Elements of Software Configuration Management

EDWARD H. BERSOFF, SENIOR MEMBER, IEEE

Abstract—**Software configuration management (SCM) is one of the disciplines of the 1980's which grew in response to the many failures of the software industry throughout the 1970's. Over the last ten years, computers have been applied to the solution of so many complex problems that our ability to manage these applications has all too frequently failed. This has resulted in the development of a series of "new" disciplines intended to help control the software process.**

This paper will focus on the discipline of SCM by first placing it in its proper context with respect to the rest of the software development process, as well as to the goals of that process. It will examine the constituent components of SCM, dwelling at some length on one of those components, configuration control. It will conclude with a look at what the 1980's might have in store.

Index Terms—**Configuration management, management, product assurance, software.**

INTRODUCTION

SOFTWARE configuration management (SCM) is one of the disciplines of the 1980's which grew in response to the many failures of our industry throughout the 1970's. Over the last ten years, computers have been applied to the solution of so many complex problems that our ability to manage these applications in the "traditional" way has all too frequently failed. Of course, tradition in the software business began only 30 years ago or less, but even new habits are difficult to break. In the 1970's we learned the hard way that the tasks involved in managing a software project were not linearly dependent on the number of lines of code produced. The relationship was, in fact, highly exponential. As the decade closed, we looked back on our failures [1], [2] trying to understand what went wrong and how we could correct it. We began to dissect the software development process [3], [4] and to define techniques by which it could be effectively managed [5]-[8]. This self-examination by some of the most talented and experienced members of the software community led to the development of a series of "new" disciplines intended to help control the software process.

While this paper will focus on the particular discipline of SCM, we will first place it in its proper context with respect to the rest of the software development process, as well as to the goals of that process. We will examine the constituent components of SCM, dwelling at some length on one of those components, configuration control. Once we have woven our way through all the trees, we will once again stand back and take a brief look at the forest and see what the 1980's might have in store.

Manuscript received April 15, 1982; revised December 1, 1982 and October 18, 1983.

The author is with BTG. Inc.. 1945 Gallows Rd.. Vienna, VA 22180.

SCM IN CONTEXT

It has been said that if you do not know where you are going, any road will get you there. In order to properly understand the role that SCM plays in the software development process, we must first understand what the goal of that process is, i.e., where we are going. For now, and perhaps for some time to come, software developers are people, people who respond to the needs of another set of people creating computer programs designed to satisfy those needs. These computer programs are the tangible output of a thought process–the conversion of a thought process into a product. The goal of the software developer is, or should be, the construction of a product which closely matches the real needs of the set of people for whom the software is developed. We call this goal the achievement of "product integrity." More formally stated, product integrity (depicted in Fig. 1) is defined to be the intrinsic set of attributes that characterize a product [9]:

- that fulfills user functional needs;
- that can easily and completely be traced through its life cycle;
- that meets specified performance criteria;
- whose cost expectations are met;
- whose delivery expectations are met.

The above definition is pragmatically based. It demands that product integrity be a measure of the satisfaction of the real needs and expectations of the software user. It places the burden for achieving the software goal, product integrity, squarely on the shoulders of the developer, for it is he alone who is in control of the development process. While, as we shall see, the user can establish safeguards and checkpoints to gain visibility into the development process, the prime responsibility for software success is the developer's. So our goal is now clear; we want to build software which exhibits all the characteristics of product integrity. Let us make sure that we all understand, however, what this thing called software really is. We have learned in recent times that equating the terms "software" and "computer programs" improperly restricts our view of software. Software is much more. A definition which can be used to focus the discussion in this paper is that software is information that is:

- structured with logical and functional properties;
- created and maintained in various forms and representations during the life cycle;
- tailored for machine processing in its fully developed state.

So by our definition, software is not simply a set of computer programs, but includes the documentation required to define, develop, and maintain these programs. While this notion is not very new, it still frequently escapes the software

Reprinted from *IEEE Transactions on Software Engineering*, Volume SE-10, Number 1, January 1984, pages 79-87.

EH0263-4/87/0000/0430$01.00 © 1984 IEEE

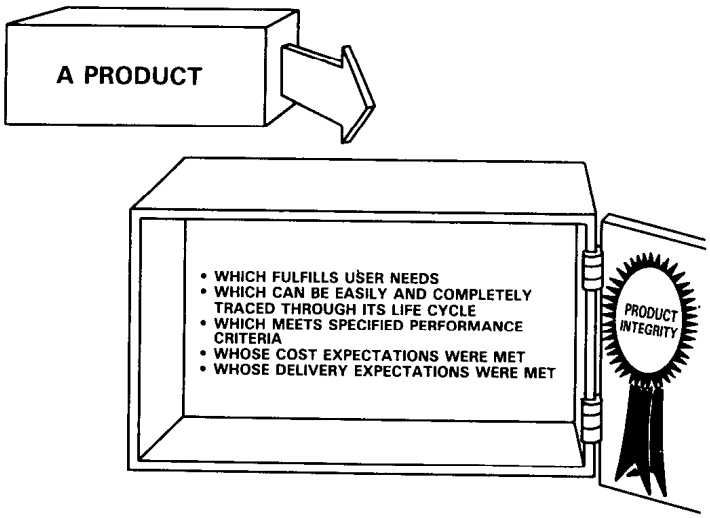

Fig. 1. Product integrity.

development manager who assumes that controlling a software product is the same as controlling computer code.

