```
/* DevelopingUnit::Interface() */
operation(lib_hdr_path, lib_hdrs)
{
	[ifile, result, child]
```

```
        $result = [: :];

        foreach ($child, @trans_uses)
            $result = $result + $child.Interface($lib_hdr_path, $lib_hdrs);

        @checkout_interface();
        $result.insert(@ifile);

        return($result);
    }
```

C.2.6 GlobalDef.vrs

```
    /* DevelopingUnit::GlobalDef() */
    operation(lib_hdr_path, lib_hdrs)
    {
        [gfile, result, parent, i]

        $result = [: :];

        foreach ($parent, @trans_partof)
            $result = $result + $parent.GlobalDef($lib_hdr_path, $lib_hdrs);

        @checkout_src();
        $result.insert(@gfile);
        foreach ($i, @Interface($lib_hdr_path, $lib_hdrs))
            $result.insert($i);

        return($result);
    }
```

C.2.7 Compile.vrs

```
    /* DevelopingUnit::Compile() */
    operation()
    {
        Print(“Compiling...\n”);
        @derived();
    }
```

C.2.8 Build.vrs

```
    /* DevelopingUnit::Build() */
    operation()
    {
        @mderived([: :], [: :], [: :]);
```

```
}
```

C.2.9 BuildExe.vrs

```
/* DevelopingUnit::BuildExe() */
operation()
{
	[ofs, f, cmd, xfile, outdate, libpaths, libs, new_cmd_sum]

	$outdate = FALSE;
	$libpaths = [: :];
	$libs = [::];
	$ofs = @mderived([: :], $libpaths, $libs);

	if ($ofs == -1)
	 return();

	$xfile = @dir + @name;
	if (FileExist($xfile)) {
		foreach ($f, $ofs)
			if (FileMTime($xfile) < FileMTime($f)) {
				$outdate = TRUE;
				break;
			}
		}
	else
		$outdate = TRUE;

	$cmd = @link_cmd($xfile, $ofs, $libpaths, $libs);

	$new_cmd_sum = StringSum($cmd);
	if ($new_cmd_sum != @link_cmd_sum) {
		$outdate = TRUE;
		@link_cmd_sum = $new_cmd_sum;
	}

	if ($outdate)
	 Sys($cmd);
}
```

C.2.10 RunExe.vrs

```
/* DevelopingUnit::RunExec() */
operation()
{
	[xfile]

	$xfile = @dir + @name;
```

```
    if (FileExist($xfile))
        Sys($xfile + " " + @exe_args );
    else
        Print($xfile, " : No such file.\n");
}
```

C.2.11 derived.vrs

```
/* DevelopingUnit::derived() */
operation()
{
    [outdate, child, parent, i, ofile, tfile, cfile, hfile,
     cc, new_cmd_sum, includes, lib_hdr_path, lib_hdrs, new_tfile_sum]

    if (@fixed)
        return(@fixed_derived());

    $outdate = FALSE;
    $ofile = @dir + @name + ".o";
    $tfile = @dir + "tt.c";

    if (!FileExist($ofile))
        $outdate = TRUE;
    else if (FileMTime($ofile) < FileMTime(@sfile))
        $outdate = TRUE;
    else if (FileMTime($ofile) < FileMTime(@ifile))
        $outdate = TRUE;

    $includes = [: :];
    $lib_hdr_path = [: :];
    $lib_hdrs = [: :];
    foreach ($child, @uses)
        foreach ($i, $child.Interface($lib_hdr_path, $lib_hdrs)) {
            if (!$outdate)
                if (FileMTime($ofile) < FileMTime($i))
                    $outdate = TRUE;
            $includes.insert($i);
        }
    foreach ($child, @trans_uses)
        foreach ($i, $child.Interface($lib_hdr_path, $lib_hdrs)) {
            if (!$outdate)
                if (FileMTime($ofile) < FileMTime($i))
                    $outdate = TRUE;
            $includes.insert($i);
        }
    foreach ($parent, @partof)
        foreach ($i, $parent.GlobalDef($lib_hdr_path, $lib_hdrs)) {
            if (!$outdate)
                if (FileMTime($ofile) < FileMTime($i))
                    $outdate = TRUE;
```

```
                $includes.insert($i);
        }
    foreach ($parent, @trans_partof)
        foreach ($i, $parent.GlobalDef($lib_hdr_path, $lib_hdrs)) {
            if (!$outdate)
                if (FileMTime($ofile) < FileMTime($i))
                    $outdate = TRUE;
            $includes.insert($i);
        }

    $cc = @compile_cmd($tfile, $ofile, $lib_hdr_path);
    $new_cmd_sum = StringSum($cc);
    if ($new_cmd_sum != @compile_cmd_sum) {
        $outdate = TRUE;
        @compile_cmd_sum = $new_cmd_sum;
    }

    Sys("echo > " + $tfile);
    foreach ($i, $lib_hdrs)
        Sys("echo '#include <" + $i + ">' >> " + $tfile);
    foreach ($i, $includes)
        Sys("echo '#include \"" + $i + "\"' >> " + $tfile);
    Sys("echo '#include \"" + @ifile + "\"' >> " + $tfile);
    Sys("echo '#include \"" + @sfile + "\"' >> " + $tfile);
    if ($outdate)
        @tfile_sum = FileSum($tfile);
    else {
        $new_tfile_sum = FileSum($tfile);
        if ($new_tfile_sum != @tfile_sum) {
            @tfile_sum = $new_tfile_sum;
            $outdate = TRUE;
        }
    }
    if ($outdate) {
        if (Sys($cc)) {
            Print("Compilation failed.\n\n");
            return(-1);
        }
        else
            Print("Compilation succeeded.\n\n");
    }
    return($ofile);
}
```

