Technical Report

CMU/SEI-90-TR-11
ESD-90-TR-212

Spectrum of Functionality in Configuration Management Systems

Susan Dart

December 1990

Technical Report
CMU/SEI-90-TR-11
ESD-90-TR-212
December 1990

Spectrum of Functionality in Configuration Management Systems

Susan Dart

Software Environments Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Spectrum of Functionality in Configuration Management Systems

Abstract: The Software Environments Project at the Software Engineering Institute has found considerable progress concerning support for software configuration management (CM) in environments and tools. This paper's intent is to highlight a spectrum of features provided by existing CM systems. The spectrum shows features as being extensions or generalizations of other features and these extensions represent the progress. As part of presenting the features, the scope of issues concerning users of CM systems is discussed. No single CM system provides all the functionality required by the different kinds of users of CM systems. Rather, each CM system addresses some part of a spectrum of functionality. To complete the report, several configuration management systems are briefly described.

1. Introduction

The Software Environments Project at the Software Engineering Institute has found considerable progress concerning software configuration management (CM) in environments and tools. This is evident from the spectrum of functionality provided by CM systems. The intention of this paper is to highlight that spectrum. To set the context for beginning such a discussion, this introduction provides some background information about CM by explaining it, the purpose of using it, and by giving a typical CM user scenario.

1.1. The Meaning of Configuration Management

Software CM is a discipline for controlling the evolution of software systems. Classic discussions about CM are given in texts such as [bersoff 80] and [babich 86]. A standard definition taken from IEEE standard 729-1983 [scm 87] highlights four classic operational aspects of CM:

1. Identification: an identification scheme is needed to reflect the structure of the product. This involves identifying the structure and kinds of components, making them unique and accessible in some form by giving each component a name, a version identification, and a configuration identification.

2. Control: controlling the release of a product and changes to it throughout the lifecycle by having controls in place that ensure consistent software via the creation of a baseline product.

3. Status Accounting: recording and reporting the status of components and change requests, and gathering vital statistics about components in the product.

4. Audit and review: validating the completeness of a product and maintaining consistency among the components by ensuring that components are in an appropriate state throughout the entire project life cycle and that the product is a well-defined collection of components.

The definition includes terminology such as configuration item, baseline, release and version, etc. At a high level, most designers of CM systems incorporate functionality of varying degrees to support these aspects. But at the implementation level, from the user's viewpoint, most CM systems have different functionality. Among the many reasons for these differences are: disparate hardware platforms, operating system, and programming languages. But an interesting reason is due to the different kinds of users of a CM system. This stems from the role the user plays in the organization. In particular, a manager, a software engineer developing an application, an application customer, and an environment builder tend to see CM differently. As a result, they may want differing (albeit complementary) functionality from their CM system. For example, to a manager the term "CM" conjures up the image of a Configuration Control Board. To a software engineer, the image of baselines arise. To a customer, versions of the application arise. And to the environment builder, mechanisms such as libraries and data compression arise. All these images obviously result in different requirements for a CM system and hence possibly different functionality. (Chapter 2 further discusses this issue and others about CM.)

1.2. The Purpose Of Using Configuration Management

CM is used for technical and contractual reasons. Technically, the purpose of using CM is to solve some of the problems pertaining to the evolution of a software product. These problems center around the lack of control and understanding of all the components that make up a software product, along with the complex co-ordination of the product's evolution over many years by many people. (See [bersoff 80] for examples of such problems.) Contractually, CM must be used by contractors developing an application for organizations such as the Department of Defense. That is, certain standards (such as [2167a 87]) must be used. This use effectively determines the nature of the product and some of the process used to develop and maintain it.

The goals of using CM are to ensure the integrity of a product and to make its evolution more manageable. Although there is overhead involved in using CM, it is generally agreed that the consequences of not using CM can lead to many problems and inefficiencies. The overhead of using CM relates to time, resources, and the effects on other aspects of the software lifecycle. For instance, CM affects many aspects in the decision making and risk assessment of a product. Decisions must be made about: how the code should evolve and at what price (in terms of time and work load the software engineers must pay); which CM system to use; and when to place the product under CM. All of these factors affect the software lifecycle. Hence, the less overhead in using a CM system, the better; in fact, the low overhead system is likely to be the system of choice. Not only should the upfront cost be low, but also the impact of the daily usage of the CM system should not be overwhelming to the users. Ideally a good CM system will be as supportive as possible and have an acceptable overhead. (Chapter 3 presents the kinds of functionality prevalent in existing CM systems.)

1.3. A Typical Configuration Management User Scenario

Before discussing CM systems, a simple, typical CM scenario of an organization is described in order to present a frame of reference.

