Configuration Management with Logical Structures

Yi-Jing Lin
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
yjl@watson.ibm.com

Steven P. Reiss
Department of Computer Science
Box 1910, Brown University
Providence, RI 02912, USA
spr@cs.brown.edu

Abstract

When designing software, programmers usually think in terms of modules that are represented as functions and classes. But using 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.

We present a framework for programming environments that handles versions and configurations directly in terms of the functions and classes in source code. We show that with this framework, configuration management issues in software reuse and cooperative programming become easier. We also present a prototype environment that has been developed to verify our ideas.

1 Introduction

Modern programming languages support constructs like functions and classes that let programmers decompose source programs into modules. However, existing programming environments do not allow programmers to handle *configuration management* (CM) directly with these functions and classes. System building and version control 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 specifies the dependencies between source files and object files.

In this paper we present a framework for programming environments that handles configuration management directly in terms of the 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 this ability can make the configuration management issues in software reuse and cooperative programming easier. 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.

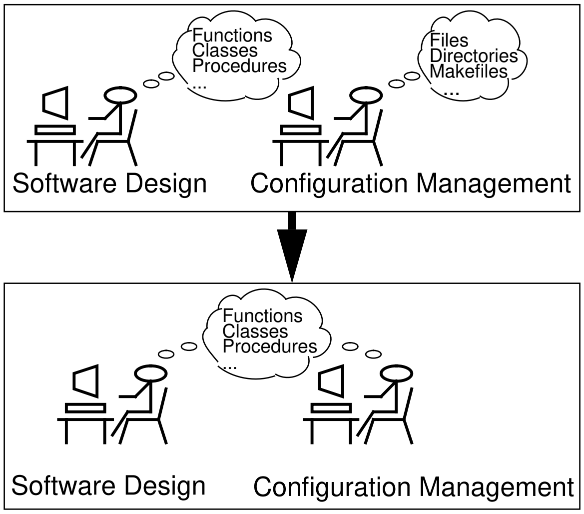


Figure 1: From CM with files to CM with logical structures

There are limitations to our approach, though. Since our framework is based on the modularization of programs, it cannot properly handle documents generated before a project is decomposed into modules. Our environment is designed to support configuration management in software design, implementation, and maintenance, but not for 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 doing some significant transformation.

2 The Problem

System building and version control are two of the most

This work is conducted at Brown University and funded by IBM Graduate Fellowship, the National Science Foundation under Grant Number CCR-9113226, ARPA order 8225, and ONR grant N00014-91-J-4052

important problems in configuration management. Higher-level activities like cooperative programming, software reuse, and software maintenance all rely on good support to handle these two problems. Although existing configuration management systems support a rich and diverse set of 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 sub-file 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 that 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 for those files too. Furthermore, since different versions of *F* may use different functions and thus depends on different files, we have to create versions of the makefiles that manages *F* and the files *F* depends on. At the same time, since there are multiple versions for 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* is subject to changes when we revert to an earlier version. A safe bet is to create versions for all the files *F possibly* depends on. But this is usually an 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 between 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 [9] 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, then 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 each 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.