C.2.12 fixed_derived.vrs

```
/* DevelopingUnit::fixed_derived() */
operation()
{
    [child, parent, i, ofile, tfile,
```

```
 cc, includes, lib_hdr_path, lib_hdrs]

@checkout_src();

$ofile = @dir + @name + ".o";

if (!FileExist($ofile)) {
    $tfile = @dir + "tt.c";
    $includes = [: :];
    $lib_hdr_path = [: :];
    $lib_hdrs = [: :];
    foreach ($child, @uses)
        foreach ($i, $child.Interface($lib_hdr_path, $lib_hdrs))
            $includes.insert($i);
    foreach ($child, @trans_uses)
        foreach ($i, $child.Interface($lib_hdr_path, $lib_hdrs))
            $includes.insert($i);
        foreach ($parent, @partof)
            foreach ($i, $parent.GlobalDef($lib_hdr_path, $lib_hdrs))
                $includes.insert($i);
    foreach ($parent, @trans_partof)
        foreach ($i, $parent.GlobalDef($lib_hdr_path, $lib_hdrs))
            $includes.insert($i);

    Sys("echo > " + $tfile);
    foreach ($i, $lib_hdrs)
        Sys("echo '#include <" + $i + ">' >> " + $tfile);
    foreach ($i, $includes)
        Sys("echo '#include \"" + $i + "\"' >> " + $tfile);
    Sys("echo '#include \"" + @ifile + "\"' >> " + $tfile);
    Sys("echo '#include \"" + @sfile + "\"' >> " + $tfile);
    $cc = @compile_cmd($tfile, $ofile, $lib_hdr_path);

    if (Sys($cc)) {
        Print("Compilation failed.\n\n");
        return(-1);
    }
    else
        Print("Compilation succeeded.\n\n");
}

return($ofile);
}
```

C.2.13 modified.vrs

```
/* DevelopingUnit::modified() */
operation()
{
    Print("Checking " + @name + "\n");
```

```
        if (FileSum(@sfile) == @sfile_sum)
            if ((FileSum(@ifile) == @ifile_sum) && (FileSum(@gfile) == @gfile_sum))
                if ((@compile_cmd_sum == @old_compile_cmd_sum) &&
                    (@tfile_sum == @old_tfile_sum))
                    return(FALSE);

        Print(@name + “is modified!\n”);
        return(TRUE);
    }
```

