Replacing Version-Control with Job-Control

Geoffrey M. Clemm

Evolutionary Software Inc.
55 Commonwealth Road, Watertown, Mass. 02172
geoff@harvard.harvard.edu

Abstract

Version-control is a mechanism for managing the multiple versions of the software objects that are created during the software development process. Traditionally, version-control consists of providing tools for generating a branching tree of versions, with facilities for reserving a given version for modification. In the Workshop System the focus of version-control is shifted from the objects produced during the software process to the software process itself. Objects called *jobs* are created in a project database to explicitly instantiate the process information. The Workshop System then provides operations for manipulating jobs, with these manipulations providing the functionality normally associated with version-control.

Keywords : job-control, version-control, Workshop System, software engineering environment, process programming, process tracking, SE-KRL, rule-based, object-oriented.

1 Introduction

A central problem in software engineering is managing the multiple source versions of software objects produced during the software development process. In practice, there are three distinct reasons for maintaining multiple source versions : configuration management, history management, and concurrency management. Historically, these three aspects of version-control are supported by separate features of a version-control system. Unfortunately, this separation increases the burden for a user of the system, because the user must separately specify the information necessary for each form of management.

In this paper, we propose that with sufficient support from the software environment, an alternative mechanism called *job-control* can provide a uniform and consistent basis for all three forms of version management. This not only decreases the burden on the software engineer, but also can result in qualitative improvement in version management, since each increment of effort devoted to one area of version management can produce a corresponding improvement in the other areas.

1.1 Configuration Management

Configuration management allows a user to specify alternative configurations of the software system through the selection of the appropriate versions. This is supported by associating attributes with each software version, and then allowing a configuration to be specified by describing the set of desired attributes.

The differences between existing systems that provide configuration management consist in the kinds of attributes that can be used and the kinds of queries that can be defined in terms of these attributes. Minimally, there are a set of *key* attributes that are guaranteed unique across all versions of a given software object, and there is a way of listing the desired versions for a given configuration. In SCCS [12] and its descendants, these key attributes take numeric values – in later systems such as RCS [13], additional attributes such as *revision-state* and symbolic revision-keys are provided. A configuration is then specified by listing the attributes desired for each of the components of the configuration.

More sophisticated approaches to the problem of specifying the versions desired can be found in systems such as DSEE [9] and Adele [5]. In DSEE, a sequence of version specifications can be provided for a system, and DSEE will select the version of each component that satisfies the earliest version specification in the sequence. In addition, the version specifications can

0

be context sensitive, in the sense that different version specification sequences can be used for different subtrees of the tree of modules for a given system. In Adele, a database is used to store the version attributes, and standard database queries can then be used to specify specific version configurations.

Another characteristic of the more advanced configuration systems such as Adele and DSEE is that they allow the user to specify that certain versions are of particular importance (i.e. versions in a distributed release of a software system), and therefore should not be modified or deleted. This feature is commonly known as *baselining.*

The main problem with even the database supported systems for configuration management is the separation between configuration management and the rest of the software engineering process. A software engineer first creates the software objects and then annotates them with additional configuration management attributes. This results in an environment where time spent generating the information for effective configuration management detracts from the time available for producing more software objects. In addition, the software engineer must maintain two mental models, one for the software objects themselves and one for the version descriptions that must be used to specify a configuration.

1.2 History Management

History management provides the user with information about the historical development of existing software objects to help guide future development. In existing version-control systems, history management is usually provided in one of the following ways : internal annotations in the form of comments in the software object source, textual *log message* attributes associated with a new version when the version is completed, and special modification request objects associated with a new version when the version is initiated.

1.2.1 Internal Annotation

Internal annotation is feasible in any version-control system. The only requirement is that the syntax of the software objects being manipulated allow the addition of comments to the software object.

Problems with this approach include :

1. clumsy or inefficient history browsing, since either the entire object must be viewed with the annotations, or a tool must extract the annotations from the object.
2. varying and conflicting annotation styles.
3. difficulty in specifying an annotation that should apply to more than one object.