Now that we more fully appreciate what we are after, i.e., to build a software product with integrity, let us look at the one road which might get us there. We have, until now, used the term "developer" to characterize the organizational unit responsible for converting the software idea into a software product. But developers are, in reality, a complex set of interacting organizational entities. When undertaking a software project, most developers structure themselves into three basic discipline sets which include:

- project management,
- development, and
- product assurance.

Project management disciplines are both inwardly and outwardly directed. They support general management's need to see what is going on in a project and to ensure that the parent or host organization consistently develops products with integrity. At the same time, these disciplines look inside a project in support of the assignment, allocation, and control of all project resources. In that capacity, project management determines the relative allocation of resources to the set of development and product assurance disciplines. It is management's prerogative to specify the extent to which a given discipline will be applied to a given project. Historically, management has often been handicapped when it came to deciding how much of the product assurance disciplines were required. This was a result of both inexperience and organizational immaturity.

The development disciplines represent those traditionally applied to a software project. They include:

- analysis,
- design,
- engineering,
- production (coding),
- test (unit/subsystem),
- installation,
- documentation,
- training, and
- maintenance.

In the broadest sense, these are the disciplines required to take a system concept from its beginning through the development life cycle. It takes a well-structured, rigorous technical approach to system development, along with the right mix of development disciplines to attain product integrity, especially for software. The concept of an ordered, procedurally disciplined approach to system development is fundamental to product integrity. Such an approach provides successive development plateaus, each of which is an identifiable measure of progress which forms a part of the total foundation supporting the final product. Going sequentially from one baseline (plateau) to another with high probability of success, necessitates the use of the right development disciplines at precisely the right time.

The product assurance disciplines which are used by project management to gain visibility into the development process include:

- configuration management,
- quality assurance,
- validation and verification, and
- test and evaluation.

Proper employment of these product assurance disciplines by the project manager is basic to the success of a project since they provide the technical checks and balances over the product being developed. Fig. 2 represents the relationship among the management, development, and product assurance disciplines. Let us look at each of the product assurance disciplines briefly, in turn, before we explore the details of SCM.

Configuration management (CM) is the discipline of identifying the configuration of a system at discrete points in time for the purpose of systematically controlling changes to the configuration and maintaining the integrity and traceability of the configuration throughout the system life cycle. Software configuration management (SCM) is simply configuration management tailored to systems, or portions of systems, that are comprised predominantly of software. Thus, SCM does not differ substantially from the CM of hardware-oriented systems, which is generally well understood and effectively practiced. However, attempts to implement SCM have often failed because the particulars of SCM do not follow by direct analogy from the particulars of hardware CM and because SCM is a less mature discipline than that of hardware CM. We will return to this subject shortly.

Quality assurance (QA) as a discipline is commonly invoked throughout government and industry organizations with reasonable standardization when applied to systems comprised only of hardware. But there is enormous variation in thinking and practice when the QA discipline is invoked for a software development or for a system containing software components. QA has a long history, and much like CM, it has been largely developed and practiced on hardware projects. It is therefore mature, in that sense, as a discipline. Like CM, however, it is relatively immature when applied to software development. We define QA as consisting of the procedures, techniques, and tools applied by professionals to insure that a product meets or exceeds prespecified standards during a product's development cycle; and without specific prescribed standards, QA entails insuring that a product meets or

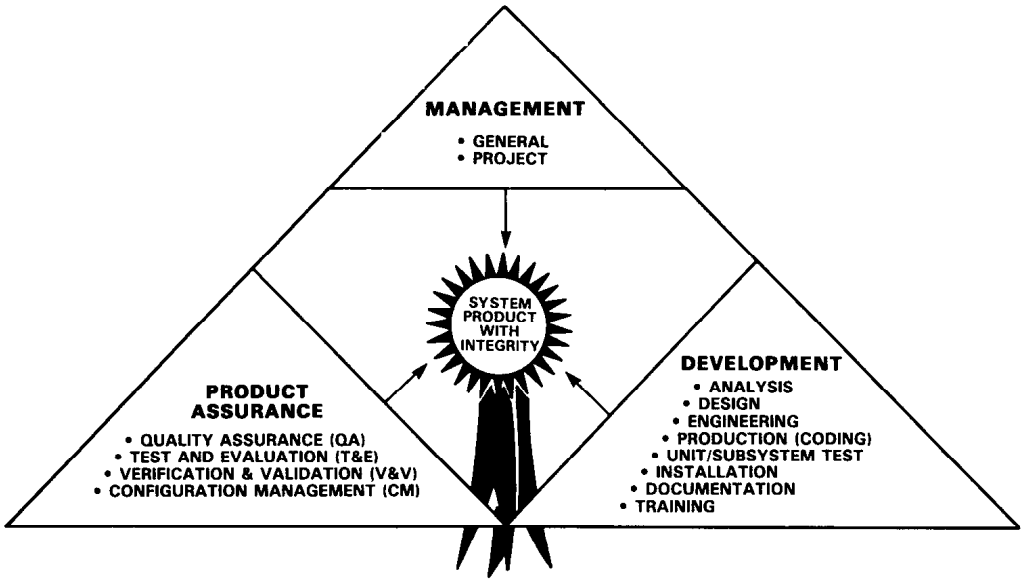

Fig. 2. The discipline triangle.

exceeds a minimum industrial and/or commercially acceptable level of excellence.