C.2.14 mderived.vrs

```
    /* DevelopingUnit::mderived() */
    operation(visited, libpaths, libs)
    {
        [child, md, cderived, lderived]

        $md = [: :];
        $visited.insert(@id);

        foreach ($child, @uses)
            if (!$visited.member($child.id)) {
                $cderived = $child.mderived($visited, $libpaths, $libs);
                if ($cderived == -1)
                    return(-1);
                $md = $md + $cderived;
            }
        foreach ($child, @trans_uses)
            if (!$visited.member($child.id)) {
                $cderived = $child.mderived($visited, $libpaths, $libs);
                if ($cderived == -1)
                    return(-1);
                $md = $md + $cderived;
            }
        $lderived = @derived;
        if ($lderived == -1)
            return(-1);
        $md.insert($lderived);
        return($md);
    }
```

C.2.15 Clean.vrs

```
    /* DevelopingUnit::Clean() */
    operation()
    {
        @clean([: :]);
    }
```

C.2.16 clean.vrs

```
/* DevelopingUnit::clean() */
operation(visited)
{
	[child]

	$visited.insert(@id);

	foreach ($child, @uses)
		if (!$visited.member($child.id))
			$child.clean($visited);
	foreach ($child, @trans_uses)
		if (!$visited.member($child.id))
			$child.clean($visited);
	if (@fixed)
		Sys("rm -f " + @dir + "*");
	else {
		Sys("rm -f " + @dir + "*.o");
	 Sys("rm -f " + @dir + @name);
	}
}
```

C.2.17 libraries.vrs

```
/* DevelopingUnit::libraries() */
operation(visited)
{
	[child, libs]

	$libs = [: :];
	$visited.insert(@id);

	foreach ($child, @uses)
		if (!$visited.member($child.id))
			$libs = $libs + $child.libraries($visited);

	foreach ($child, @trans_uses)
		if (!$visited.member($child.id))
			$libs = $libs + $child.libraries($visited);

	return($libs);
}
```

C.2.18 SetSubsystemFlags.vrs

```
/* DevelopingUnit::SetSubsystemFlags() */
operation(flags)
{
	@set_flags([: :], $flags);
}
```

C.2.19 set_flags.vrs

```
/* DevelopingUnit::set_flags() */
operation(visited, new_flags)
{
	[child]

	$visited.insert(@id);
	if (@build_flags != $new_flags)
		@build_flags = $new_flags;

	foreach ($child, @uses)
		if ((!$visited.member($child.id)) && (!$child.fixed))
			$child.set_flags($visited, $new_flags);
	foreach ($child, @trans_uses)
		if ((!$visited.member($child.id)) && (!$child.fixed))
			$child.set_flags($visited, $new_flags);
}
```

C.2.20 SetSUCompiler.vrs

```
/* DevelopingUnit::SetSUCompiler() */
operation(new_compiler)
{
	if (@compiler != $new_compiler) {
		@compiler = $new_compiler;
	 Print("Set compiler = ", @compiler, "\n");
	}
}
```

C.2.21 SetSubsystemCompiler.vrs

```
/* DevelopingUnit::SetSubsystemCompiler() */
operation(compiler_name)
{
	@set_compiler([: :], $compiler_name);
}
```

C.2.22 set_compiler.vrs

```
/* DevelopingUnit::set_compiler() */
```

```
operation(visited, new_compiler)
{
	[child]

	$visited.insert(@id);
	if (@compiler != $new_compiler)
		@compiler = $new_compiler;

	foreach ($child, @uses)
		if ((!$visited.member($child.id)) && (!$child.fixed))
			$child.set_compiler($visited, $new_compiler);

	foreach ($child, @trans_uses)
		if ((!$visited.member($child.id)) && (!$child.fixed))
			$child.set_compiler($visited, $new_compiler);
}
```

C.2.23 SetExeArgs.vrs

```
/* DevelopingUnit::SetExeArgs() */
operation(args)
{
	@exe_args = $args;
}
```

Bibliography

[1] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Holzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko, "The SELF 3.0 Programmer's Reference Manual," Sun Microsystems, Inc., Mountain View, California, USA.

[2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

[3] Larry Allen, Gary Fernandez, Kenneth Kane, David Leblang, Debra Minard, and John Posner, "ClearCase MultiSite: Supporting Geographically-Distributed Software Development," in *Proceedings of the Fifth International Workshop on Software Configuration Management*, April 1995.