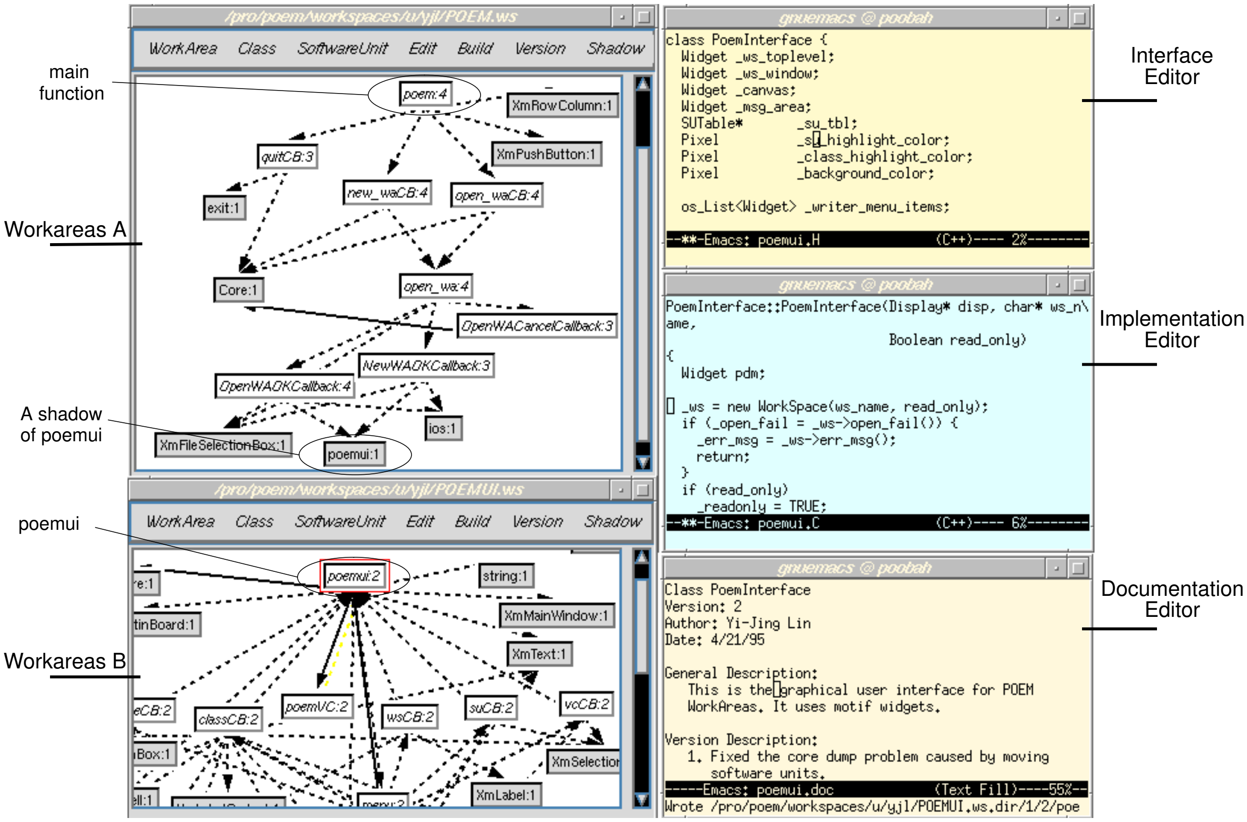
The situation is even worse when we are developing large programs that use multiple makefiles. Since makefiles lack formal communication channels like parameters and return values, the exchange of information between makefiles can only rely on implicit protocols. If we have a makefile *A* that uses the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

As we know, the major benefit of modularization 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.

3 Configuration Management in POEM

To solve the problems listed in the previous section, we are developing a programming environment called POEM (Programmable Object-centered EnvironMent) [16]. POEM manage system building and version control directly in terms of the functions and classes in the source code. It lets users interact with the logical structure of their software via a graphical user interface, and allows multiple programmers to develop software cooperatively over a local area network.

Figure 2 shows a typical screen of POEM. There are two *workareas* 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 them represent a function or a class. If we click mouse button on a software unit, then POEM will load 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 a textual description of the software unit. Programmers can put anything their think appropriate there.



```
class PoemInterface {
  Widget _ws_toplevel;
  Widget _ws_window;
  Widget _canvas;
  Widget _msg_area;
  SUTable*        _su_tbl;
  Pixel           _su_highlight_color;
  Pixel           _class_highlight_color;
  Pixel           _background_color;

  os_List<Widget> _writer_menu_items;

--**-Emacs: poemui.H          (C++)---- 2%--------
```

```
PoemInterface::PoemInterface(Display* disp, char* ws_n\
ame,
                             Boolean read_only)
{
  Widget pdm;

  _ws = new WorkSpace(ws_name, read_only);
  if (_open_fail = _ws->open_fail()) {
    _err_msg = _ws->err_msg();
    return;
  }
  if (read_only)
    _readonly = TRUE;
--**-Emacs: poemui.C          (C++)---- 6%--------
```

```
Class PoemInterface
Version: 2
Author: Yi-Jing Lin
Date: 4/21/95

General Description:
   This is the graphical user interface for POEM
   WorkAreas. It uses motif widgets.

Version Description:
   1. Fixed the core dump problem caused by moving
      software units.
-----Emacs: poemui.doc          (Text Fill)----55%--
Wrote /pro/poem/workspaces/u/yjl/POEMUI.ws.dir/1/2/poe
```