The QA discipline has not been uniformly treated, practiced or invoked relative to software development. First, very few organizations have software design and development standards that compare in any way with hardware standards for detail and completeness. Second, it takes a high level of software expertise to assess whether a software product meets prescribed standards. Third, few buyer organizations have provided for or have developed the capability to impose and then monitor software QA endeavors on seller organizations. Finally, few organizations have been concerned over precisely defining the difference between QA and other product assurance disciplines, CM often being subservient to QA or vice versa in a given development organization. Our definition of software given earlier suggests still another reason for the software QA discipline being in the same state as SCM so far as its universal application within the user, buyer, and seller communities. Software, as a form of information, cannot be standardized; only structures for defining/documenting software can be standardized. It follows that software development techniques can only be meaningfully standardized in relation to information structures, not information content.

The third of the four product assurance disciplines is validation and verification (V&V). Unlike CM and QA, V&V has come into being expressly for the purpose of coping with software and its development. Unlike QA, which prinicipally deals with the problem of a product's adherence to preestablished standards, V&V deals with the issue of how well software fulfills functional and performance requirements and the assurance that specified requirements are indeed stated and interpreted correctly. The verification part of V&V assures that a product meets its prescribed goals as defined through baseline documentation. That is, verification is a discipline imposed to ascertain that a product is what it was intended to be relative to its preceding baseline. The validation part of V&V, by contrast, is levied as a discipline to assure that a product not only meets the objectives specified through baseline documentation, but in addition, does the right job. Stated another way, the validation discipline is invoked to insure that the end-user gets the right product. A buyer or seller may have misinterpreted user requirements or, perhaps, requirements have changed, or the user gets to know more about what he needs, or early specifications of requirements were wrong or incomplete or in a state of flux. The validation process serves to assure that such problems do not persist among the user, buyer, and seller. To enhance objectivity, it is often desirable to have an independent organization, from outside the developing organization, perform the V&V function.

The fourth of the product assurance disciplines is test and evaluation (T&E), perhaps the discipline most understood, and yet paradoxically, least practiced with uniformity. T&E is defined as the discipline imposed outside the development project organization to independently assess whether a product fulfills objectives. T&E does this through the execution of a set of test plans and procedures. Specifically in support of the end user, T&E entails evaluating product performance in a live or near-live environment. Frequently, particularly within the miliatry arena, T&E is a major undertaking involving one or more systems which are to operate together, but which have been individually developed and accepted as stand-alone items. Some organizations formally turn over T&E responsibility to a group outside the development project organization after the product reaches a certain stage of development, their philosophy being that developers cannot be objective to the point of fully testing/evaluating what they have produced.

The definitions given for CM, QA, V&V, and T&E suggest some overlap in required skills and functions to be performed in order to invoke these disciplines collectively for product assurance purposes. Depending on many factors, the actual overlap may be significant or little. In fact, there are those who would argue that V&V and T&E are but subset functions of QA. But the contesting argument is that V&V and T&E have come into being as separate disciplines because conventional QA methods and techniques have failed to do an adequate job with respect to providing product assurance, par-

ticularly for computer-centered systems with software components. Management must be concerned with minimizing the application of excessive and redundant resources to address the overlap of these disciplines. What is important is that all the functions defined above are performed, not what they are called or who carries them out.

The Elements of SCM

When the need for the discipline of configuration management finally achieved widespread recognition within the software engineering community, the question arose as to how closely the software CM discipline ought to parallel the extant hardware practice of configuration management. Early SCM authors and practitioners [10] wisely chose the path of commonality with the hardware world, at least at the highest level. Of course, hardware engineering is different from software engineering, but broad similarities do exist and terms applied to one segment of the engineering community can easily be applied to another, even if the specific meanings of those terms differ significantly in detail. For that reason, the elements of SCM were chosen to be the same as those for hardware CM. As for hardware, the four components of SCM are:

- identification,
- control,
- auditing, and
- status accounting.

Let us examine each one in turn.

Software Configuration Identification: Effective management of the development of a system requires careful definition of its baseline components; changes to these components also need to be defined since these changes, together with the baselines, specify the system evolution. A system baseline is like a snapshot of the aggregate of system components as they exist at a given point in time; updates to this baseline are like frames in a movie strip of the system life cycle. The role of software configuration identification in the SCM process is to provide labels for these snapshots and the movie strip.

A baseline can be characterized by two labels. One label identifies the baseline itself, while the second label identifies an update to a particular baseline. An update to a baseline represents a baseline plus a set of changes that have been incorporated into it. Each of the baselines established during a software system's life cycle controls subsequent system development. At the time it is first established a software baseline embodies the actual software in its most recent state. When changes are made to the most recently established baseline, then, from the viewpoint of the software configuration manager, this baseline and these changes embody the actual software in its most recent state (although, from the viewpoint of the software developer, the actual software may be in a more advanced state).

