Software Design Revision Control
or,
How to Keep Too Many Cooks from Spoiling the Broth

Robert Babb II
Dick Hamlet

Oregon Graduate Center
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 88-004

December, 1988

Software Design Revision Control or, How to Keep Too Many Cooks from Spoiling the Broth

Robert Babb II
Dick Hamlet

Department of Computer Science and Engineering
Oregon Graduate Center
19600 NW Von Neumann Drive
Beaverton, OR 97006 USA
(503) 645-1121

Abstract — **Existing revision control systems deal mainly with the problem of recovering previous versions of source code segments. After a series of changes to program source code have been made, program maintenance tools such as "make" can be used to restore system consistency, but provide little guidance to the programmer about the impact and scope of proposed modifications to a system design. This paper describes a revision control system that will extend these ideas to assist programmers in coordinating a wider range of system design data. In addition to the usual tasks of coordinating versions of program source code and related object code, the system will manage dependencies involving specifications, interface and interconnection definitions, and test data. Design dependency information will be used to warn programmers about the impact of proposed updates to the system design, as well as to prevent the problems that can occur when two programmers attempt to change related pieces of design data simultaneously.**

Index Terms — **Revision Control Systems, Program Maintenance, Software Development Process, Source Code Control, Unit Testing, Integration Testing.**

Dr. Hamlet's work was partially supported by the Air Force Office of Scientific Research (Contract F49620-83-K-0018).

1. INTRODUCTION

It is possible to develop large software systems only because the systems are subdivided into relatively independent program modules. This division comprises the art of modular design, and has two distinct purposes. First, modular design allows many people to work in parallel on software development. This ability to bring many programmers to bear is most important during the coding phase, but it can also be important during preliminary design of very large systems, and in maintenance. Second, intellectual control of software depends on limiting its complexity, and modularization can help with this also. The analysis that division into modules makes possible (of their separate specifications, interfaces, unit tests, etc.) is important throughout the lifetime of a software product.

A software design is one of the most volatile products of the development process, because it is made in ignorance of the difficulties that will arise during the coding of the modules it calls for. As the coding proceeds, problems that arise in one place frequently require changes affecting a great many other places. Similarly, during maintenance, changes can propagate from one module to another, modifying the design even though the official design documents may not be updated to reflect them. Sets of test data must also be kept up to date with respect to changes in the modules that they are supposed to exercise.

Revision control systems such as SCCS[1] and more recently RCS[2] [3] are software tools intended to help keep design changes under control. They are used most often to control source code versions but can also manage a variety of other program development-related documents, such as code skeletons, stubs, and program design language modules,

Revision control can be combined with automated update programs that compare "latest modification times" for a set of inter-related files and perform compilation or other actions to keep the set of files consistent. The UNIX† "make" program is an example[4]. Such systems attempt to prevent the construction of systems whose integrity has been compromised by incomplete changes to program source files. In conjunction with proper languages and linkers (not necessarily used) and if not subverted by a programmer who "knows" that a particular file need not be regenerated, it can guarantee that a linked system will incorporate the latest versions of all program source files.

However, the automating of the module integration process is not the most important function of revision control. Changes are made by people, perhaps many people working together, and it is better to aid them at all steps of program design, coding and testing, rather than merely to attempt to repair design damage periodically. The aid we envision would usually take the form of enhanced communication between different

† UNIX is a trademark of Bell Laboratories.

programmers, but should also be of considerable use to an individual working alone on a program.

informed about the global implications of design changes as they are proposed, instead of diagnosing after the fact those that disrupt an evolving system design. The extra information will also help extend the design checking that can be done from syntax toward semantics.

2. SYSTEMS AND COMPONENTS

A revision control system is *not* an integrated software development environment, but only a relatively minor piece of such an environment. Its focus is on source code, and on other surrounding phases of development that support coding (design, testing), insofar as they are needed for changes. Under its control are units (modules) that can be assembled into a running software system. Programmers should be able to work on any unit in safety, knowing that a modified test system can be constructed without losing the older "current best" system by accident. Furthermore, programmers should know that there will be no unexpected problems caused by multiple simultaneous updates of the same or distinct source components by different people. And during the modification process, the system should provide global awareness of the scope and impact of any proposed change, taking advantage of all available machine-encoded system design information.