[4] Atria Software, Inc., "ClearCase Product Summary," Atria Software, Inc., 24 Prime Park Way, Natick, MA 01760, 1995.

[5] Atria Software, Inc., "ClearCase Architecture and Database," Atria Software, Inc., 24 Prime Park Way, Natick, MA 01760, August 1993

[6] Naser S. Barghouti, "Supporting Cooperation in the MARVEL Process-Centered SDE," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Herbert Weber, ed.), pp. 21-31, December 1992. Published as *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 5, December 1992.

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

[8] N. Belkhair and J. Estublier, "Experience with a Data Base of Programs," in *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pp. 84-91, December 1986.

[9] Noureddine Belkhatir, Jacky Estublier, and Walcelio L. Melo, "Adele2: A Support to Large Software Development Process," in *Proceedings of the First International Conference on the Software Process*, pp. 159-170, October 1991.

[10] Israel Z. Ben-Shaul and Gail E. Kaiser, "A Paradigm for Decentralized Process Modeling and its Realization in the OZ Environment," in *Proceedings of the 16th International Conference on Software Engineering*, pp. 179-188, Sorrento, Italy, May 1994.

[11] Jon Bentley, "Programming Pearls," *Communications of the ACM*, Vol. 29, No. 5, pp. 364-369, May 1986, and Vol. 29, No. 6, pp. 471-483, June 1986,

[12] Brian Berliner, "CVS II: Parallelizing Software Development," Prisma, Inc., 5465 Mark Dabling Blvd, Colorado Springs, CO 80918.

[13] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas, "An Overview of PCTE and PCTE+," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 248-257, November 1988. Published as *ACM SIGSOFT Software Engineering Notes*, Vol. 13, No. 5, November 1988, and *ACM SIGPLAN Notices*, Vol. 24, No. 2, February 1989.

[14] Mark R. Brown and John R. Ellis, "Bridges: Tools to Extend the Vesta Configuration Management System," DEC System Research Center Research Report No. 108, June 1993.

[15] P. Brown, "Integrated Hypertext and Program Understanding Tools," in *IBM System Journal*, Vol. 30, No. 3, pp. 363-392, 1991.

[16] Todd Brunhoff and Jim Fulton, "Imake – C Preprocessor Interface to the Make Utility," Tektronix and MIT Project Athena, Cambridge, MA.

[17] Todd Brunhoff and Jim Fulton, "Makedpend - Create Dependencies in Makefiles," Tektronix and MIT Project Athena, Cambridge, MA.

[18] John N. Buxton and Larry E. Druffel, "Requirements for An Ada Programming Support Environment: Rationale for STONEMAN," in *Proceedings of COMPSAC 80*, pp. 66-72, 1980.

[19] Martin R. Cagan, "Software Configuration Management Redefined," CaseWare, Inc., 108 Pacifica, Irvine, CA 92718, March 1992.

[20] Sheng-Yang Chiu and Roy Levin, "The Vesta Repository: A File System Extension for Software Development," DEC System Research Center Research Report No. 106, June 1993.

[21] W. Courington, "The Network Software Environment," Technical Report Sun FE 197-0, Sun Microsystems Inc., February 1989.

[22] Michael L. Creech, Dennis F. Freeze and Martin L. Griss, "Using Hypertext in Selecting Reusable Software Components," in *Hypertext '91 Proceedings*, pp. 25-38, December 1991.

[23] Susan Dart, "Concepts in Configuration Management Systems," in *Proceedings of the 3rd International Workshop on Software Configuration Management* (Peter Feiler, ed.), pp. 1-18, June 1991.

[24] Mark Dowson, "ISTAR – An Integrated Project Support Environment," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 27-33, December 1986.

[25] ECMA, "A Reference Model for Frameworks of Software Engineering Environments (Version 2). *ECMA Report Number TR/55 (Version 2), NIST Report Number SP 500-201*, December 1991.

[26] J. Estublier, S. Ghoul, and S. Krakowiak, "Preliminary Experience with a Configuration Control System for Modular Programs," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering*

Symposium on Practical Software Develpment Environment, pp. 149-156, April 1984.

[27] Jacky Estublier, "A Configuration Manager: The Adele Data Base of Programs," in *Workshop on Software Engineering Environments for Programming-in-the-Large*, June 1985.