A typical CM operational scenario involves a project manager who is in charge of a software group, a configuration manager who is in charge of the CM procedures and policies, the software engineers who are responsible for developing and maintaining the software product, and the customer who uses the product. In the scenario, assume that the product is a small one involving about 15,000 lines of code being developed by a team of 6 people. (Note that other scenarios of smaller or larger teams are possible but, in essence, there are generic issues that each of these projects face concerning CM.)

At the operational level, the scenario involves various roles and tasks. For the project manager, the goal is to ensure that the product is developed within a certain time frame. Hence, the manager monitors the progress of development and recognizes and reacts to problems. This is done by generating and analyzing reports about the status of the software system and by performing reviews on the system.

The goals of the configuration manager are to ensure that procedures and policies for creating, changing, and testing of code are followed, as well as to make information about the project accessible. To implement techniques for maintaining control over code changes, this manager introduces mechanisms for making official requests for changes, for evaluating them (via a Change Control Board that is responsible for approving changes to the software system), and for authorizing changes. The manager creates and disseminates task lists for the engineers and basically creates the project context. Also, the manager collects statistics about components in the software system, such as information determining which components in the system are problematic.

For the software engineers, the goal is to work effectively. This means engineers do not unnecessarily interfere with each other in the creation and testing of code and in the production of supporting documents. But, at the same time, they try to communicate and coordinate efficiently. Specifically, engineers use tools that help build a consistent software product. They communicate and coordinate by notifying one another about tasks required and tasks completed. Changes are propagated across each other’s work by merging files. Mechanisms exist to ensure that, for components which undergo simultaneous changes, there is some way of resolving conflicts and merging changes. A history is kept of the evolution of all components of the system along with a log with reasons for changes and a record of what actually changed. The engineers have their own workspace for creating, changing, testing, and integrating code. At a certain point, the code is made into a baseline from which further development continues and from which variants for other target machines are made.

The customer uses the product. Since the product is under CM control, the customer follows formal procedures for requesting changes and for indicating bugs in the product.

Ideally, a CM system used in this scenario should support all these roles and tasks; that is, the roles determine the functionality required of a CM system. The project manager sees CM as an auditing mechanism; the configuration manager sees it as a controlling, tracking, and policy making mechanism; the software engineer sees it as a changing, building, and access control mechanism; and the customer sees it as a quality assurance mechanism.

1.4. Terminology Clarification

As an aside, it is worthwhile clarifying one minor notion for this report, the notion of a CM *system* and a CM *tool*. A CM system is part of an environment where the CM support is an integral part of it and is sold in that manner as part of a package. For instance, the Rational [rational 88] environment has CM functionality that is an integral part of it. A CM tool is a stand-alone tool. For instance, the Revision Control System(RCS) [rcs 85]) is a CM tool since it is intended to be installed into an existing environment. Because the distinction is not important to this report, the term *CM system* will be used to represent both concepts.

As to what constitutes a CM system, there is no agreement. That is, there is no unified concept of a CM system. For instance, if a system has version control, is it a CM system? Generally speaking, any system that provides some form of version control, configuration identification, and system modeling and has the intent of providing CM is considered to be a CM system. But it should be noted that existing CM systems provide their own combination of functionality rather than a standard set.

1.5. The Contents of This Report

This introduction has given an explanation of CM, the purpose of using it, and an example of a typical CM scenario, thereby hinting at the typical kinds of CM users and their requirements for CM systems. Chapter 2 describes the scope of CM issues for users of CM systems. This chapter suggests a variety of issues that can affect users' requirements for a CM system. Chapter 3 highlights the features of existing CM systems in the form of a spectrum. Chapter 4 makes some observations about the future of CM systems and Chapter 5 gives a conclusion. Appendix A presents a short overview of the functionality of some existing CM systems.

2. Issues for Users of Configuration Management Systems

Many issues related to CM affect the user of a CM system. Existing CM systems address these issues in a variety of ways. Although the intent of this report is to discuss some of the features in existing CM systems, it is worthwhile presenting the issues because all affect the user's expectations for a CM system.

Figure 2-1 is a snapshot of the issues, which are:

- User roles: there are different kinds of users of CM systems and, consequently, different functionality requirements for CM systems.
- When to start using CM: the point at which a project group may start using a CM system depends on the capabilities of the CM system.
- Control Level: a CM system can impose different levels of control over the product and its management.
- Process and product: an ideal CM system provides for the CM process as well as for the product and its artifacts under CM.
- Automation level: fulfilling CM functions is generally a combination of using both manual procedures and an on-line CM system.
- Functionality: CM systems have features that implement a spectrum of CM functionality.

Figure 2-1 also shows that the issues are not mutually exclusive but are part of the whole environment arena. That is, all the issues are related to one another and to the environment of which the CM system is a part. This paper focuses only on the issues themselves rather than on the environment. The following sections discuss each issue in further detail.