Figure 2: A typical screen of POEM

The dash-line arrows between software units are uses_interface links. If there is a uses_interface link from software unit *A* to software unit *B*, it means that the implementation of *A* depends on the interface of *B*. It is analogous to including a ".h" file in a ".c" file under traditional environments. The solid-line arrows between software units are t_uses_interface links. If there is a t_uses_interface links from software unit *A* to software unit *B*, it means that the interface of *A* depends on the interface of *B*. It is analogous to including a ".h" file in another ".h" file under 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 backend. 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.

3.1 System Building

The example in Figure 2 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 the workarea *A*. To build the executable code of POEM, we select this software unit by clicking mouse button on it, and choose *BuildExe* under the *Build* menu. POEM will compile all the software units whose object code is outdated, and link their object code into a executable program. To run the program, we can choose *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 workareas windows work like a graphical makefile. They 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 a big picture of the general structure of our program.

3.2 Version Control

If we want to keep the current status of a class or a function, we can select its corresponding software unit and choose *Snapshot* under the *Version* menu. Figure 3 shows the result of applying the snapshot operation on a software unit poemui. 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 link 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.

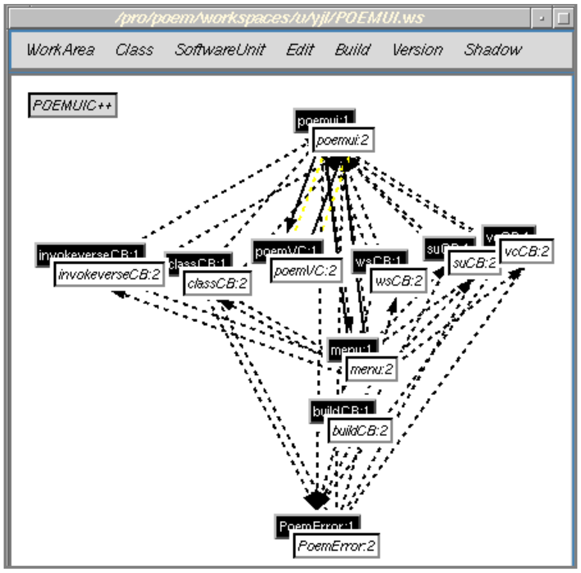


Figure 3: Making snapshot of a class

Fixed software units are read only, but they can be used in the same way as other software units are. If we click on a fixed software unit, POEM will bring 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 internally use RCS [24] 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. Operations on one version of a software unit will not interfere with other versions.

3.3 Cooperative Programming

The software units in one workarea may have links pointing to software units in other workareas. In the workareas in Figure 2, white boxes are locally defined software units, and grey boxes are actually the *shadows* of software units defined in other workareas. For example, the grey box labeled poemui:1 near the bottom of workarea *A* is a shadow of the earlier version of poemui:2, 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 that are not shown in Figure 2. But in spite of this complexity, it is represented as a single shadow box in its client, work-area *A*. When we are using a software unit from another workarea, we can ignore all its internal complexity.

In this example, workarea *B* is actually owned by another programmer working on a different machine. As a client, we cannot modify its contents, but we can use the software units it contains by pointing links to them. To reduce interference, programmers are recommended to use only the older but stable versions of software units developed by other programmers.

3.4 Software Reuse

In POEM, reusable libraries are represented as special workareas. Figure 4 shows the workarea for the Motif library. Each software unit in this workarea correspond to a Motif widget class or a Motif function. If we want to use one of them, we can simply establish a link to it. These software units behave just like normal ones. If we click mouse on a library software unit, POEM will load its header file into the interface editor, and its manual pages into the documentation editor.