The most elementary entity in the software configuration identification labeling mechanism is the software configuration item (SCI). Viewed from an SCM perspective, a software baseline appears as a set of SCI's. The SCI's within a baseline are related to one another via a tree-like hierarchy. As the software system evolves through its life cycle, the number of branches in this hierarchy generally increases; the first baseline may consist of no more than one SCI. The lowest level SCI's in the tree hierarchy may still be under development and not yet under SCM control. These entities are termed design objects or computer program components (see Fig. 3). Each baseline and each member in the associated family of updates will exist in one or more forms, such as a design document, source code on a disk, or executing object code.

In performing the identification function, the software configuration manager is, in effect, taking snapshots of the SCI's. Each baseline and its associated updates collectively represents the evolution of the software during each of its life cycle stages. These stages are staggered with respect to one another. Thus, the collection of life cycle stages looks like a collection of staggered and overlapping sequences of snapshots of SCI trees. Let us now imagine that this collection of snapshot sequences is threaded, in chronological order, onto a strip of movie film as in Fig. 4. Let us further imagine that the strip of movie film is run through a projector. Then we would see a history of the evolution of the software. Consequently, the identification of baselines and updates provides an explicit documentation trail linking all stages of the software life cycle. With the aid of this documentation trail, the software developer can assess the integrity of his product, and the software buyer can assess the integrity of the product he is paying for.

Software Configuration Control: The evolution of a software system is, in the language of SCM, the development of baselines and the incorporation of a series of changes into the baselines. In addition to these changes that explicitly affect existing baselines, there are changes that occur during early stages of the system life cycle that may affect baselines that do not yet exist. For example, some time before software coding begins (i.e., some time prior to the establishment of a design baseline), a contract may be modified to include a software warranty provision such as: system downtime due to software failures shall not exceed 30 minutes per day. This warranty provision will generally affect subsequent baselines but in a manner that cannot be explicitly determined *a priori*. One role of software configuration control is to provide the administrative mechanism for precipitating, preparing, evaluating, and approving or disapproving all change proposals throughout the system life cycle.

We have said that software, for configuration management purposes, is a collection of SCI's that are related to one another in a well-defined way. In early baselines and their associated updates, SCI's are specification documents (one or more volumes of text for each baseline or associated update); in later baselines and their associated updates, each SCI may manifest itself in any or all of the various software representations. Software configuration control focuses on managing changes to SCI's (existing or to be developed) in all of their representations. This process involves three basic ingredients.

1) Documentation (such as administrative forms and supporting technical and administrative material) for formally precipitating and defining a proposed change to a software system.

2) An organizational body for formally evaluating and

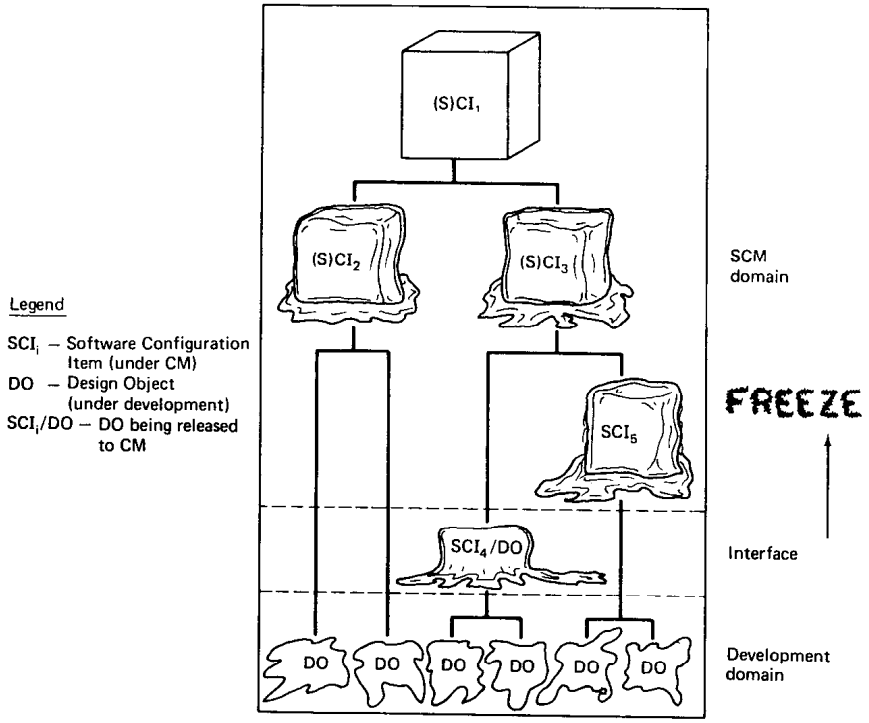

Fig. 3. The development/SCM interface.