[28] Jacky Estublier and Jean-Marie Favre, "Structuring Large Versioned Software Products," in *Proceedings of IEEE COMPSAC 89*, pp. 404-411, September 1989.

[29] Jacky Estublier, Noureddine Belkhatir, Mohamed A. Nacer, and Walcelio L. Melo, "Process-centered SEE and Adele," in *Proceedings of the 5th international Workshop on Computer-Aided Software Engineering* (CASE '92), pp. 156-165 July 1992.

[30] Jacky Estublier and Rubby Casallas, "The Adele Configuration Manager," in *Configuration Management* (W. F. Tichy, ed.), John Wiley & Son Ltd., 1994.

[31] Peter H. Feiler, "Configuration Management Models in Commercial Environments," Technical Report CMU/SEI-91-TR-7, Software Engineering Institute, Carnegie Mellon University, March 1991.

[32] Stuart I. Feldman, "Make – a Program for Maintaining Computer Programs," in *Software - Practice & Experience*, 9(4), pp. 255-265, April 1979.

[33] James C. Ferrans, David W. Hurst, Michael A. Sennett, Burton M. Covnot, Wenguang Ji, Peter Kajka, and Wei Ouyang, "HyperWeb: A Framework for Hypermedia-Based Environments," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Herbert Weber, ed.), pp. 1-10, December 1992. Published as *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 5, December 1992.

[34] Ferdinando Gallo, Regis Minot, and Ian Thomas, "The Object Management System of PCTE as a Software Engineering Database Management System," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 12-15, December 1986.

[35] David Garlan, Linxi Cai, and Robert L. Nord, "A transformational Approach to Generating Application-Specific Environments," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Herbert Weber, ed.), pp. 68-77, December 1992. Published as *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 5, December 1992.

[36] William G. Griswold and David Notkin, "Automated Assistance for Program Restructuring," *ACM Transactions on Software Engineering and Methodology*, Vol. 2, No. 3, pp. 228-269, July 1993.

[37] William G. Griswold, "Direct Update of Data Flow Representations for a Meaning-Preserving Program Restructuring Tool," in *Proceesdings of the First ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pp. 42-55, December, 1993.

[38] A. Nico Habermann and David Notkin, "Gandalf: Software Development Environments," in *IEEE Transactions on Software Engineering*, Vol. 12, No. 12, pp.1117-1127, December 1986.

[39] Christine B. Hanna and Roy Levin, "The Vesta Language for Configuration Management," DEC System Research Center Research Report No. 107, June 1993.

[40] William H. Harrison, John Shilling, and Peter Sweeney, "Good News, Bad News: Experience Building

a Software Development Environment Using the Object-Oriented Paradigm," in *ACM OOPSLA '89 Proceedings*, pp. 85-94, October, 1989.

[41] Hewlett-Packard Company, "Exploring HP SoftBench: a Beginner's Guide," Hewlett-Packard Company, Ft. Collins, Colorado, 1989.

[42] Susan Horwitz, "Identifying the Semantics and Textual Differences Between Two Versions of a Program," in *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 234-245, June 1990.

[43] Scott E. Hudson and Roger King, "The Cactis Project: Database Support for Software Environments," in *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, pp. 709-719, June 1988.

[44] Simon M. Kaplan, William J. Tolone, Alan M. Carroll, Douglas P. Bogia and Celsina Bignoli, "Supporting Collaborative Software Development with ConversationBuilder," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Herbert Weber, ed.), pp. 11-20, December 1992. Published as ACM SIGSOFT Software Engineering Notes, Vol. 17, No. 5, December 1992.

[45] Randy H. Katz, "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys*, Vol. 22, No. 4, pp. 375-408, December 1990.

[46] Setrag N. Khoshafian and George P. Copeland, "Object Identity," *SIGPLAN Notices*, Vol. 21, No. 11, pp. 406-415, November 1986.

[47] Gail E. Kaiser and Peter H. Feiler, "Intelligent Assistance for Software Development and Maintenance," *IEEE Software*, May 1988.

[48] Donald E. Knuth, "Literate Programming," *The Computer Journal*, Vol. 27, No. 2, pp. 97-111, 1984.