1.2.2 Log-Messages

The use of a textual log-message attribute is provided by systems such as RCS and its descendents through the log entry specified by the user when a new version of a file is completed. This approach addresses the first and (to some extent) the second problem associated with internal annotations, but the third problem remains.

An additional problem with the log approach is that without internally or externally generated discipline, the log messages frequently contain little information of value. This is a result of the gap between the time a log entry is specified and the time the log entry is used. Although a software engineer believes that a carefully designed log message could prove vital at some future point, he knows for a fact that if the next bug is not fixed he will fail to meet his production deadline, and yet another terse message appears in the version log. This problem is further exacerbated by organizations in which the person using the log entries tends to be different from the person generating them. Here even the motivation of enlightened self-interest is minimized – the result is log messages with the minimal information that altruism or management enforcement requires.

1.2.3 Modification Requests

A more advanced form of history management involves the creation of special *modification request* objects. These objects allow software engineers (or their managers) to describe tasks that should be performed, and then associate new versions of a file with the appropriate modification request. Examples of this kind of history management can be found in the SERS [11] and the ISTAR system [4] (in which the history management objects are called *contracts*). The DSEE system also provides this functionality with the notion of *tasks.*

A generic problem with the use of modification requests for history management is that modification requests are designed to track the specification of what should be done, while history management needs to track what actually was done. Unless this distinction is explicitly recognized by the history management system, either the development process will be unduly constrained or the actual history will be lost.

In addition, a common problem with current commercial systems that support modification requests is that they are too inflexible. The development process is a very idiosyncratic one, and an automated system that cannot be tailored to closely model the organizations actual development process will generate more problems than it will solve. More flexible approaches can be found in a research environments such as CLF [2], but these systems are based on technology that does not scale up to the problems faced by large-scale commercial software engineering (i.e. hundreds of programmers working concurrently on millions of lines of code).[1]

1.3 Concurrency Management

Concurrency management handles the potentially conflicting updates from multiple concurrent developers. In this case, multiple source versions arise not because they are desired, but because different users generate multiple versions of objects when they copy them into text editor buffers or into local file systems for manipulation. In the absence of concurrency management, a common scenario is for two users to be editing the same file concurrently. The changes of one user are saved when he exits from his editing session, only to be overwritten and lost when the second user exits from her editing session.

1.3.1 Check-Out/Check-In

The most popular approach to this problem is found in systems such as SCCS, RCS, and DSEE. These systems require that a file that is to be modified be *checked-out* by a software developer, and then require that the new version be *checked-in* when it is completed. If a second check-out request is received while an earlier check-out is still active (i.e. has not been followed by a check-in), the second check-out is disallowed and an error message is generated indicating which user currently has the file checked-out.

The basic problem with this approach is that production schedules can require that two tasks be performed immediately, and these tasks involve different people making changes to the same file concurrently. In this case, either the production schedule must be delayed or the strict check-out rules must be suspended. If suspending the check-out rules is difficult or time-consuming, a developer will often skip the check-out process until the correct set of changes have been determined, thereby effectively eliminating the concurrency management support.

1.3.2 Copy/Modify/Merge

An alternative paradigm for concurrency management results when it is accepted that the set of objects to be changed cannot be effectively specified until after the changes have been made. This paradigm is followed in the NSE [8] system, and is described as *Copy/Modify/Merge*. In this approach, a user requests copies of all files of interest, modifies some subset of the requested files, and then merges the modified copies back into the original files. The NSE system is responsible for indicating to a user when an intervening merge to a file has taken place, and provides tools to simplify the combine-merge process.