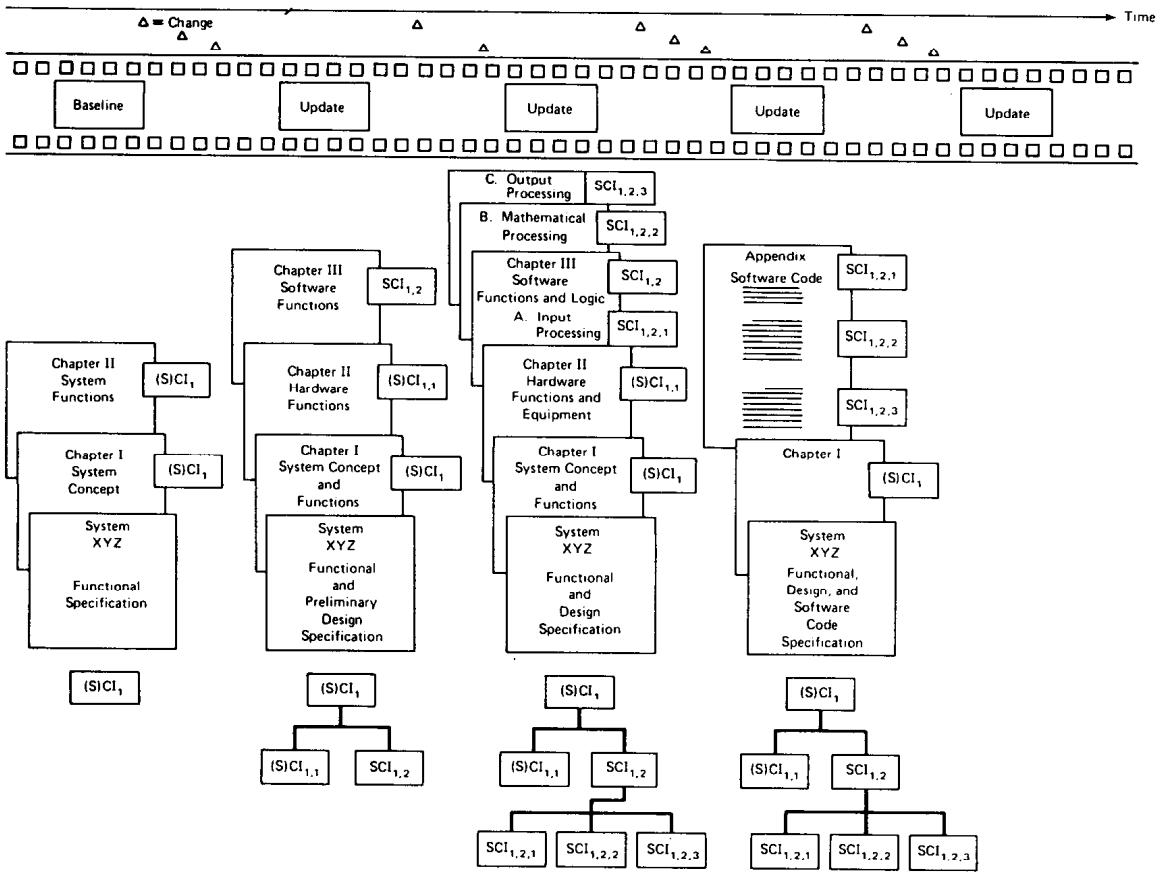

Fig. 4. SCI evolution in a single document.

approving or disapproving a proposed change to a software system (the Configuration Control Board).

3) Procedures for controlling changes to a software system.

The Engineering Change Proposal (ECP), a major control document, contains information such as a description of the proposed change, identification of the originating organization, rationale for the change, identification of affected baselines and SCI's (if appropriate), and specification of cost and schedule impacts. ECP's are reviewed and coordinated by the CCB, which is typically a body representing all organizational units which have a vested interest in proposed changes.

Fig. 5 depicts the software configuration control process.