[49] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinred, "The ObjectStore Database System," in *Communications of the ACM*, Vol. 34, No. 10, pp. 50-63, October 1991.

[50] Ronald Lange and Robert W. Schwanke, "Software Architecture Analysis: A Case Study," in *Proceedings of the 3rd International Workshop on Software Configuration Management*, pp. 19-28, June, 1991.

[51] David B. Leblang and Robert P. Chase, Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," *SIGPLAN Notices*, Vol. 19, No. 5, pp. 104-113, April 1984.

[52] Roy Levin and Paul R. McJones, "The Vesta Approach to Precise Configuration of Large Software Systems," DEC System Research Center Research Report No. 105, June 1993.

[53] Henry Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Languages," in *ACM OOPSLA '86*, pp. 214-223, September, 1986.

[54] Yi-Jing Lin and Steven P. Reiss, "An Object-Oriented Approach to Designing Programming Environments," Technical Report CS-93-38, Department of Computer Science, Brown University, September 1993.

[55] Yi-Jing Lin, "VERSE - The DML of POEM," Department of Computer Science, Brown University, February, 1995.

[56] Yi-Jing Lin and Steven P. Reiss, "Configuration Management in terms of Modules," in *Proceedings of the Fifth International Workshop on Software Configuration Management*, April 1995.

[57] Yi-Jing Lin and Steven P. Reiss, "Configuration Management with Logical Structures," Technical Report CS-95-23, Department of Computer Science, Brown University, August 1995.

[58] Axel Mahler and Andreas Lampen, "An Integrated Toolset for Engineering Software Configurations," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 191-200, November 1988. Published as *ACM SIGSOFT Software Engineering Notes*, Vol. 13, No. 5, November 1988, and *ACM SIGPLAN Notices*, Vol. 24, No. 2, February 1989.

[59] Scott Meyers and Steven P. Reiss, "An Empirical Study of Multiple-View Software Development," in *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments* (Herbert Weber, ed.), pp. 47-57, December 1992.

[60] Naftaly H. Minsky and David Rozenshtein, "A Software Development Environment for Law-Governed Systems," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 65-75, November 1988. Published as *ACM SIGSOFT Software Engineering Notes*, Vol. 13, No. 5, November 1988, and *ACM SIGPLAN Notices*, Vol. 24, No. 2, February 1989.

[61] Thomas M. Morgan, "Configuration Management and Version Control in the Rational Programming Environment," in *Ada in Industry – Proceedings of the Ada-Europe International Conference*, pp. 17-28, June 1988.

[62] H. A. Muller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji, "A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models," in *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments* (Herbert Weber, ed.), pp. 88-98, December 1992.

[63] Gail C. Murphy and David Notkin, "Lightweight Source Model Extraction," in *Proceedings of the Fifth ACM SIGSOFT Symposium on the Foundation of Software Engineering*, October 1995.

[64] William F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, 1992.

[65] Harold Ossher and William Harrison, "Support for Change in RPDE3," in *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments* (Richard Taylor, ed.), pp. 218-228, December 1990.

[66] Leon Osterweil, "Software Processes Are Software Too," in *Proceedings of the 9th International Conference on Software Engineering*, pp. 2-13, Monterey CA, March-April 1987.

[67] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," in *Communications of the ACM*, pp. 1053-1058, Vol. 15, No. 12, December 1972.

[68] PROCASE Corporation, "SMARTsystem, Technical Overview," PROCASE Corporation, 3130 De La Cruz Boulevard, Suite 100, Santa Clara, CA 95054, 1989.

[69] Steven P. Reiss, "Connecting Tools Using Message Passing in the FIELD Program Development Envi-

ronment," *IEEE Software*, pp. 57-67, July 1990.

[70] Steven P. Reiss, *The FIELD Programming Environment: A Friendly Integrated Environment for Learning and Development*, Kluwer Academic Publishers, Norwell, Massachusetts, USA, 1995.

[71] Hal Render and Roy Campbell, "An Object-Oriented Model of Software Configuration Management," in *ACM Proceedings of the 3rd International Workshop on Software Configuration Management*, pp. 127-139, June 1991.

