# Configuration Management in terms of Modules

Yi-Jing Lin and Steven P. Reiss

Department of Computer Science
Brown University
Providence, Rhode Island 02912

**CS-94-45**
May 1995

*Published in Proceedings of the 5th International Workshop on Software Configuration Management, April 1995, pp. 17-26*

# Configuration Management in terms of Modules *

Yi-Jing Lin and Steven P. Reiss
Department of Computer Science
Box 1910, Brown University
Providence, RI 02912
yjl@cs.brown.edu spr@cs.brown.edu

## Abstract

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* directly with them. Instead, system building and version control are usually handled with different decomposition structures. The modules used in configuration management do not always match the modules in the source code. This is both inconvenient and error-prone, since there is a gap between handling the source code and managing the configurations.

In this research we propose a framework for programming environments that handles configuration management directly in terms of the modules 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 with the ability to handle a large module as a single unit, software reuse and cooperative programming becomes easier. We also design and implement a prototype environment to verify our ideas.

## 1. Introduction

Modulization is one of the most important techniques used to harness the complexity of software. It is a common practice to decompose source programs into units such as functions, procedures, and classes that represent modules. However, existing programming environments do not handle configuration management directly in terms of these logical modules. In more traditional environments, versions are managed in terms of files and directories, which only roughly correspond to the modules in the source code. Some advanced environments replace the underlying file system with object-oriented databases, but usually the organization of objects in these environments do not match the structure of the source code either. Most existing environments also describe the process of system building with separate descriptions (e.g. makefiles). These descriptions decompose a software system in a separate structure that does not necessarily match the decomposition structure of the source code.

Managing configurations and source code with different structures is not only cumbersome but also error-prone. Maintaining a mapping between the logical structure of source code and the physical structure of software artifacts is an unnecessary burden on programmers. Whenever the source code is modified, programmers have to check if some corresponding modification is required at the configuration management level. When software systems become larger, this mapping between source code and configuration management may become very complicated.

Another problem with most current environments is that the configuration management tasks are not well decomposed into modules with simple interfaces. At the source code level, since a function or a class hides the details about the functions and classes it uses, using a high-level function or class is as easy as using a low-level one. However, this is usually not true in configuration management. If we want to use a large module, we have to handle the derived objects generated by its submodules. If a module contains many submodules and uses many files, then managing the modules's versions becomes difficult.

In this research, we present a new framework for programming environments that handles configuration management directly in terms of the functions and classes in the source code. Under this framework, users can create and use versions of a large module with simple operations, no matter how many submodules the module contains. Users can also describe the system building process by establishing the module-submodule relationships. We show that with the ability to handle a large module as a single unit, cooperative programming becomes easier, since programmers will be dealing with modules, instead of files, that are generated by other programmers.

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.

There are limitations to our approach, though. Since

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

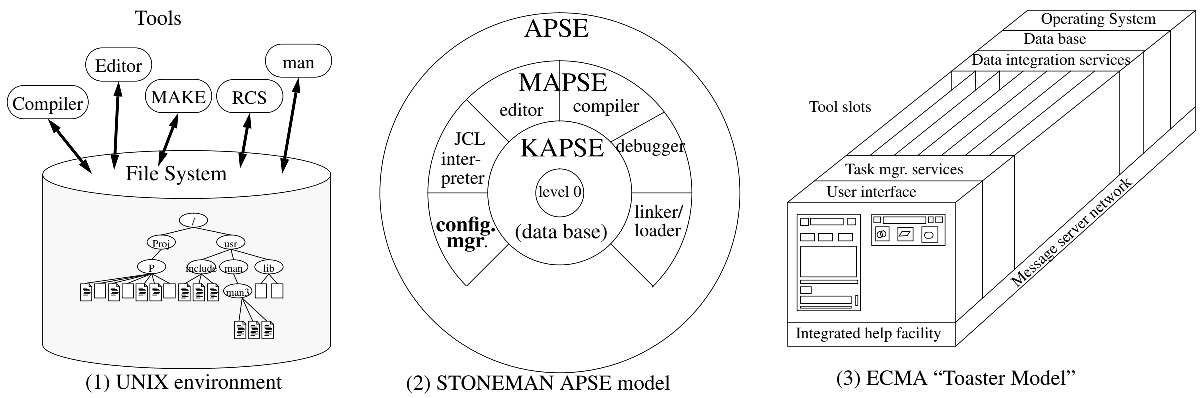



Figure 1: Current environments use separate tools to handle configuration management.

our framework is based on the modulization of programs, it cannot properly handle documents generated before the modulization. Our environment is designed to support configuration management in software design, implementation, and maintenance, but not for activities like requirement specifications and feasibility studies.