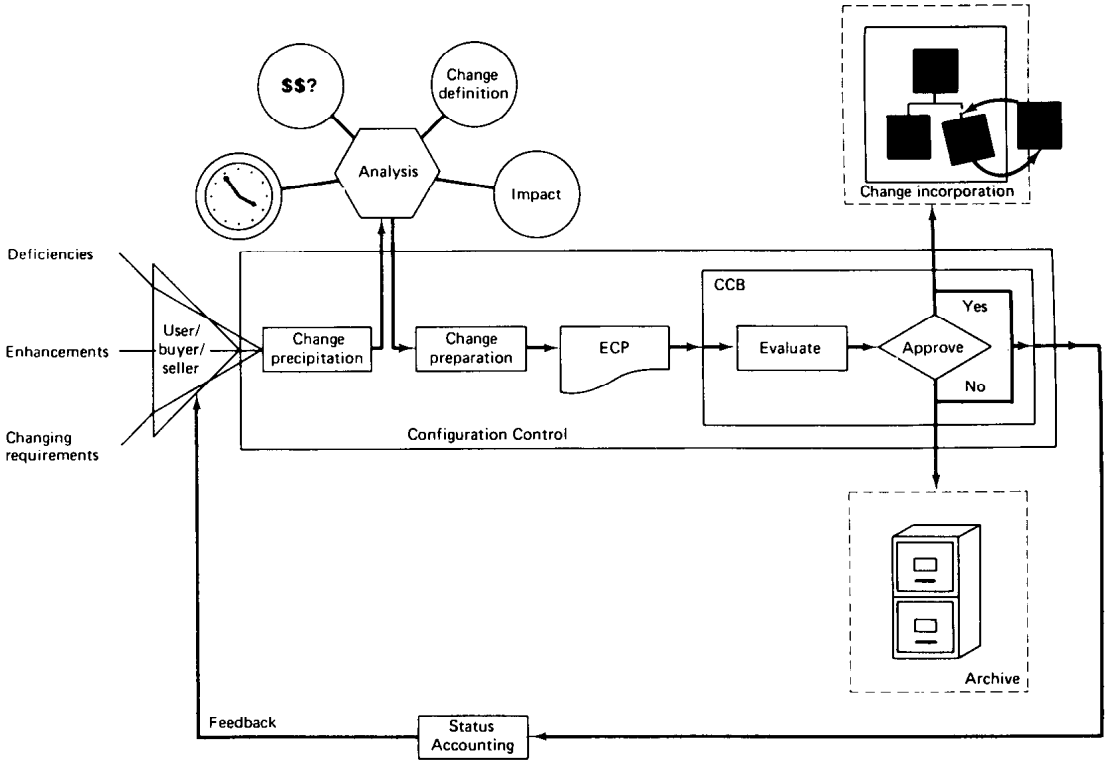

Fig. 5. The control process.

As the figure suggests, change incorporation is not an SCM function, but monitoring the change implementation process resulting in change incorporation is. Fig. 5 also emphasizes that the analysis that may be required to prepare an ECP is also outside the SCM purview. Note also from the figure how ECP's not approved by the CCB are not simply discarded but are archived for possible future reference.

Many automated tools support the control process. The major ones aid in controlling software change once the coding stage has been reached, and are generically referred to as program support libraries (PSL's). The level of support provided by PSL's, however, varies greatly. As a minimum, a PSL should provide a centralized and readily available repository for authoritative versions of each component of a software system. It should contain the data necessary for the orderly development and control of each SCI. Automation of other functions, such as library access control, software and document version maintenance, change recording, and document reconstruction, greatly enhance both the control and maintenance processes. These capabilities are currently available in systems such as SOFTOOL's change and configuration control environment (CCC).

A PSL supports a developmental approach in which project personnel work on a common visible product rather than on independent components. In those PSL's which include access controls, project personnel can be separately assigned read/write access to each software document/component, from programs to lines of code. Thus, all project personnel are assured ready access to the critical interface information necessary for effective software development. At the same time, modifications to various software components, whether sanctioned baselines or modules under development, can be closely controlled.

Under the PSL concept, the programmer operates under a well-defined set of parameters and exercises a narrower span of detailed control. This minimizes the need for explicit communication between analysts and programmers and makes the inclusion of new project personnel less traumatic since interface requirements are well documented. It also minimizes the preparation effort for technical audits.

Responsibility for maintenance of the PSL data varies depending on the level of automation provided. For those systems which provide only a repository for data, a secretary/librarian is usually responsible for maintaining the notebooks which will contain the data developed and used by project personnel and for maintenance of the PSL archives. More advanced PSL systems provide real time, on-line access to data and programs and automatically create the records necessary to fully trace the history of the development. In either case the PSL provides standardization of project recordkeeping, ensures that system documentation corresponds to the current system configuration, and guarantees the existence of adequate documentation of previous versions.

A PSL should support three main activities: code development, software management, and configuration control. Support to the development process includes support to design, coding, testing, documentation, and program maintenance along with associated database schema and subschema. A PSL provides this support through:

- storage and maintenance of software documentation and code,
- support to program compilation/testing,
- support for the generation of program/system documentation.

Support to the management of the software development process involves the storage and output of programming data such as:

- collection and automatic reporting of management data related to program development,

- control over the integrity and security of the data in the PSL,
- separation of the clerical activity related to the programming process.

PSL's provide support to the configuration control process through:

- access and change authorization control for all data in the library,
- control of software code releases,
- automatic program and document reconstruction,
- automatic change tracking and reporting,
- assurance of the consistency between documentation, code, and listings.

A PSL has four major components: internal libraries in machine-readable form, external libraries in hardcopy form, computer procedures, and office procedures. The components of a PSL system are interlocked to establish an exact correspondence between the internal units of code and external versions (such as listings) of the developing systems. This continuous correspondence is the characteristic of a PSL that guarantees ongoing visibility and identification of the developing system.

Different PSL implementations exist for various system environments with the specifics of the implementation dependent upon the hardware, software, user, and operating environment. The fundamental correspondence between the internal and external libraries in each environment, however, is established by the PSL librarian and computer procedures. The office procedures are specified in a project CM Plan so that the format of the external libraries is standard across software projects, and internal and external libraries are easily maintainable.

Newer PSL systems minimize the need for both office and computer procedures through the implementation of extensive management functionality. This functionality provides significant flexibility in controlling the access to data and allocating change authority, while providing a variety of status reporting capabilities. The availability of management information, such as a list of all the software structures changed to solve a particular Software Trouble Report or the details on the latest changes to a particular software document, provides a means for the control function to effectively operate without burdening the development team with cumbersome procedures and administrative paperwork. Current efforts in PSL refinement/development are aimed at linking support of the development environment with that of the configuration control environment. The goal of such systems is to provide an integrated environment where control and management information is generated automatically as a part of a fully supported design and development process.