2.1. Components of a System under Development

In support of the stored source code, the following are examples of useful components in the design control system we envision:

Module Specifications. Although natural-language specification is the rule in large systems, it will probably not be very useful as part of automated change support. Abstract data type modules can be described by axioms in a way that allows testing, and assertional specifications can also be checked by tests. Therefore it will be worthwhile to experiment with the inclusion of such information, but this is the least well understood part of the proposed system.

Module Interface and Interconnection Definitions. Managing the data aspects of interaction between modules involves, for example, ensuring type agreement on communication paths (parameters for subroutines, message channels for concurrent processes). Some of the control aspects of the system design can also be captured by explicitly representing potential use relationships among modules (a software "wirelist"). These interface and interconnection definitions can be kept up to date automatically as part of the editing of encoded design specifications.

Concurrency constraints. It is a special feature of systems intended for parallel execution that their design should characterize what is allowable and what is erroneous run-time concurrent behavior. This information can be expressed in terms that may be verified — for example as grammatical descriptions of permitted and forbidden execution traces for sets of test points.

Unit- and Integration-test data. Interface definitions are static, depending only on the program structure expressed in its syntax. To make use of any other information requires the program semantics. Test data is the most practical form stored semantic information could take. Unit tests can be linked syntactically with specific data interfaces, and semantically with module specifications. Integration tests are judged against the system's specification as a whole. In addition, tests may be required to meet coverage criteria of some kind.

2.2. Properties to be Preserved by Revisions

Maximum help is provided to a programmer seeking to make a change if potential effects are displayed explicitly by the system. This is quite different from later checking for potential flaws caused by changes. For example, if a programmer were notified that his/her proposed change would affect almost every interface in the system, and would invalidate almost all unit test data thereby, he/she might think twice about making it.

Any change begins with the editing of a component stored under the design revision system's control. The potential implications of the change can be calculated from the syntactic relationships among components:

> *Changing a specification* implies that the specified module will be changed, potentially its interfaces, and potentially other modules that use it. If test data has been specified, it must also be changed to remain consistent with specification changes.
>
> *Changing an interface* may require changes to modules (both those that define it, and those that make use of it), and changes to specifications and tests using that interface.
>
> *Changing code* may require changes in all the other stored information supporting that code.
>
> *Changing concurrency constraints* may require changes to the module communication structure or may involve only program specification changes.
>
> *Changing tests* may affect all other stored information.

In each of these cases, the potential implications are usually more extensive than the actual ones. The programmer who intends to edit a component can be asked to predict the extent of the change, and when the actual change is available, it can be checked against this intention. (There can be a difficulty with simultaneous update, should expectation and reality prove too different — see below.)

In the traditional "batch" mode for system revisions, when a change is made its implications for other components are not investigated. Later, when all editing is completed, a batch of such changes are incorporated, with some syntactic diagnosis for system integrity. In the proposed "interactive" design mode, the checks would be applied immediatedly as each change is incorporated. The latter is a possibility because the syntactic sites of far-reaching changes are easy to isolate due to incorporation of the modular design structure information into the revision control system. For example, changes to an abstract data type module can propagate only if they occur in the exported portions of the code, and then only to modules that have data links to the module in question.

In response to actual changes, the system can prompt for other associated changes necessary to maintain the type consistency of specification, interface, code, and test data. When the syntax of changes is consistent, the system can execute using appropriate stored test data to determine partial consistency of specification and code, and partial satisfaction of concurrency constraints.

2.3. The Modification Process

The features described above are all intended to help the individual programmer, not to control multiple simultaneous access to system components. A design revision control system must also provide facilities to help people work together. The mechanism for this is a "check-out/check-in" scheme based on potential propagation of changes.