## 1.1. The Model

In current environments, configuration management is handled with tools that run *on top* of data repositories. For example, traditional UNIX programming environments manage configuration with tools like MAKE [Feldman 79] and RCS [Tichy 85] that operate on files and directories. STONEMAN [Buxton 80] envisioned configuration management in an Ada Programming Support Environment (APSE) as tools running on top of a database. In the ECMA [ECMA] "Toaster Model," low-level configuration management is supported by a database, but higher level activities are still carried out by separate tools.

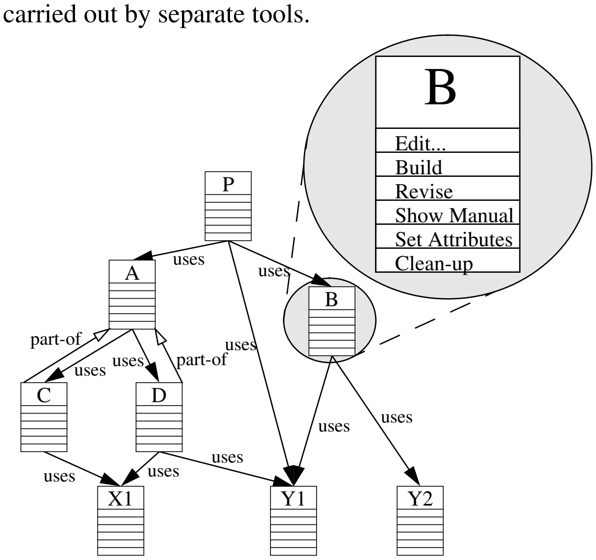



Figure 2: An object-centered programming environment

As illustrated in Figure 2, our framework represents functions and classes of a program as objects called *software units*, and represent the relations between them as links. 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. In this environment, programmers manage software by invoking the operations of software units, instead of using tools that are separated from the data repository.

We organize software artifacts at the *configuration management* level in parallel with the modulization 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 modulization and encapsulation in configuration management. It is well known that modulization and encapsulation is very useful in managing source programs. Similarly, they can help us greatly in handling configuration management. As pointed out by Osterweil [Osterweil 87], 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 modulization and encapsulation on these software artifacts, we can simplify configuration management greatly. Modulization helps us to decompose the configuration management tasks into manageable units. Encapsulation helps us to hide the different configuration management requirements of individual modules.

## 1.2. Software Unit, Module, and Work Area

From a user's point of view, our environment consists of a set of software units that are interconnected by links. Software units represent modules, and links represent the module-submodule relations. More specifically, if we are programming in C or C++, a software unit corresponds to a function or a class. A link corresponds to the relation traditionally represented by a "`#include`" directive. It may be the calling of a function, the usage of a class, or the inheritance between two classes.

As illustrated in Figure 3, 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.

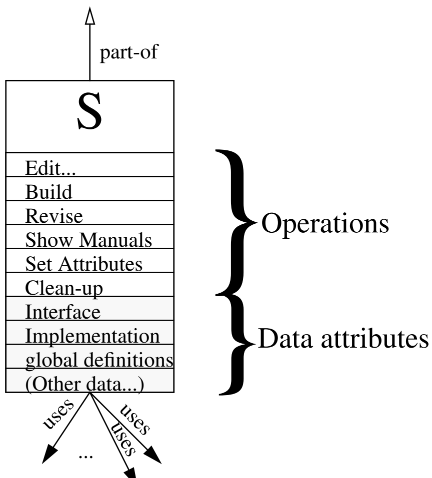



Figure 3: A software unit S.