[72] Marc J. Rochkind, "The Source Code Control System," *IEEE Transactions on Software Engineering*, pp. 364-370, Vol. 1 No. 4, December 1975.

[73] James Rumbaugh, "Controlling Propagation of Operations Using Attributes on Relations," *ACM OOPSLA '88 Proceedings*, pp. 285-296, September 1988.

[74] Ian Simmonds, "Configuration Management in the PACT Software Engineering Environment," in *Proceedings of the 2nd International Workshop on Software Configuration Management* (Peter H. Feiler, ed.), pp. 118-121, October, 1989.

[75] Software Maintenance & Development Systems, Inc., "Aide-de-Camp, Product Overview," Software Maintenance & Development Systems, Inc., 200 Baker Avenue, Suite 300, Concord, Massachusetts 01742, August 1993.

[76] Lynn Andrea Stein, "Delegation is Inheritance," in *ACM OOPSLA '87*, pp. 138-146, October 1987.

[77] Lynn Andrea Stein, Henry Lieberman, and David Ungar, "A Shared View of Sharing: The Treaty of Orlando," *Concepts, Applications and Databases* (Kim and Lochovsky, eds.), pp. 31-48, Addison Wesley, Reading, Massachusetts, 1989.

[78] Sun Microsystems, Inc., "SPARCworks/TeamWare Solutions Guide," Sun Microsystems, Inc., 2550 Garcia Ave., Mountain View, CA, 94043-1100, USA, May 1995.

[79] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michael Young, "Foundations for the Arcadia Environment Architecture," in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Peter Henderson, ed.), pp. 1-13, November 1988. Published as *ACM SIGSOFT Software Engineering Notes*, Vol. 13, No. 5, November 1988, and *ACM SIGPLAN Notices*, Vol. 24, No. 2, February 1989.

[80] Warren Teitelman, "A Tour Through Cedar," in *IEEE Software*, pp. 44-73, April 1984.

[81] Ian Thomas, "Tool Integration in the Pact Environment," in *Proceedings of the 11th International Conference on Software Engineering*, pp. 13-22, 1989.

[82] Walter F. Tichy, "RCS – A System for Version Control," *Software – Practice & Experience*, Vol. . 15, No. 7, pp. 637-654, July 1985.

[83] Douglas Wiebe, "Generic Software Configuration Management: Theory and Design," Technical Report 90-07-03, Computer Science Department, University of Washington, Seattle, WA 98195, 1990.

[84] David Ungar and Randall B. Smith, "Self: The Power of Simplicity," in *ACM OOPSLA '87*, October

1987.

[85] Peter Wegner, “Concepts and Paradigms of Object-Oriented Programming,” OOPS Messenger, Vol. 1, No. 1, pp. 7-87, August 1990.

[86] Bernhard Westfechtel, “Structure-Oriented Merging of Revisions of Software Documents,” in *Proceedings of the 3rd International Workshop on Software Configuration Management*, pp. 68-79, June 1991.

[87] David Whitgift, *Methods and Tools for Software Configuration Management*, Wiley Series in Software Engineering Practice, John Wiley & Sons Ltd. England, 1991.

[88] Jurgen F. H. Winkler, “Language Constructs and Library Support for Families of Large Ada Programs,” in *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pp. 17-28, June 1985.

[89] Wuu Yang, Susan Horwitz, and Thomas Reps, “A Program Integration Algorithm that Accommodates Semantics-Preserving Transformations,” in Proceedings of the Fourth ACM SIGSOFT Symposium on Practical Softwarwe Development Environments, pp. 133-143, November 1990.

[90] Wuu Yang, Susan Horwitz, and Thomas Reps, “A New Program Integration Algorithm,” TR-899, Computer Sciences Department, University of Wisconsin, Madison, December 1989.

[91] Wuu Yang, “Semantics-based Program Integration,” TR-962, Computer Sciences Department, University of Wisconsin, Madison, August 1990.

[92] Stanley B. Zdonik, “Version Management in an Object-Oriented Database,” in *Advanced Programming Environments* (R. Conradi, T. M. Didriksen, and D. H. Wanvik, eds.), pp. 405-422, 1986.

[93] Stanley B. Zdonik and David Maier (eds). *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., Palo Alto, CA, 1990.