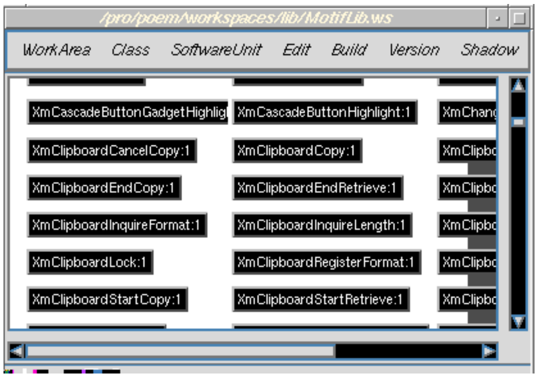


Figure 4: The workarea for Motif library

In existing environments, reusing a library function or class is more tedious. To reuse a library function or class, programmers have to locate its archive file and its header file, and then modify the makefiles to incorporate them. At the same time, programmers have 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.

An more important advantage of POEM is that we can directly reuse functions and classes that are not made into libraries. For example, if we want to reuse the class represented by the software unit poemui, which is defined in workarea *B* of Figure 2, we can simply establish links from our new software units to poemui. This is shown in Figure 5. Although poemui 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.

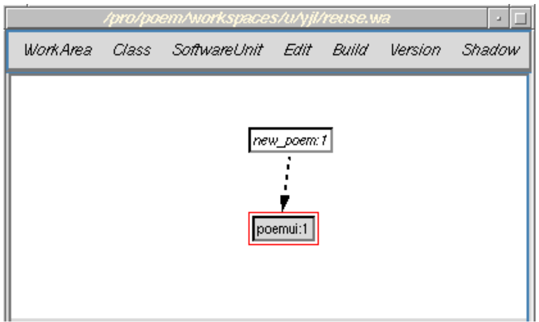


Figure 5: Reusing a class

Actually, POEM does not differentiate between a reused software unit and a locally created one. All software units in a project are treated the same. 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. The 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 library 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. Besides, modifying makefiles to incorporate a large module is usually difficult.

Another problem with existing environments is that modules can be reused easily only when they are made into libraries. Since making large modules into libraries is not trivial, programmers seldom do this with their code. As a result, many chances for software reuse are missed.

4 The Model

The model of POEM has four major concepts - *software units*, *subsystems*, *workareas* and *classes of software units*. In this section, we discuss these basic concepts and their roles in system building and version control.

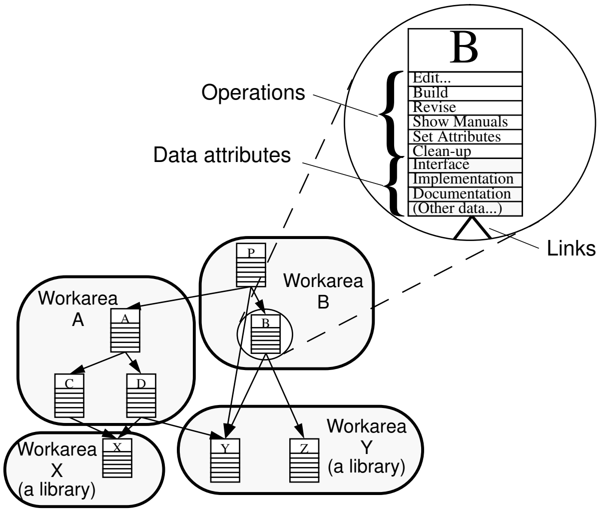


Figure 6: The POEM model

From a programmer's point of view, POEM consists of a set of software units that are interconnected by links. As illustrated in the inset of Figure 6, 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 they originate from, but can be modified directly by programmers. To support the basic functions of our framework, we only need two kinds of links - uses_interface links and t_uses_interface links.

A software unit usually represent a function or a class in C and C++. But if it is 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.

The *subsystem* designated by a software unit *X*, *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* will generate 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 6 the subsystem *S(A)* contains software units *A*, *C*, *D*, *X* and *Y*; the subsystem *S(B)* contains *B*, *Y*, and *Z*. *Y* appears in both subsystems.

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 a *owner* programmer. Only the owner can edit software units in a workarea. The ownership of a workarea, however, can be transferred between programmers.

In each workarea, there are 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 will edit active software units in one workarea, and access 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 [1][7][17][22]. 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. 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. In DSEE [14] and SHAPE [17], the sharing of derived objects is handled by more complicated mechanisms.

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 it is created from. Users can also create new classes of software units to meet their special needs.

To insure the compatibility between software units, we require that all new classes should be created as the subclasses of existing ones. A subclass can redefine the operations it inherits from its parent class, but cannot remove them. By doing so, we can insure that the operations defined in system-defined classes will be available in all classes.