A problem with this approach is that in some situations a file really should be locked against other modifications or perhaps a specific warning should be given to any user that requests a copy. In general, the problem with NSE (as well as with the other systems that provide concurrency management) is that it only supports a single style of concurrency management. Within a given project, different concurrency management policies are appropriate for different software objects, for different users, and for different phases in the software development process. Unless these policies can be accurately specified and automated, either the general lack of constraints causes chaos in critical project sections, or the general enforcement of constraints prevents necessary work from being performed on schedule.

2 The Workshop System

The Workshop System is a software engineering environment designed to support all phases of development by large teams of programmers working concurrently on developing large software systems. An overview of the Workshop System can be found in [3] – here only the aspects of the Workshop System essential for version-control will be described.

2.1 The Global Database

In the Workshop System, all information in a software project is stored in the form of objects in a shared project database called the Global Database. Unlike database-supported programming environments such as Adele [5] and PCTE [7] in which the source objects are primarily files, the Workshop System encourages and supports the use of much finer grained source software objects, such as function or type definitions. These fine grained objects can then be arbitrarily grouped as elements of aggregation objects, which in turn can be nested within other aggregation objects.

The three basic classes of aggregation objects are *modules*, *jobs*, and *descriptions*. Module objects are used to specify structural aggregations, and can model constructs such as files, directories, packages, and systems. Orthogonal process-oriented aggregations are

[1]The question of how the Workshop System scales up to large-scale commercial software engineering is beyond the scope of this paper – interested parties should contact the author at Evolutionary Software Inc.

provided by job objects – the components of a job are all the software objects produced during a single step of the software process. Finally, description objects are used to specify dynamic aggregations. The defining attribute for a description is a database query, and the components of a description are the set of objects that currently match that query.

Although aggregation objects are used in the Workshop System for a variety of purposes, in the context of version-control, description objects are used as system models, modules are used to specify baselines, and jobs are used to identify versions. This use of jobs will be explained fully in the section on *job-control*.

2.2 Process Programming and Process Tracking

A central principle of the Workshop System is that the software development process must be explicitly captured and manipulated as a software object. In the Workshop System, there are two essential components to capturing the software process : *process programming* and *process tracking*.

2.2.1 Process Programming

Process programming [10] is the specification of the software process in an executable language; the execution of a process program guides and supports the desired behavior of a software engineer. The notion of process programming is still controversial, but has received considerable conceptual support from the research community [6], with the central problem being the design of a suitable process programming language.

In the Workshop System, the process programming language is called SE-KRL[2] (pronounced *see-krull*). SE-KRL was specifically designed for the specification of software engineering knowledge. It combines object-oriented and rule-based technology to maximize the ease of modifying and extending existing process programs through the additions of new classes, operations, and rules. A summary of SE-KRL's language constructs can be found in [3].

From a user's perspective, the Workshop System appears to be an expert system that supports the software process described by a SE-KRL program. For naive users, this process can be tailored by selecting alternative pre-defined modules for the SE-KRL program. For sophisticated users, the SE-KRL program can be hand-tailored to provide an environment that directly supports a personal software process.

[2]Software Engineering Knowledge Representation Language

2.2.2 Process Tracking

Process tracking consists of recording the actual development process of a software system. In the Workshop System, process tracking is performed by the creation of special objects called *jobs* that capture the individual steps of the software process. The attributes of these job objects capture the relationships between these steps and the other software objects that result from the software process. A complete description of job objects appears below in the context of of *job-control*.

The need for process tracking illustrates the distinction between process programming in the Workshop System and other descriptions of process programming that omit this concept. When a process program is conceived as a procedure that strictly enforces a given software process, process tracking is unnecessary. Since the software process must follow the process program, the only relevant data for evaluating the process program is the software products that are produced. In contrast, a process program in the Workshop System is conceived as primarily an advisor or guide. Some dangerous things will be forbidden and various alternatives will be suggested, but the decision of what to do next will primarily be the result of the creative intuition of the software engineer. In this paradigm, it is essential to track the actual software process which is the result of this creative intuition. If a given software project succeeds, the jobs can be examined for patterns that could be added to the process program. If the project fails, the jobs can be examined for choices that conflicted with the process program (in which case the process program is not at fault, but rather the software engineer who should have heeded the process program's advice). Alternatively, if a particular product fails, the jobs that produced that product can be examined for patterns that the process program should detect and advise against.