2.1. Roles and Requirements

As indicated by the typical CM scenario of Section 1.3, there are different kinds of users of CM systems.[1] Each of these users can have a different view of CM and, hence, different expectations for a CM system. These expectations are distinct and generally complementary. Figure 2-2 highlights a set of functionality that the various users of a CM system expect. Each box in Figure 2-2 represents a major functionality area. These areas are:

[1]There are other kinds of roles pertinent to CM systems: the environment/tool builder and the environment/tool integrator. These roles are not strictly user roles in the sense that this paper presents. They are really related to developing a CM system for the above kinds of users.

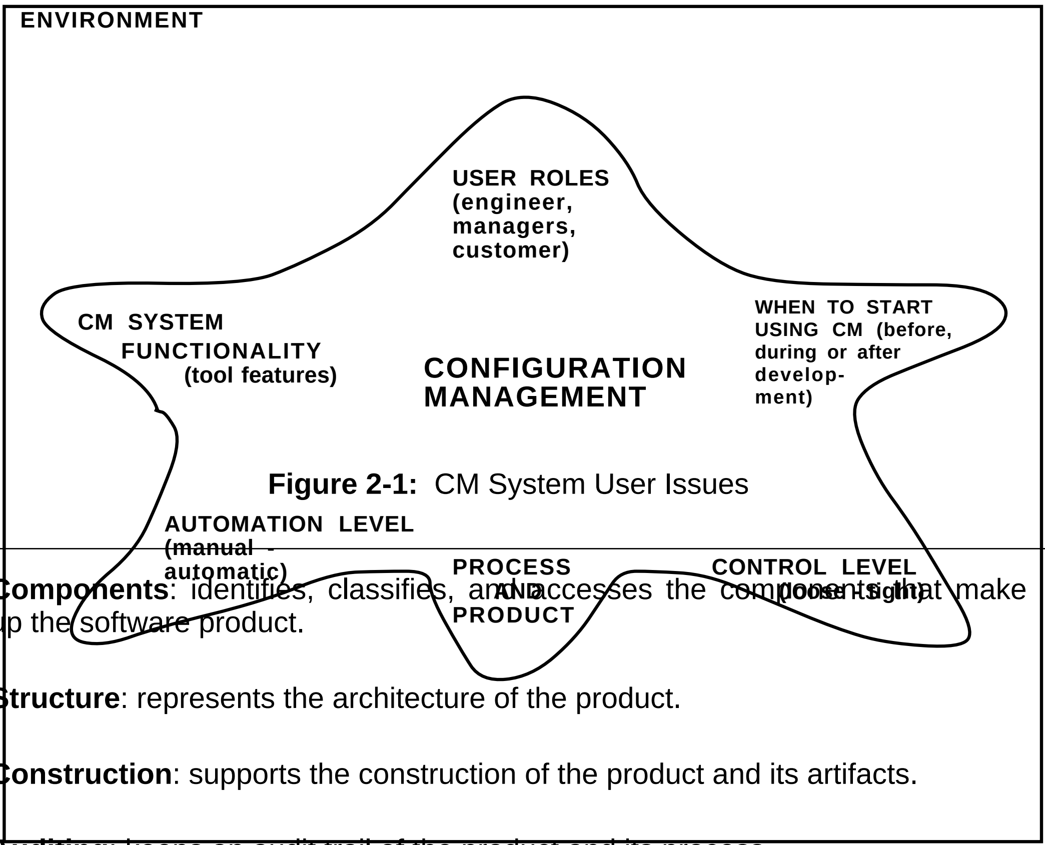
ENVIRONMENT
USER ROLES (engineer, managers, customer)
CM SYSTEM FUNCTIONALITY (tool features)
CONFIGURATION MANAGEMENT
WHEN TO START USING CM (before, during or after develop- ment)
AUTOMATION LEVEL (manual - automatic)
PROCESS AND PRODUCT
CONTROL LEVEL (loose - tight)

Figure 2-1: CM System User Issues

- **Components**: identifies, classifies, and accesses the components that make up the software product.
- **Structure**: represents the architecture of the product.
- **Construction**: supports the construction of the product and its artifacts.
- **Auditing**: keeps an audit trail of the product and its process.
- **Accounting**: gathers statistics about the product and the process.
- **Controlling**: controls how and when changes are made.

- **Process**: supports the management of how the product evolves.
- **Team**: enables a project team to develop and maintain a family of products.