4.1 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 between 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 the manipulation of derived objects. 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 simple 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 should invoke the build operation of *X*. If the building is successful, then the executable file will be returned as the result of the build operation. If the building fails, then the error messages will be associated with the software units that contain the erroneous code. Programmers can also build the derived objects 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 derived objects are outdated, propagates the build message along the links, and then collect derived objects from the subsystems it uses. As illustrated in Figure 7, the propagation of build messages is essentially a depth-first traversal of the graph formed by software units and their links.

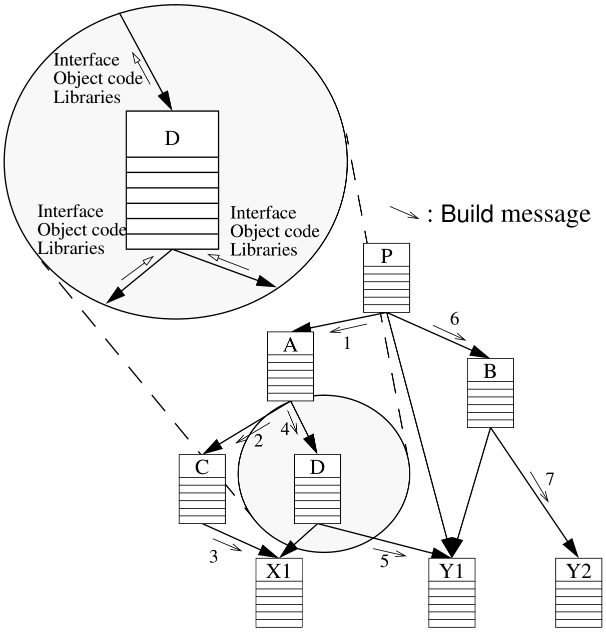


Figure 7: The system building process

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, derived objects are also passed upstream along the links. This flow of software artifacts is illustrated in the inset of Figure 7.

An important problem we have to deal with is the cycles formed by the links. Cycles of links are sometimes unavoidable when software units are mutually dependent. These cycles 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 on each software unit. These parameters include 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 operations are also propagated along links, so that the same setting will take effect on the whole subsystem. This propagation can be prohibited when programmers want to confine the effects of the new setting to the root software unit.

4.2 Version Control

Our version control system has two major goals. First, we want to provide facilities that allow programmers to create, select, and use versions in terms of subsystems. Second, we want to simplify the 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 vers*ion* 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. This is illustrated in Figure 8.

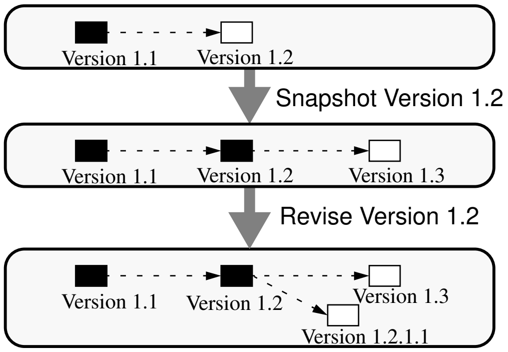


Figure 8: Version control of a single software unit

In our model, invoking the snapshot operation or revise operation on a software unit *X* will affect the whole subsystem *S(X)*. When the snapshot operation is invoked, our model generates fixed versions for all the modified software units in *S(X)*, and connect them together in the same way as in the original version. When the revise operation of a active software unit is invoked, our model will create 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 in remote workareas because remote workareas are usually owned by other programmers, and usually we do not want to directly modify the code owned by other programmers.

Figure 9 shows the effect of revise and snapshot operations on a subsystem. Notice that since we do not modify *C* and *X* in Figure 9(c), no new snapshots are created for them in Figure 9(d). However, although *D* is not directly modified either, a new snapshot of *D* is created because it uses a modified version of *Y*.