Among the data attributes of a software unit, *interface*, *implementation*, and *global definitions* are source code that is filled in by programmers. The interface part specifies the facilities provided by a module. The implementation part realizes what is specified in the corresponding interface. The global definitions contain the definitions to be shared by submodules. If we are programming in C or C++, then the implementation part corresponds to a ".c file" under a traditional programming environment. The interface part corresponds to a public header file that declares the facilities defined in the ".c file." And the global definition parts correspond to a private header file that declares the common definitions to be shared by all ".c files" of a module.

There are two kinds of links in our framework - *uses* links and *part-of* links. Each software unit has a set of `uses` links pointing to the software units representing its submodules, and an optional `part-of` link pointing to the software unit representing its parent module. These links define the set of definitions to be imported from other modules. A `uses` link imports the interface from a submodule, and a `part-of` link imports the global definition from the parent module.

`Part-of` links are the inverse links of `uses` links, but they are required only when software units need the global definitions of their parents. Use of `part-of` links should be avoided whenever possible, because they increase the interdependence between modules. As we will see later, software units without `part-of` links have nice properties for software reuse and cooperative programming.

In our framework, a *module M(X)* is defined as a *root software unit X* and all the software units that are reachable from *X* by `uses` links. A module can be handled 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 module *M(X)*, not just those of the root software unit *X*.

During the design and implementation stages, software units work like templates. Programmers create new software units and then fill in their interface, implementation and global definitions by invoking the `edit` operation. The `edit` operation brings up editors for this purpose. `Uses` links and `part-of` links can be specified either before or after the contents of software units are filled in. Therefore, programmers may layout the general structures of their programs before doing any coding.

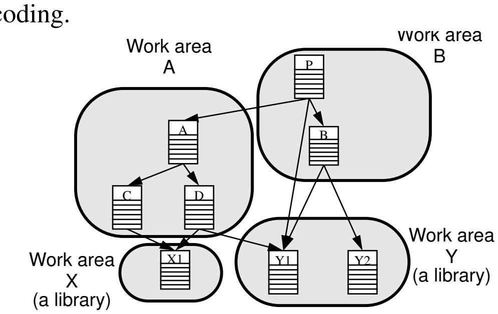



Figure 4: Software units are grouped into work areas

As illustrated in Figure 4, software units are grouped into mutually exclusive *work areas*. Work areas 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 work area contains some mutable software units that are under development, and some immutable ones that represent old versions. The immutable software units can be accessed by all programmers, but the mutable ones are visible only to the *owner* of a work area.

To reduce interference, only one programmer can be the owner of a work area at a time. Usually, a programmer will edit mutable software units in one work area, and at the same time access immutable software units in other work areas. Immutable software units in remote work areas can be accessed directly without check-in or check-out operations.

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, and some software units may be library functions that do not have implementation parts. We use the *delegation model* [Lieberman 86][Ungar 87][Stein 87][Orlando 89] instead of the more conventional *class/inheritance model* to describe the sharing of behavior between software units. Delegation can simulate the class/inheritance mechanism while supporting the sharing of attributes between individual objects.

With the delegation mechanism, we define a set of *prototypes* for commonly used software units, like those for C, C++, and YACC. These prototypes work like *classes* in a conventional object-oriented system. All software units are created from these prototypes, and a software unit inherits the operations and default attribute values from its prototype. But with delegation, we can also add new attributes and redefine operations on individual software units to meet special requirements.

Our framework requires no modification to the syntax of C and C++. But since the `uses` links and `part-of` 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.

# 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 `uses` links and `part-of` links, 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 module without knowing what derived objects it generates. Lastly, we want to let programmers customize the system building process in terms of modules. For example, programmers should be able to turn on the debugging flag of a module 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 modules 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 module. Each software unit compiles its source code if its derived objects are outdated, propagates the `build` message along the `uses` links, and then collect derived objects from its submodules. As illustrated in Figure 5, the propagation of `build` messages is essentially a depth-first traversal of the graph formed by software units and their `uses` links.