In the context of process programming, it is especially important that the concept of process tracking not be confused or blended with that of modification requests (contracts, tasks). A modification request is an indication of what should be done (possibly even generated by the process program) – process tracking requires an accurate indication of what actually has been done.

2.2.3 Process Tracking and Version-Control

Once the value of process tracking has been accepted, the practical problem remains of how to elicit this information from the only accurate source – the software engineer performing the process. Some tracking can be done automatically, but without major advances in automated learning and induction, the majority of process tracking information must be provided by a human. This problem closely resembles the problem indicated earlier with eliciting accurate version-control information, namely, that the software engineer is busily attempting to clear his backlog of work, with little time for "inessential" activities such as process tracking.

A solution to both of these problems is to design the software environment so that accurate process tracking provides significantly improved version-control. If this can be achieved, the software engineer will provide accurate process tracking as a side-effect of his normal version-control activity. The software engineer will view process tracking as just an improved form of version-control, while the process programmer will view version-control as a way of eliciting accurate process tracking information.

3 Job-Control

In the Workshop System, the three aspects of version-control are unified into the single concept of *job-control.* The central idea of job-control is the introduction of software objects, called *jobs*, that are intended to record the actual software process that produced a software system. Examples of jobs would be *Create an Icon compiler, Add color graphics to the user interface*, or *Prevent user interrupt signals from crashing the system.* Using the concepts of the previous section, job-control is the application of process tracking to the problem of version-control.

In order to develop a more concrete understanding of what a job is, it is valuable to know what attributes a job can possess. Initially, the attributes of a job include the following information :

Name : a short (one-line) description of the job.

Documentation : a longer and more detailed description of the job.

Job-of : the relationship with a person performing the job (a *person* object exists in the Global Database for each user of the system).

Job-State : Future if the job has not been started, Open if the job is being performed, Suspended if the job is unfinished, Completed if the job is finished, and Aborted if the job is expected to remain unfinished.

Has-Product : the software objects produced during the performance of the job.

Has-SubJob : jobs that are components of the job.

Open-Job-of : *sessions* during which this job was actively being performed, where a session is a particular kind of job that is automatically created for a user when the user logs in to the system.

Has-Prerequisite : jobs that must be performed before this job.

Has-Alternative : jobs that are alternative ways of performing the same task.

Using SE-KRL, the job model can easily be extended to include any additional information that either is necessary to accurately record the software process or is of interest to a given user or organization. For example, an integer valued *Job-Priority* attribute would be very useful both to record what were considered the critical job steps, and to support automated job scheduling. Once the desired job-attributes have been specified, the rule-based component of SE-KRL can be used to automate the generation of much of the job information, with input from the user required only as a last resort.

In the following sections, the use of job control to support all three aspects of version management will be described.

3.1 Configuration Management

When job-control is available for configuration management, the notion of a *job-stamp* replaces the more common notion of a date-stamp or version number. The job-stamp of a version is the job that produced that version. The most noticeable effect of the use of job-stamps is that English text replaces the dewey-decimal numbering or date that is found in many version-control systems. For example, in configuration specifications, instead of

```
fast-sort<1.7.3>
```

or

```
fast-sort<3:15:22,6/2/88>
```

one would see

```
fast-sort
<allow user defined comparison operators>
```

The important point here is that the user is not burdened with creating a unique key for configuration purposes. Instead, the same information that was used to keep track of what was being done, is re-used to request specific configurations. The same mental model used to create the software objects can then also be used for the specification of system configurations.

A date-stamp (or even dewey-decimal numbering) could alternatively be used for people or situations that require them, but especially in the case of composite selection specifications[3], it is expected that the job stamp will be far more useful.