Software Configuration Auditing: Software configuration auditing provides the mechanism for determining the degree to which the current state of the software system mirrors the software system pictured in baseline and requirements documentation. It also provides the mechanism for formally establishing a baseline. A baseline in its formative stages (for example, a draft specification document that appears prior to the existence of the functional baseline) is referred to as a "to-be-established" baseline; the final state of the auditing process conducted on a to-be-established baseline is a sanctioned baseline. The same may be said about baseline updates.

Software configuration auditing serves two purposes, configuration verification and configuration validation. Verification ensures that what is intended for each software configuration item as specified in one baseline or update is actually achieved in the succeeding baseline or update; validation ensures that the SCI configuration solves the right *problem* (i.e., that customer needs are satisfied). Software configuration auditing is applied to each baseline (and corresponding update) in its to-be-established state. An auditing process common to all baselines is the determination that an SCI structure exists and that its contents are based on all available information.

Software auditing is intended to increase software visibility and to establish traceability throughout the life cycle of the software product. Of course, this visibility and traceability are not achieved without cost. Software auditing costs time and money. But the judicious investment of time and money, particularly in the early stages of a project, pays dividends in the latter stages. These dividends include the avoidance of costly retrofits resulting from problems such as the sudden appearance of new requirements and the discovery of major design flaws. Conversely, failing to perform auditing, or constraining it to the later stages of the software life cycle, can jeopardize successful software development. Often in such cases, by the time discrepancies are discovered (if they are), the software cannot be easily or economically modified to rectify the discrepancies. The result is often a dissatisfied customer, large cost overruns, slipped schedules, or cancelled projects.

Software auditing makes visible to management the current status of the software in the life cycle product audited. It also reveals whether the project requirements are being satisfied and whether the intent of the preceding baseline has been fulfilled. With this visibility, project management can evaluate the integrity of the software product being developed, resolve issues that may have been raised by the audit, and correct defects in the development process. The visibility afforded by the software audit also provides a basis for the establishment of the audited life cycle product as a new baseline.

Software auditing provides traceability between a software life cycle product and the requirements for that product. Thus, as life cycle products are audited and baselines established, every requirement is traced successively from baseline to baseline. Disconnects are also made visible during the establishment of traceability. These disconnects include requirements not satisfied in the audited product and extraneous features observed in the product (i.e., features for which no stated requirement exists)

With the different point of view made possible by the visibility and traceability achieved in the software audit, management can make better decisions and exercise more incisive control over the software development process. The result of a software audit may be the establishment of a baseline, the redirection of project tasking, or an adjustment of applied project resources.

The responsibility for a successful software development project is shared by the buyer, seller, and user. Software auditing uniquely benefits each of these project participants. Appropriate auditing by each party provides checks and

balances over the development effort. The scope and depth of the audits undertaken by the three parties may vary greatly. However, the purposes of these differing forms of software audit remain the same: to provide visibility and to establish traceability of the software life cycle products. An excellent overview of the software audit process, from which some of the above discussion has been extracted, appears in [11].

Software Configuration Status Accounting: A decision to make a change is generally followed by a time delay before the change is actually made, and changes to baselines generally occur over a protracted period of time before they are incorporated into baselines as updates. A mechanism is therefore needed for maintaining a record of how the system has evolved and where the system is at any time relative to what appears in published baseline documentation and written agreements. Software configuration status accounting provides this mechanism. Status accounting is the administrative tracking and reporting of all software items formally identified and controlled. It also involves the maintenance of records to support software configuration auditing. Thus, software configuration status accounting records the activity associated with the other three SCM functions and therefore provides the means by which the history of the software system life cycle can be traced.

Although administrative in nature, status accounting is a function that increases in complexity as the system life cycle progresses because of the multiple software representations that emerge with later baselines. This complexity generally results in large amounts of data to be recorded and reported. In particular, the scope of software configuration status accounting encompasses the recording and reporting of:

1) the time at which each representation of a baseline and update came into being;
2) the time at which each software configuration item came into being;
3) descriptive information about each SCI;
4) engineering change proposal status (approved, disapproved, awaiting action);
5) descriptive information about each ECP;
6) change status;
7) descriptive information about each change;
8) status of technical and administrative documentation associated with a baseline or update (such as a plan prescribing tests to be performed on a baseline for updating purposes);
9) deficiencies in a to-be-established baseline uncovered during a configuration audit.

Software configuration status accounting, because of its large data input and output requirements, is generally supported in part by automated processes such as the PSL described earlier. Data are collected and organized for input to a computer and reports giving the status of entities are compiled and generated by the computer.

The Management Dilemma

As we mentioned at the beginning of this paper, SCM and many of the other product assurance disciplines grew up in the 1970's in response to software failure. The new disciplines were designed to achieve visibility into the software engineering process and thereby exercise some measure of control over that process. Students of mathematical control theory are taught early in their studies a simple example of the control process. Consider being confronted with a cup of hot coffee, filled to the top, which you are expected to carry from the kitchen counter to the kitchen table. It is easily verified that if you watch the cup as you carry it, you are likely to spill more coffee than if you were to keep your head turned away from the cup. The problem with looking at the cup is one of overcompensation. As you observe slight deviations from the straight-and-level, you adjust, but often you adjust too much. To compensate for that overadjustment, you tend to overadjust again, with the result being hot coffee on your floor.