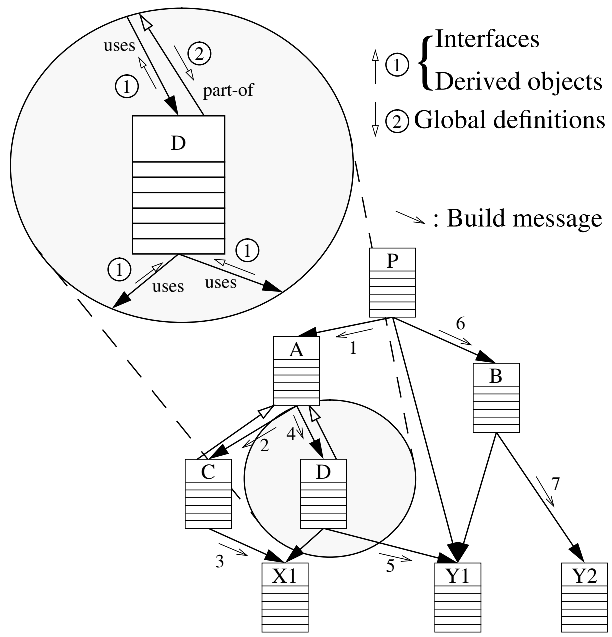



Figure 5: Propagation of the `build` operation.

During the system building process, software artifacts are passed along the `uses` links and `part-of` links. When compiling, a software unit gets the interfaces of its submodules along its `uses` links, and gets the global definitions of its parent module along its `part-of` link. In the meantime, derived objects are passed upstream along the `uses` links. This flow of software artifacts is illustrated in the inset of Figure 5.

An important problem we have to deal with is the cycles formed by the `uses` links. Cycles of `uses` links are sometimes unavoidable when modules 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 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 the `uses` links, so that the same setting will take effect on the whole module. This propagation can be prohibited when programmers want to confine the effects of the new setting to the root software unit.

# 3. Version Control

Our version control system also has three major goals. First, we want to provide facilities that allow programmers to create, select, and use versions in terms of modules. Second, we want to simplify the access to old versions while reducing the space they occupy. Finally, we want to handle the sharing of derived objects automatically.

In terms of version control, we classify software units into *fixed versions* and *active versions*. A fixed version represents an immutable snapshot of a module. An active version represents a working copy of a module. An active version provides a *freeze* operation that turns itself into a fixed version. 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. Then by repeatedly applying `freeze` and `revise` operations, we can create a version history tree for each software unit.

Since a fixed software unit represents an immutable snapshot of a module, it should always produce the same derived objects during the system building process. To insure this property, a fixed software unit should freeze all the software units it uses. If a fixed software unit has a `part-of` link, then the parent software unit should also be frozen.

Similarly, since an active software unit represents a developing module, it should use active software units in the local work area. Invoking the `revise` operation of a fixed software unit will create a module that contains active software units in the local work area and fixed software units in remote work areas. The `revise` operation does not create active versions in remote work areas because active software units cannot be accessed across the work area boundaries, and because it is seldom what programmers want.

Figure 6 shows the effect of `revise` and `freeze` operations on a module. Notice that since we do not modify *C* and *X* in Figure 6(c), they are *reverted* to their previous versions in Figure 6(d). However, although *D* is not directly modified either, a new version of *D* is created because it uses a modified version of *Y*.

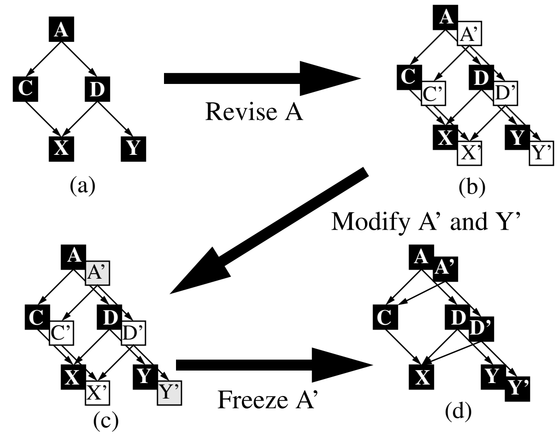


Figure 6: Version control of a module

Like the system building process, the revising and freezing of a module is carried out by the collaboration of individual software units. When revising a module, the `revise` message is propagated along the `uses` link until it hits the boundaries of a work area. Each software unit then uses its own `revise` operation to process its data attributes. The freezing of a module is carried out similarly by propagating the `freeze` message. Basically, a `freeze` operation should reduce the space occupied by a software unit. A `revise` operation should make modifiable copies of its data attributes.