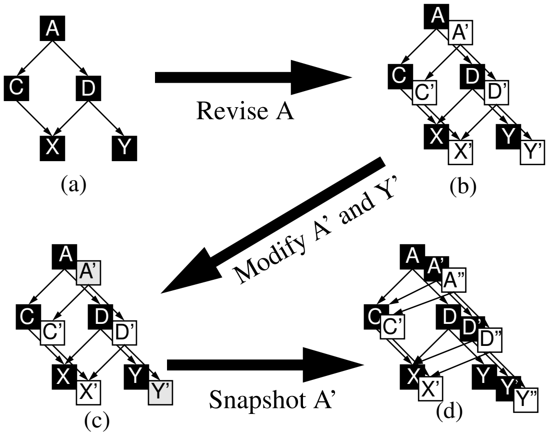


Figure 9: Version control of a subsystem

In our model, 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. By having this property, we can guarantee that given the same argument in build operations, *S(X)* will always generate the same derived objects.

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.

Our model to handle composite objects is similar to that of PCTE [10][4]. 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 derived objects 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 derived objects 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, the selection of versions is made 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)* contains, 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 get files that are not compatible, since two reliable files do not imply that their combination are also reliable. While this problem is less likely to happen in our framework, we have to insure that only one version of a software unit is used by a subsystem. For example, in Figure 9(b) we have to insure that *C'* and *D'* use the same version of *X*. 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.

Our framework also aims to simplify the access to old versions, while reducing the space they occupy. To save space, a software unit may delete its derived objects and check-in its source code in its snapshot operation. But we require that all these space conserving actions be hidden from the users, so that a fixed version can be accessed in the same way as an active version is. 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. In other words, 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 invoked the clean-up operations of its least-recently-used software units.

4.3 Discussion

Our framework draws its power from two principles. First, 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.

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 [19], 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 on these software artifacts, 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.

5 Implementation

As illustrated in Figure 10, the architecture of POEM is composed 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. Currently we are using an object-oriented database system called ObjectStore [13] to implement this layer. ObjectStore is basically an extension of the C++ language that supports persistent objects. It allows multiple clients to work on the same database simultaneously in a distributed environment, and it supports transaction facilities to serialize the operations on objects.

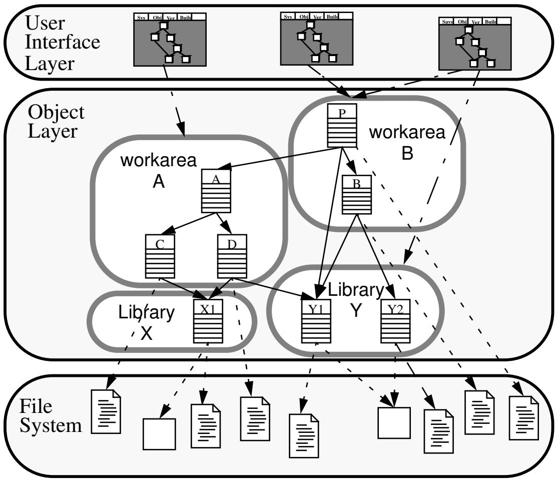


Figure 10: The architecture of POEM

To increase the flexibility of this environment, we supply an interpretive language that allows users to define new classes of software units, or change the definition of individual software units. Since most activities in POEM are carried out by the operations of software units, defining new classes of software units has the effect of tailoring the whole environment.

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 between 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.

The implementation of POEM involves about 8,000 lines of C++ code on top of Motif, ObjectStore, 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 possible because our model is rather simple, and because we utilize many existing tools.

6 Related Work

MAKE [9] is the most commonly used system building tool, but it has rather weak support for the modularization 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. The VESTA configuration management system [15] aims to solve these problems by the modularization and parameterization of system models. It uses a functional language to describe system models, and emphasizes the modularization and parameterization of system models. However, since the nature of system building requires lots of parameters for each system building functions, 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. Our framework shares the same goals with VESTA in system building, but uses an object-oriented approach instead of a functional approach. The parameters for system building are stored on software units instead of passed as arguments to functions.

Early version control systems like SCCS [20] and RCS [24] handle the versions of files only. CVS [3] enhances RCS by letting programmers handle versions of directories. DSEE [14] allows version selection in terms of *threads* that refers to files or other threads. ClearCase [1] enhances the functionality of DSEE and runs on several platforms without special supports from the underlying operating system.

POEM controls versions in terms of software units, which are 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 [25]. PCTE+ [4] supports operations to manage versions of composite objects. Katz surveyed version modeling in engineering databases, and proposed a unified framework [12].

