

SOFTWARE VERSION MANAGEMENT SYSTEM

BACKGROUND OF THE INVENTION

This invention relates to software version management system and method for handling and maintaining software, e.g. software updating uniformly across the system, particularly in a large software development environment having a group of users or programmers. The system is also referred to as the "System Modeller".

Programs consisting of a large number of modules need to be managed. When the number of modules making up a software environment and system exceeds some small, manageable set, a programmer cannot be sure that every new version of each module in his program will be handled correctly. After each version is created, it must be compiled and loaded. In a distributed computing environment, files containing the source text of a module can be stored in many places in a distributed system. The programmer may have to save it somewhere so others may use it. Without some automatic tool to help, the programmer cannot be sure that versions of software being transferred to another user or programmer are the versions intended to be used.

A programmer unfamiliar with the composition of the program is more likely to make mistakes when a simple change is made. Giving this new programmer a list of the files involved is not sufficient, since he needs to know where they are stored and which versions are needed. A tool to verify a list of files, locations and correct versions would help to allow the program to be built correctly and accurately. A program can be so large that simply verifying a description is not sufficient, since the description of the program is so large that it is impractical to maintain it by hand.

The confusion of a single programmer becomes much worse, and the cost of mistakes much higher, when many programmers collaborate on a software project. In multi-person projects, changes to one part of a software system can have far-reaching effects. There is often confusion about the number of modules affected and how to rebuild affected pieces. For example, user-visible changes to heavily-used parts of an operating system are made very seldom and only at great cost, since other programs that depend on the old version of the operating system have to be changed to use the newer version. To change these programs, the "correct" versions of each have to be found, each has to be modified, tested, and the new versions installed with the new operating system. Changes of this type often have to be made quickly because the new system may be useless until all components have been converted. Members or users of large software projects are unlikely to make such changes without some automatic support.

The software management problems faced by a programmer when he is developing software are made worse by the size of the software, the number of references to modules that must agree in version, and the need for explicit file movement between computers. For example, a programming environment and system used at the Palo Alto Research Center of Xerox Corporation at Palo Alto, Calif., called "Cedar" now has approximately 447,000 lines of Cedar code, and approximately 2000 source and 2000 object files. Almost all binary or object files refer to other binary or object files by explicit version stamp. A program will not run until all references to an binary or object file refer to the

same version of that file. Cedar is too large to store all Cedar software on the file system of each programmer's machine, so each Cedar programmer has to explicitly retrieve the versions he needs to run his system from remote storage facilities or file servers.

Thus, the problem falls in the realm of "Programming-the-Large" wherein the unit of discourses the software module, instead of "Programming-in-the-Small", where units include scalar variables, statements, expressions and the like. See the Article of Frank DeRemer and H. Kron, "Programming-in-the-Large versus Programming in the small", *IEEE Transactions on Software Engineering*, Vol. 2(2), pp. 80-86, June 1976.

To provide solutions solving these problems over-viewed above, consider the following:

1. Languages are provided in which the user can describe his system.

2. Tools are provided for the individual programmer that automate management of versions of his programs. These tools are used to acquire the desired versions of files, automatically recompile and load a program, save new versions of software for others to use, and provide useful information for other program analysis tools such as cross-reference programs.

3. In a large programming project, software is grouped together as a release when the versions are all compatible and the programs in the release run correctly. The languages and tools for the individual programmer are extended to include information about cross-package dependencies. The release process is designed so production of release does not lower the productivity of programmers while the release is occurring.

To accomplish the foregoing, one must identify the kinds of information that must be maintained to describe the software systems being developed. The information needed can be broken down into three categories:

1. File Information: For each version of a system, the versions of each file in the system must be specified. There must be a way of locating a copy of each version in a distributed environment. Because the software is always changing, the file information must be changeable to reflect new versions as they are created.

2. Compilation Information: All files needed to compile the system must be identified. It must be possible to compute which files need to be translated or compiled or loaded and which are already in machine runnable format. This is called "Dependency Analysis." The compilation information must also include other parameters of compilation such as compiler switches or flags that affect the operation of the compiler when it is run.

3. Interface Information: In languages that require explicit delineation of interconnections between modules (e.g. Mesa, Ada), there must be means to express these interconnections.

There has been little research in version control and automatic software management. Of that, almost none has built on other research in the field. Despite good reasons for it, e.g. the many differences between program environments, and the fact that programming environments usually emphasize one or two programming languages, so the management systems available are often closely related to those programming languages, this fact reinforces the singularity of this research. The following is brief review of previous work in this area.

(1) Make Program