Our model to handle composite objects is similar to that of PCTE [Gallo 86][Boudier 88]. 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 `freeze` operations. This is necessary because different kinds of software units may have different data attributes.

Our framework allow users to choose versions directly in terms of modules. If programmers want to use a certain version of module *M(X)*, they can simply point a `uses` link to the corresponding version of the root software unit *X*. Programmers do not have to know which software units *M(X)* uses, or whether different versions of *M(X)* use a different set of software units.

Like DSEE [Leblang 84], SHAPE [Mahler 88], and Adele [Estublier 85], we use *version selection rules* to make the selection of versions more flexible. But since versions are selected by the `uses` links in our framework, we associate these rules with `uses` links instead of putting them in a separate description. A version selection rule contains a predicate made in terms of the attributes of software units. The effective destination of a `uses` link with a version selection is the latest version of a software unit that satisfies this predicate. However, since the `uses` links with version selection predicates do not have fixed destinations, they cannot be used by fixed software units. A `freeze` operation should freeze them into normal `uses` links.

Compared with DSEE and SHAPE, version selection in our framework is more hierarchical. The selection of versions is in terms of modules instead of files, and each software unit hides the selections of its submodules from its parent software unit. In DSEE and SHAPE, 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 module. For example, in Figure 6(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.

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 `freeze` 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 bringing up an editor. We also require that the `freeze` operation not change the semantics of a module. In other words, a fixed module 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 should reduce the space occupied by a module without affecting its behavior. Again, we propagate the `clean-up` messages along the `uses` 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 work area is short of disk space, it may invoked the `clean-up` operations of its least-recently-used software units.

We also use the delegation mechanism to save space in version control. The basic idea of delegation is that if an attribute is not defined on an object, the underlying system will try to find that attribute on the *prototype* of the current object, and use that value as if it is locally defined. In our environment, a new version internally uses its predecessor version as its prototype, and stores only the attributes that are modified on the new version.

Delegation, in this respect, is similar to version control systems like RCS [Tichy 85] and SCCS [Rochkind 75], where only the deltas between versions are stored. But using delegation has an additional benefit - versions can be accessed directly without check-in and check-out operations. Internally, attributes are shared between versions. But to users, it looks as if every version keeps its own copy of all the attributes.

# 4. Software Reuse

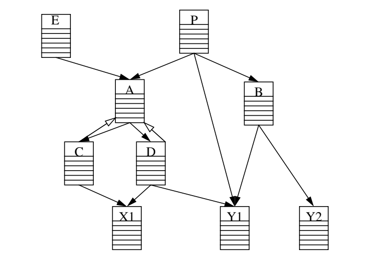
The ability to handle a large module as a single unit simplifies the process of incorporating reusable modules into new projects. It makes the reuse of library modules easier. It also helps in reusing modules that are not made into libraries.

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, 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.

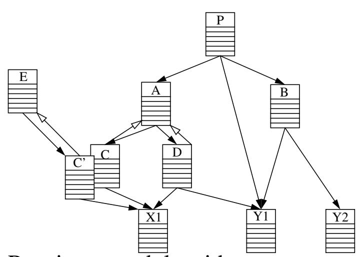
In our framework, we represent library modules as software units. These software units have different internal structures but bear the same interface as other software units. To reuse one such software unit, programmers can simply establish a `uses` link to it. With this single action, its header file,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

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.

In our framework, modules can be reused directly without being made into libraries. Because we designate each modules by a single root software unit regardless of its internal complexity, a large module can be used in the same way as a library module is used. As illustrated in Figure 7(a), if a module does not have a `part-of` link, we can reuse it by pointing a `uses` link to it. With this single operation, the reuser gets not only the derived objects of the reused module, but also its source code, manual pages, version history, and the operations to handle it. In contrast, if we want to reuse a module that are not made into libraries in a traditional environment, 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.




(a) Reusing a module without a `part-of` link.




(b) Reusing a module with a `part-of` link.