This little diversion from our main topic of SCM has an obvious moral. There is a fundamental propensity on the part of the practitioners of the product assurance disciplines to overadjust, to overcompensate for the failures of the development disciplines. There is one sure way to eliminate failure completely from the software development process, and that is to stop it completely. The software project manager must learn how to apply his resources intelligently. He must achieve visibility and control, but he must not so encumber the developer so as to bring progress to a virtual halt. The product assurers have a virtuous perspective. They strive for perfection and point out when and where perfection has not been achieved. We seem to have a binary attitude about software; it is either correct or it is not. That is perhaps true, but we cannot expect anyone to deliver perfect software in any reasonable time period or for a reasonable sum of money. What we need to develop is software that is good enough. Some of the controls that we have placed on the developer have the deleterious effect of increasing costs and expanding schedules rather than shrinking them.

The dilemma to management is real. We must have the visibility and control that the product assurance disciplines have the capacity to provide. But we must be careful not to overcompensate and overcontrol. This is the fine line which will distinguish the successful software managers of the 1980's from the rest of the software engineering community.

Acknowledgment

The author wishes to acknowledge the contribution of B. J. Gregor to the preparation and critique of the final manuscript.

References

[1] "Contracting for computer software development—Serious problems require management attention to avoid wasting additional millions," General Accounting Office, Rep. FGMSD 80-4, Nov. 9, 1979.
[2] D. M. Weiss, "The MUDD report: A case study of Navy software development practices," Naval Res. Lab., Rep. 7909, May 21, 1975.
[3] B. W. Boehm, "Software engineering," *IEEE Trans. Comput.*, vol. C-25, pp. 1226–1241, Dec. 1976.
[4] *Proc. IEEE* (Special Issue on Software Engineering), vol. 68, Sept. 1980.
[5] E. Bersoff, V. Henderson, and S. Siegel, "Attaining software product integrity," *Tutorial: Software Configuration Management*, W. Bryan, C. Chadbourne, and S. Siegel, Eds., Los Alamitos, CA, IEEE Comput. Soc., Cat. EHO-169-3, 1981.
[6] B. W. Boehm *et al.*, *Characteristics of Software Quality, TRW Series of Software Technology*, vol. 1. New York: North-Holland, 1978.
[7] T. A. Thayer, *et al.*, *Software Reliability, TRW Series of Software Technology*, vol. 2. New York: North-Holland, 1978.

[8] D. J. Reifer, Ed., *Tutorial: Automated Tools for Software Eng.*, Los Alamitos, CA, IEEE Comput. Soc., Cat. EHO-169-3, 1979.
[9] E. Bersoff, V. Henderson, and S. Siegel, *Software Configuration Management*. Englewood Cliffs, NJ: Prentice-Hall, 1980.
[10] ——, "Software configuration management: A tutorial," *Computer*, vol. 12, pp. 6–14, Jan. 1979.
[11] W. Bryan, S. Siegel, and G. Whiteleather, "Auditing throughout the software life cycle: A primer." *Computer*, vol. 15, pp. 56–67, Mar. 1982.
[12] "Software configuration management," Naval Elec. Syst. Command, Software Management Guidebooks, vol. 2, undated.

Edward H. Bersoff (M'75–SM'78) received the A.B., M.S., and Ph.D. degrees in mathematics from New York University, New York.

He is President and Founder of BTG, Inc., a high technology, Washington, DC area based, systems analysis and engineering firm. In addition to his corporate responsibilities, he directs the company's research in software engineering, product assurance, and software management. BTG specializes in the application of modern systems engineering principles to the computer based system development process. At BTG, he has been actively involved in the FAA's Advanced Automation Program where he is focusing on software management and software configuration management issues on this extremely complex program. He also participates in the company's activities within the Naval Intelligence community, providing senior consulting services to a wide variety of system development efforts. He was previously President of CTEC, Inc. where he directed the concept formulation and development of the Navy Command and Control System (NCCS), Ocean Surveillance Information System (OSIS) Baseline now installed at all U.S. Navy Ocean Surveillance Centers. He also served as Experiment Director for the Joint ARPA, Navy, CINCPAC Military Message Experiment. This test was designed to examine the usefulness of secure, automated message processing systems in an operational military environment and to develop design criteria for future military message processing systems. Prior to joining CTEC, Inc., he was Manager of Engineering Operations and Manager of FAA Operations for Logicon, Inc.'s Process Systems Division. He joined Logicon from the NASA Electronics Research Center. He has taught mathematics at universities in Boston, New York, and Washington, DC. His technical contributions to the fields of software requirements and design range from early publications in computer architecture, reliability and programming languages, to more recent publications in software quality and configuration management. A textbook entitled *Software Configuration Management* (Prentice-Hall) represents the product of three years of research in the field by Dr. Bersoff and his colleagues.

Dr. Bersoff is a member of AFCEA, American Management Association, MENSA, and the Young Presidents' Organization.