When a programmer intends to edit a component he/she is asked to specify the likely extent of the change, from the potential sphere of influence calculated by the system. All components specified are then checked-out to her/him, in a "sheltered workspace" so that the existing system is not compromised. Others may not specify for check-out any components already checked out. As changes are made, the check-out list might need to grow when an unexpected interaction occurs. Should the needed component then be checked out elsewhere, a potential for deadlock exists, and the system should simply so inform the people involved. In any case, addition of a new component to the checked out list implies that the newly checked out source be examined closely for recent changes, since its history is not known.

The system can calculate, for the changes made by one person, all secondary information that it possessed for the unmodified system. This information includes test results, the module interconnection pattern, and parallel execution sequences realized in testing. Unlike syntactic consistency, which can profitably be checked interactively, this secondary information is semantic, dependent on the existence of a complete, executable system, and so easily checked only when changes are complete. Test results must meet specifications and concurrency constraints must be observed. There is no necessary compromise of system integrity if other secondary information changes, but the programmer should be informed of such changes.

When a change is complete, all checked out components are again checked-in and the sheltered workspace cleared. When several people are working together, it can happen that two syntactically independent changes (hence ones in which no conflicts arose in check-out) may nevertheless conflict semantically at the time the second person tries to check-in. For example, the concurrency constraints may be violated because two apparently unrelated modules were changed by different people, the first checked in change may be really to blame, but the problem does not appear until the second check-in. Such a situation is obviously nasty, but the only solution seems to lie in providing the complete history of modification since check-out to the person who has semantic integrity problems on check-in.

It should be noted that nothing in the description of "changes" precludes application to that drastic change from design to coded system that is called "initial development." Test results cannot be used until fairly late in the development process, but multiple update control and syntactic consistency checks will be useful.

As a specific example of some of the ideas presented above, in the next section we describe briefly experience with using an existing revision control system to support a simple, highly constrained model for software structure. This model is especially relevant in this context because it is a unified software specification/design/implementation method, allowing examination of the "broader spectrum" aspects of revision control referred to above.

3. USING AN EXISTING REVISION CONTROL SYSTEM

The need for revision control support arose in a software engineering laboratory setting. A group of 25 students was divided up into four job classifications: coders, testers, management, and tool builders. The objective was to develop as a group a program to format the output resulting from a query to a relational database system. The program had to support interactive formatting and display of portions of the tabular query output data. The primary problem in developing the program was how to organize this relatively large group of participants so that they could work together to produce this moderate-sized program (less than 5000 lines of source code) in a relatively short time (10 weeks) working only part-time. Furthermore, the entire program development life cycle (from interviewing the "user" to testing the product) was to be acted out during the project, so the chaos that typically reigns during the early design stages of a project had to be planned for and supported.

The program was developed using the Large-Grain Data Flow (LGDF) model of software structure. Only a brief summary of the features of this model relevant to the design revision control problem are given here. Further details on this model of computation are discussed elsewhere[5] [6].

3.1. Overview of the Large-Grain Data Flow Computation Model

LGDF programs are made up of *processes* connected by uni-directional *datapaths* resembling UNIX "pipes" [7]. (See Fig. 1.) LGDF processes are activated and controlled by the arrival, consumption, and production of data values on associated data paths. At various stages of development of an LGDF process network, the circles can represent either program stubs or actual programs. The directed arcs represent data interfaces along which messages can be passed. Data interfaces are initially abstract, perhaps described only by a name. At later stages of development, concrete data type declarations are associated with each arc. Simulations of system execution can be run

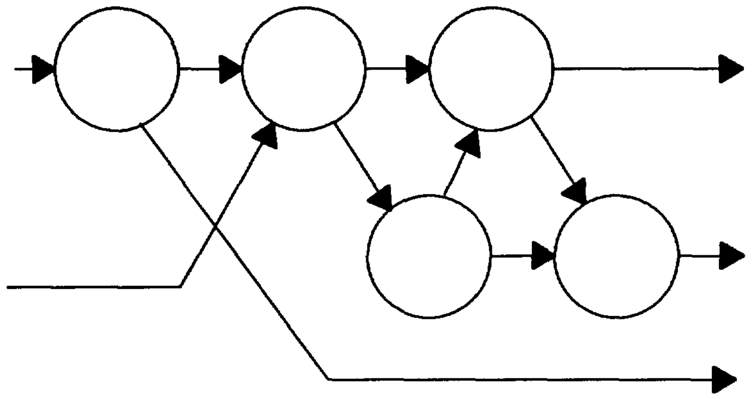

Fig. 1. A Large-Grain Data Flow process network.

at very early stages of specification by specifying default data production and consumption actions.