Another feature of the Workshop System that is orthogonal to the notion of job-control, but that is an important contribution to decreasing the burden of configuration management, is the use of a common language for information retrieval and configuration specification. This means that the same concepts used to retrieve information for browsing and editing are also used for configuration specification. The result is that an arbitrary user is prepared to read and even modify configuration specifications based on his experience with browsing and editing.

3.2 History Management

Job-control is essentially a combination of the best characteristics of the three forms of history management, functionally extended to provide the capabilities needed for configuration and concurrency management. Like modification requests, job-control introduces a new kind of software object that is related to all the source objects created while performing a given task, as opposed to comments that appear in a single source object. Like log entries, a job is intended to model what has been done, as opposed to modification requests that specify what should be done. Like source comments, jobs are related to the individual programming objects created, as opposed to just the file containing those objects.

In addition, since jobs are objects whose structure is explicitly declared in the SE-KRL language, the problems of conflicting annotation styles can be effectively addressed. For example, a SE-KRL rule could ensure that every job had a *Job-Kind* attribute classifying the job as an *extension*, a *bug-fix*, or an *efficiency improvement*. Based on this classification, other rules could gather further information appropriate to the specified class (such as whether the efficiency improvement is in space or time, and what the expected improvement factor is).

Even with the automated support provided by the Workshop System, the problem of eliciting clear and accurate job descriptions remains. As indicated earlier, a distinction between jobs and modification requests is that a job is intended to capture the actual software process performed, not just the description of what initially was expected to be performed. In the commercial world, the software engineer's backlog of "work to be done yesterday" tends to minimize the effort expended on tasks that are not essential to clearing this backlog. Unfortunately, ensuring that a job description remains accurate tends to fall into this non-essential category.

[3] a composite selection specification is one that involves the combination of several attributes to specify a desired version

The approach used in the Workshop System for the problem of inaccurate job descriptions is to increase the perceived value of accurate descriptions. This is achieved by ensuring that the description of the job that produced an object is displayed whenever the object is manipulated through the Workshop System's user interface. Based on initial usage of the Workshop System, this approach not only has encouraged accurate updating of job descriptions, but even more importantly, it has been the most effective way of expanding the software engineer's attention from the software objects to the software process itself.

It is important to note that a critical component to the success of job-control for history management is the fine-grained nature of software objects in the Workshop System. In a file based approach, where only the current job would be attached to an editor buffer of objects being modified, the sense of historical development is lost. In the Workshop System, each object in an editor buffer (i.e. function, type-definition, etc.) would be annotated with the job that produced it, thereby giving a historical summary of the processes used to bring that module to its current state. In addition, if the software engineer is interested in the evolution of a specific definition, this information is directly obtainable – in a file-based system, it is often necessary to carefully scan dozens of successive file versions to detect the four that affected the definition of interest.

3.3 Concurrency Management

The default concurrency control model of the Workshop System is most similar to the NSE model. Software objects are downloaded to a programmer's workstation, manipulated, and then some set of new versions can be submitted back into the Global Database. The main differences between the Workshop System and NSE models result from smaller grain size, more support for intermediate results, and flexible addition of stronger concurrency management constraints.

The reason for the simplicity of most merges in the NSE system is that concurrent changes tend to apply to different definitions in a given file. In the Workshop System, each definition is a separate object with its own version history, therefore the situations resulting in these simple NSE merges do not even produce collisions in the Workshop System.

The second improvement over the basic copy/modify/merge model is the use of jobs as a mechanism for indicating intermediate or partial results. As soon as some exploratory set of changes have been made, a user of the Workshop System is encouraged to save them in the Global Database. The job of which these new objects are products is marked as still being *open*, therefore it is explicitly known that these are just exploratory changes (i.e. can be modified or canceled at will by the user). Anyone considering (or in the process of) modifying the source version of any of these objects would be immediately notified of this exploratory work. Such people can then decide to synchronize or to continue working in parallel. In addition, the job name and documentation provides information about why this work is being performed. The user is free at any point to leave a job in an unfinished state, knowing that the Global Database contains all partial results and that anyone looking at these objects will be aware of their provisional status.