Users of POEM do not have to manage derived objects by themselves. Several other systems also aim to free programmers from the management of derived objects. Cedar [23] and DSEE [14] use *source oriented system models*. Derived objects are not directly managed by programmers, but programmers still have to address them indirectly by some functions. VESTA hides the management of derived objects by passing them as the results of building functions.

Using POEM, programmers do not have to write makefiles. But most existing configuration management systems use separate documents to describe the system building process. CaseWare [6] stores the information about how to build a particular type of object on the object type definition itself, and places the build context information on objects. But the compositions of configurations are still described by separate collections called *assemblies*.

Our framework aims to narrow the gap between handling source code and managing configurations. In Ada [1], 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. Besides, an Ada environment is responsible to decide which units need to be recompiled based on the logical relations between them [5]. 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 [18] presented one of such mechanisms.

Gandalf [11] uses a module concept where a set of versions implement a single interface. Adele [8] extends this model so that interfaces themselves may have versions. It also describes the relation between interfaces and implementations as an AND/OR graph. Compared with our framework, Adele is more flexible to handle variants of versions, but it is also more complicated.

Rumbaugh [21] 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 suffice to meet our requirements.

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 allow programmers to handle large subsystems as a single unit.

While the basics of this framework is established, there are still many problems to be solved. We are developing a formal model to describe the mechanisms in our framework, and prove the properties we claimed. We are also working on mechanisms to manage variants of subsystems, so that programmers can switch between them more easily.

We also need more field test on our ideas. Until now, our experience with POEM has been more than satisfactory. But limited by the resources we have in a university, we have not use 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 will be different from those using workspaces in existing programming environments.

Acknowledgments

We would like to thank David Notkin, Peter Wegner, Stanley Zdonik, Hsueh-I Lu, and Anthony Cassandra for their suggestions and comments.

References

[1] Atria Software, Inc., "ClearCase Architecture and Database," Atria Software, Inc., Massachusetts, USA, August 1993.

[2] J. G. P. Barnes, "An Overview of Ada," in *Software Practice and Experience*, vol. 10, pp. 851-887, 1980.

[3] Brian Berliner, "CVS II: Parallelizing Software Development," Prisma, Inc., 5465 Mark Dabling Blvd, Colorado Springs, CO 80918.

[4] 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.

[5] 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.

[6] Martin R. Cagan, "Software Configuration Management Redefined," CaseWare, Inc., 108 Pacifica, Irvine, CA 92718, March 1992.

[7] W. Courington, "The Network Software Environment," Technical Report Sun FE197-0, Sun Microsystems Inc., February 1989.

[8] Jacky Estublier, "A Configuration Manager: The Adele data base of programs," in *Workshop on software engineering environments for programming-in-the-large*, June 1985

[9] Stuart I. Feldman, "Make - a Program for Maintaining Computer Programs," in *Software - Practice & Experience*, 9(4), pp. 255-265, April 1979.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] David B. Leblang and Robert P. Chase, Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," in *SIGPLAN Notices*, vol. 19, No. 5, pp. 104-113, April 1984.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Leon Osterweil, "Software Process are Software Too," in *Proceedings of the 9th International Conference on Software Engineering*, pp. 2-13, Monterey CA, March-April 1987.

[20] Marc J. Rochkind, "The Source Code Control System," in *IEEE Transactions on Software Engineering*, pp. 364-370, Vol 1 No 4, December 1975.

[21] James Rumbaugh, "Controlling Propagation of Operations using Attributes on Relations," *ACM OOPSLA '88 Proceedings*, pp. 285-296, September 1988.

[22] SunPro, "SPARCworks/TeamWare Solution Guide, Sun Microsystems, Inc., California, USA, 1994.

[23] " Warren Teitelman, "A tour through Cedar," in *IEEE Software*, pp. 44-73, Apr 1984.

[24] Walter F. Tichy, "RCS - A System for Version Control," in *Software - Practice & Experience,* Vol. 15, No. 7, pp. 637-654, July 1985.

[25] Stanley B. Zdonik, "Version Management in an Object-Oriented database," in *Advanced Programming Environments*, (R. Conradi, T. M. Didriksen, and D. H. Wanvik, ed.), pp. 405-422, 1986.