For this project, we needed a way to break up the program into a fairly large number of process modules (say 40) so that the 16 programmers could each have several modules to work on. However, since this is not that large a program, the resulting modules were decidedly *not* independent. Also, since we had over 20 people attempting to contribute to the modularization of the system during the the initial design phases, the system "design" was anything but stable during the first four or five weeks of the project.

The basic pieces of design data we needed to control were:

1) *documentation* (user requirements, management memos, etc.)
2) *test data* (both system-wide and local unit test data sets)
3) *LGDF process definitions* (program source code)
4) *LGDF data definitions* (data type interface definitions shared by two or more processes)
5) the *wirelist* (encoding which datapaths are inputs and outputs of which processes)

For this experiment, we did not attempt to maintain formal links between documentation or test data, and the evolving LGDF model.

3.2. Design Revision Control Tools for LGDF

We developed a set of UNIX shell scripts that used the facilities of RCS to help coordinate ongoing multiple simultaneous updates to LGDF process definitions, data definitions, and the "wirelist". The objective of the tools was to ensure that at any point a consistent test version of the entire program could be built that included all the latest work contributed by anyone in the class. We also wanted to have available at all times an up-to-date picture of the current module structure and an indication of who was currently working on what parts of the system. The shell scripts were used to provide a customized user interface to the basic RCS facilities, as well as to provide some LGDF-specific features.

RCS provides facilities to check-out (make a local copy) of a system file stored in a special RCS directory. The RCS "co" facility allows check-out of a file under version control either with or without a lock being set. A check-out with a lock provides protection against the situation where two programmers check- out the same piece of program text at approximately the same time, both make changes and then check the new versions back in. (Only the last person to check-in the file will have any effect). An attempt to check-out a file already locked by another programmer results in an informative message giving the owner of the lock and the version number in contention. RCS also provides a facility to break another person's lock (see "sebo" below).

What we wanted to provide programmers was a way to create the "sheltered" work environment mentioned above where documentation, test data, process specifications, and/or datapath definitions could be experimented with (and hopefully improved) in a way that did not preclude a large number of other people from doing the same thing simultaneously on different parts of the system design. Given below is a brief description of the functions developed for the laboratory project:

seco — (Software Engineering Check-Out) Activates the RCS check-out facility (with lock) on the specified files. If a file is currently checked out, the file is not checked out, and the standard RCS error message is given.

sebo — (Software Engineering Break-Out) Breaks the lock on a specified RCS file and version and then checks out the file (with a new lock). Mail is sent automatically to the owner of the previous lock indicating the event.

secc — (Software Engineering macro expansion and C Compile) Macro expands local (checked-out) LGDF process specifications and compiles the resulting C source code.

seci — (Software Engineering Check-In) Checks to see whether the specified files have been changed. If a file has not been changed since its last check-in, the local copy is merely deleted. For changed program source files, if an appropriate object file has been successfully generated by "secc", the object file is archived in a master system object file library and the source file is checked in. Otherwise, an appropriate error message is generated.

sebi — (Software Engineering Bag-It) Deletes any local copies of the specified files, and releases associated locks.

seld — (Software Engineering LoaD) Builds an executable test system in the current directory, with any locally available (changed) object files over-riding archived object files.

selog — (Software Engineering LOG) Lists the RCS change history for the specified files.

3.3. An Example of Use of the LGDF Revision Control Tools

Suppose a programmer wanted to work on (change) a particular process specification. To prevent another programmer from changing the data interface specification to this module out from underneath her/him, the programmer would check-out (and thus lock) that process specification and all directly associated data path specifications. Now the programmer can safely change anything within the program and be protected from global changes. The programmer would compile and test the modified process with appropriate test data, and if satisfied, check-in the changed process code†.