Finally, if more stringent concurrency control such as the check-out/check-in paradigm is found to be desirable or necessary, it can be specified in SE-KRL to apply to critical components of the software system or critical phases of the software process. The meta-policy of how the concurrency control policies can be changed could in turn be specified in SE-KRL.

4 Conclusions

Replacing version-control with job-control can significantly improve the quality of support provided by a software environment. Job-control is only feasible though, if the user is provided with mechanisms for specifying both the software model and the software process that is appropriate for the work being performed.

Essential technical components of job-control are decreased grain size for the software objects, the use of job-stamps to distinguish different versions, concurrency control through job rather than version attributes, and easily extended mechanisms for automatically collecting any information that does not require direct user input.

When job-control is effectively supported by the software environment, software developers are made explicitly aware of the software process that is underlying their activities, and are encouraged to manipulate descriptions of these processes so that they track the actual development process. Unlike modification-request systems in which process tracking can appear as a burdensome activity orthogonal to the actual generation of software, job-control presents an improved version-control model that produces accurate process tracking as a beneficial side-effect.

5 Implementation Status

The Workshop System is implemented in CommonLisp and ART [1] and runs on a network of Symbolics using Chaos communication protocols. The Global Database component of the Workshop System is implemented in CommonLisp, but is designed to be easily replaced by a commercial object-oriented database. The Workshop System is currently being ported to C using TCP/IP communication protocols, under a one year contract with HP Research Labs.

References

[1] *The ART User's Manual.* Inference Corporation, 5300 W. Century Blvd, LA, CA.

[2] Robert M. Balzer. Living with the next generation operating system. In *Proc. 4th World Computer Conference*, September 1986.

[3] Geoffrey M. Clemm. The Workshop System - a practical knowledge-based software environment. In *Proc. 3rd Software Engineering Symposium on Practical Software Development Environments*, pages 55–64, November 1988.

[4] Mark Dowson. ISTAR – an integrated project support environment. In *Proc. 2nd Software Engineering Symposium on Practical Software Development Environments*, pages 27–33, December 1986.

[5] Jacky Estublier. A configuration manager: the Adele data base of programs. In *Proc. Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 140–147, June 1985.

[6] Peter H. Feiler and Roger Smeaton. Managing development of very large systems : implications on integrated environments. In *Proc. 1st International Workshop on Software Version and Configuration Control*, pages 62–82, January 1988.

[7] Ferdinando Gallo, Regis Minot, and Ian Thomas. The object management system of PCTE as a software engineering database management system. In *Proc. 2nd Software Engineering Symposium on Practical Software Development Environments*, pages 12–15, December 1986.

[8] Masahiro Honda. Support for parallel development in the Sun network software environment. In *Proc. 2nd International Workshop on Computer-Aided Software Engineering*, pages 5-5 – 5-7, 1988.

[9] D.B. Leblang and R.P. Chase. Computer-aided software engineering in a distributed workstation environment. In *Proc. 1st Software Engineering Symposium on Practical Software Development Environments*, pages 104–112, April 1984.

[10] Leon J. Osterweil. The software process is software, too. In *Proc. 9th International Conference on Software Engineering*, 1987.

[11] Saeed Reghabi and David G. Wright. Software engineering release system (SERS). In *Proc. 1st International Workshop on Software Version and Configuration Control*, pages 244–263, January 1988.

[12] Mark J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, December 1975.

[13] Walter F. Tichy. Design, implementation, evaluation of a revision control system. In *Proc. 6th International Conference on Software Engineering*, pages 58–67, 1982.

[14] Walter F. Tichy. Tools for software configuration mangement. In *Proc. 1st International Workshop on Software Version and Configuration Control*, pages 1–20, January 1988.