Figure 7: Reusing modules

Modules with `part-of` links represent modules that need information from their parent. To reuse one of them, we should create an active version of the reused software unit and make its `part-of` link point to the new parent software unit. The reuser software unit is responsible for providing appropriate global definitions to the reused one. This is illustrated in Figure 7(b). Although different derived objects will be generated internally when this kind of module is reused, no other action is required. Since new derived objects will be kept as attributes of the new active version instead of overwritten to the existing version, the reuse will not interfere the original project. In contrast, to reuse this types of modules in a traditional environment, we have to copy all the source files of the module, or create a separate directory for the new derived objects and modify the makefile in a non-trivial way.

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 the `uses` 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.

# 5. Cooperative Programming

The major goal of our support for cooperative programming is to let programmers deal directly with modules, instead of files or individual objects, that are generated by other programmers. Our framework also enables us to reduce the interference between programmers, and increase the sharing of software artifacts more naturally.

Our scenario for cooperative programming follows the general strategy of DSEE [Leblang 84] and SHAPE [Mahler 88]. 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 *modules* generated by other programmers.

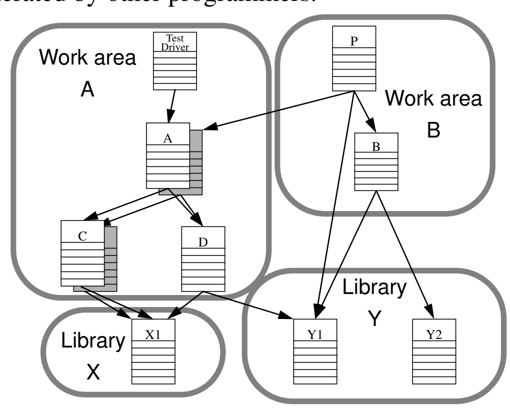



Figure 8: Cooperative programming.

Figure 8 shows a simple example of cooperative programming. Suppose that a program *P* is cooperatively developed by two programmers *A* and *B*. Programmer *A* works on module *M(A)*. Programmer *B* works on the main program *P* and module *M(B)*. In order to use *M(A)*, programmer *B* points a `uses` link from software unit *P* to software unit *A*. With this single `uses` link, the whole module *M(A)* is incorporated into the program. Programmer *B* does not have to know which software units *M(A)* contains, or what derived object it will generate. All the internal complexity of *M(A)* is hidden from programmer *B*. This is especially helpful when *M(A)* is very large, or when *M(A)* itself uses modules from other programmers.

Work areas are used to reduce the interference between programmers. Since active software units in a work area are visible only to the work area owner, the on-going work will not interfere or be interfered by other programmers. On the other hand, fixed software units of a work area are accessible to all programmers. 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 between programmers naturally. If two programmers use the same version of a module, then they will automatically shared all the source objects and derived objects of that module. For example, in Figure 8 since the same version of module *M(D)* is used by both programmer *A* and programmer *B*, all its data attributes will be shared. Moreover, because of the delegation mechanism described in section 3, programmers *A* and *B* will also share the attributes of software unit *A* and *C* that are not modified in the new version. In DSEE and SHAPE, the sharing of derived objects are handled by more complicated mechanisms.

# 6. POEM - A Prototype Environment

To verify the ideas of our approach, we are developing a prototype environment called POEM (Programmable Object-centered EnvironMent). As illustrated in Figure 9, the architecture of POEM is composed of a *user interface layer* and an *object layer* on top of the file system.

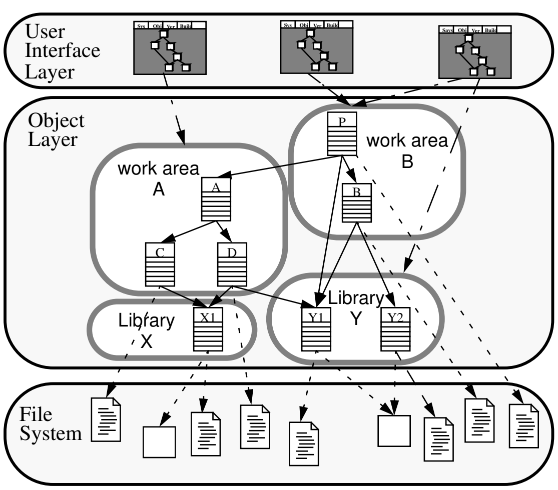


Figure 9: The architecture of POEM

The object layer is where the software units are stored. It is basically an object-oriented database. Currently we are using an object-oriented database system called ObjectStore [Lamb 91] 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.