Suppose as a result of work within the module, the programmer decides that a global data interface definition (a datapath in LGDF terminology) had to be changed. The programmer would then need to check out the process specification on the other end of the data link, and all datapaths associated with that process. Now, as before, both processes must at least compile successfully before the system would allow a check-in. It can be seen from this example that even a minor change to an intermodule data interface can be a much more serious disruption to the system design than major changes to the inner workings of a module.

3.4. Shortcomings of RCS Facilities

While the RCS tools worked generally as expected, there were a few relatively minor but annoying features of the revision control facilities it provides.

The first problem arose because RCS does not maintain the current version of a file in a user-accessible form. This meant that when we wanted to see if a particular file checked out with a lock had been changed, we had no way to do this other than breaking the lock on the old revision and checking it out again. We programmed around this problem by saving an extra copy of the latest version of all files in another directory on every check-in.

The second problem arose because there is only one kind of lock available under RCS, essentially a lock for exclusive update privilege. This meant that two programmers working on "adjacent" programs (those that shared a data interface) simultaneously, a frequent occurrence, would have to exchange mail (breaking each others locks), even though neither programmer wanted or needed to update the shared datapath specification. Provision of a "read privilege" lock that would not conflict with another read lock would have been very helpful.

4. A DESIGN REVISION CONTROL TESTBED

We propose to implement a software design revision control system incorporating the ideas of syntactic and semantic consistency described above. The purpose of

†If the process was changed, the shell script required that a corresponding object file exist before allowing the check-in. This simple check turned out to be a very powerful way to ensure that a consistent executable test system could always be built!

implementation will be to experiment with those ideas in practice; in particular, to perform *in vivo* evaluations in graduate classes doing software engineering projects.

The proposed system in its initial form will be an experimental testbed, not a production system. It is intended to be used mainly in experiments to evaluate its features, but only by forgiving and motivated users. The best way to construct systems for experimentation is to make them as cheap and flexible as possible so that ideas emerging during evaluation can be incorporated. Use of existing tools (such as the prototype Large-Grain Data Flow tools, "RCS" and "make") where possible should prove beneficial. We also intend to make extensive use of a simple graphics-based user interface, so that complex information about inter-related components can be displayed and understood quickly.

5. CONCLUSION

Current revision control technology is appropriate only when a system's design is relatively stable, so that the configuration and interconnection of modules changes only slowly. The more isolation between modules, the better. Extending revision control methods to very large projects, or to software designs consisting of a great many highly-interdependent, tightly-coupled modules, or both, will require improvements in our current software engineering tools. Study of these problems using as case studies specialized, highly constrained models of software structure, such as LGDF, should prove beneficial in extending the methods to more traditional software development techniques.

6. REFERENCES

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

[2] Walter F. Tichy, "Design, Implementation, and Evaluation of a Revision Control System," in *Proc. 6th Int. Conf. on Software Engineering,* Tokyo, Japan, Sept. 1982, pp. 58-67.

[3] Alan L. Glasser, "The Evolution of a Source Code Control System", *Software Engineering Notes,* Vol. 3, No. 5, Nov. 1978, pp. 122-125.

[4] Stuart I. Feldman, "Make — A Program for Maintaining Computer Programs", *Software Practice and Experience,* Vol. 9, No. 4, April 1979, pp. 255-265.

[5] R. G. Babb II, "Data-driven implementation of data flow diagrams," in *Proc. 6th Int. Conf. on Software Engineering*, Tokyo, Japan, Sept. 1982, pp. 309-318.

[6] R. G. Babb II, "Parallel Processing with Large-Grain Data Flow Techniques," *Computer*, Vol. 17, No. 7, July 1984, pp. 55-61.

[7] B. W. Kernighan and R. Pike, *The UNIX Programming Environment.* Englewood Cliffs, NJ: Prentice-Hall, Inc., 1984.