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. Therefore, users can tailor the environment to their needs. The compatibility between 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 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.

# 7. Related Work

MAKE [Feldman 79] is the most commonly used system building tool, but it 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. The VESTA configuration management system [Levin 93] aims to solve these problems by the modulization and parameterization 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 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 [Rochkind 75] and RCS [Tichy 85] handle the versions of files only. CVS [Berliner] enhances RCS by letting programmers handle versions of directories. DSEE [Leblang 84] allows version selection in terms of *threads* that refers to files or other threads. 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 [Zdonik 86]. PCTE+ [Boudier 88] supports operations to manage versions of composite objects. Katz surveyed version modeling in engineering databases, and proposed a unified framework [Katz 90].

Several other systems also aim to free programmers from the management of derived objects. Cedar [Teitelman 84] and DSEE [Leblang 84] 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.

Most existing configuration management systems use separate document to describe the system building process. CaseWare [Cagan 92] 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*.

In Ada [Barnes 80], 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 re-compiled based on the logical relations between them. 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 [Morgan 88] presented one of such mechanisms.

Gandalf [Habermann 86] uses a module concept where a set of versions implement a single interface. Adele [Estublier 85] 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 [Rumbaugh 88] 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 work areas to control the propagation is suffice to meet our requirements.

# 8. 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 modules as a single unit.

While the basics of this framework is established, there are still many problems to be solved. Our framework does not have enough support for managing variants of modules at this moment. We plan to enhance this part by exploiting the delegation mechanism, so that module variants with many overlapping parts can be created without actually duplicating the data.

# Acknowledgments

We want to thank Clarence Clark at IBM for giving us numerous ideas and suggestions. Thanks also to Anthony Cassandra and Hsueh-I Lu for giving us suggestions on writing this paper.

# References

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

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

[Boudier 88] 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.

[Buxton 80] 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.

[Cagan 92] Martin R. Cagan, “Software Configuration Management Redefined,” CaseWare, Inc., 108 Pacifica, Irvine, CA 92718, March 1992.

[ECMA] 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.

[Estublier 85] Jacky Estublier, “A Configuration Manager: The Adele data base of programs,” in *Workshop on software engineering environments for programming-in-the-large*, June 1985

[Feldman 79] Stuart I. Feldman, “Make - a Program for Maintaining Computer Programs,” in *Software - Practice & Experience*, 9(4), pp. 255-265, April 1979.

[Gallo 86] 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.

[Habermann 86] 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.

[Katz 90] Randay H. Katz, “Toward a Unified Framework for Version Modeling in Engineering Databases,” *ACM Computing Surveys*, Vol 22, No 4, pp. 375-408, December 1990.

[Lamb 91] 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.

[Leblang 84] 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.

[Levin 93] 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.

[Lieberman 86] Henry Lieberman, “Using Prototypical Objects to Implement Shared Behavior in Object Oriented Languages,” in *ACM OOPSLA ‘86*, pp. 214-223, September, 1986.

[Mahler 88] 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.

[Morgan 88] 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.

[Orlando 89] Lynn Andrea Stein, Henry Lieberman, and David Ungar, “A Shared View of Sharing: The Treaty of Orlando,” In *Concepts, Applications and Databases*, pp. 31-48, Addison Wesley, Reading, Massachusetts, 1989.

[Osterweil 87] 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.

[Rochkind 75] Marc J. Rochkind, “The Source Code Control System,” in *IEEE Transactions on Software Engineering*, pp. 364-370, Vol 1 No 4, December 1975.

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

[Stein 87] Lynn Andrea Stein, “Delegation is Inheritance,” in *ACM OOPSLA ‘87*, pp. 138-146, October 1987.

[Teitelman 84] Warren Teitelman, “A tour through Cedar,” in *IEEE Software*, pp. 44-73, Apr 1984.

[Simmonds 89] 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.

[Tichy 85] Walter F. Tichy, “RCS - A System for Version Control,” in *Software - Practice & Experience,* Vol. 15, No. 7, pp. 637-654, July 1985.

[Ungar 87] David Ungar and Randall B. Smith, “Self: The Power of Simplicity,” in *ACM OOPSLA ‘87*, October 1987.

[Zdonik 86] 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.