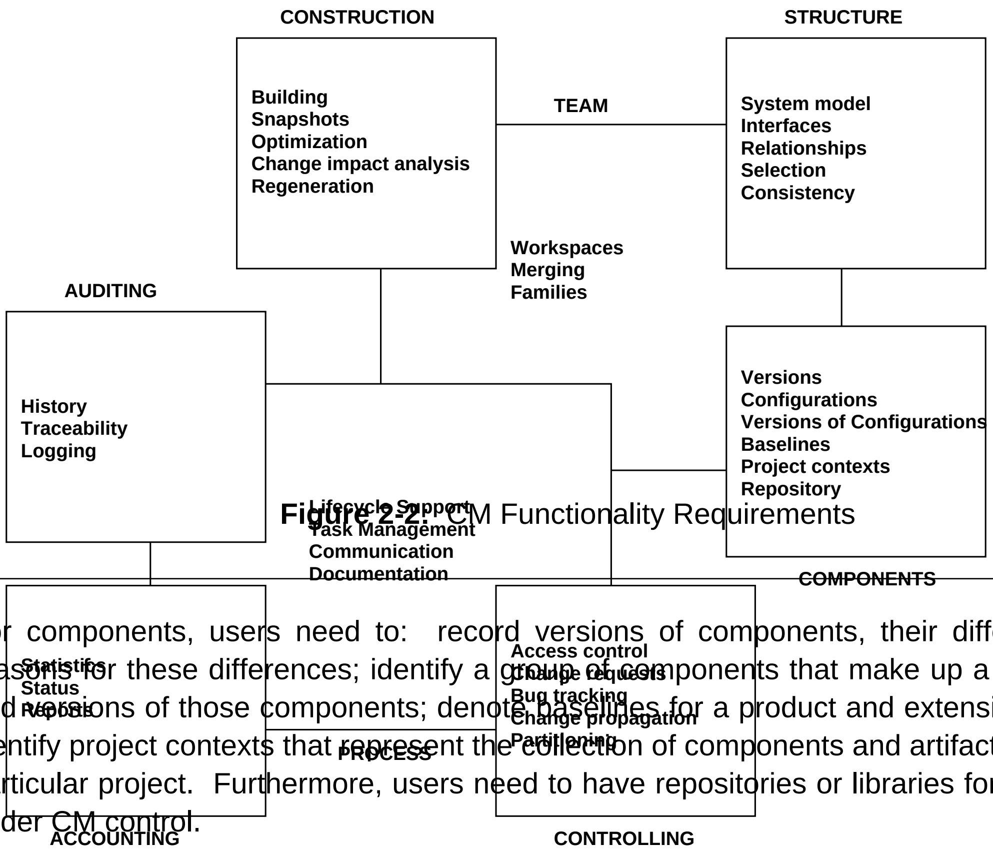

Figure 2-2: CM Functionality Requirements

For components, users need to: record versions of components, their differences, and reasons for these differences; identify a group of components that make up a configuration and versions of those components; denote baselines for a product and extensions to those; identify project contexts that represent the collection of components and artifacts related to a particular project. Furthermore, users need to have repositories or libraries for components under CM control.

For structure, users need to: model the structure of the product via a system model that represents the inventory of components for that product; specify interfaces among components, versions, and configurations, thereby making them reusable; identify and maintain relationships between components; and select compatible components to be made into a valid and consistent version of the product.

For construction, users need: means to easily construct or build the product; the ability to take a snapshot or freeze the status of the product at any time; mechanisms for optimizing efforts at constructing systems by reducing the need to recompile components; facilities for doing change impact analysis that indicates all the ramifications of making a change before the change is made; and easy regeneration of any phase or part of the product at any point in time.

For auditing, users require: a history of all changes; traceability between all related components in the product and their evolution; and a log of all the details of work done.

For accounting, users need: a mechanism to record statistics, to examine the status of a product, and to easily generate reports about all aspects of the product.

For controlling, users need: controlled access to components in the system to avoid any unwarranted changes or change conflicts; on-line support for change request forms and problem reports; means for tracking bugs and how, when, and by whom they are dealt with; propagation of changes in a controlled manner across different versions of the product; and a way of partitioning the product for limiting the effects of changes to it.

For process requirements, users need: support for their lifecycle model and their organization's policies; the ability to identify tasks to be done and how and when they are completed; the ability to communicate information to appropriate people about relevant events; and the facilities for documenting knowledge about the product.

For team requirements, users need: individual and group workspaces; the merging of multiple changes along with conflict analysis; and facilities for supporting the creation of a family of products.

Note that the process box and team box are seen as being the significant areas of functionality. This is because they affect, and are affected by, all the other areas. For any user, an ideal CM system would support all the areas of functionality with team and process support fully integrated with other areas. No existing system provides all the functionality for each area.

2.2. When to Start Using a Configuration Management System

It varies as to when project teams start using a CM system on the products they are developing and maintaining. Some teams choose to do so when the product has been through its development lifecycle and is ready for shipment to the customer site. On the other hand, others choose to put everything under CM from the initiation of a project. Both choices have their own overheads. For instance, a team may make the choice based on the overheads associated with asking for a change. That is, if there are a number of manual procedures (such as filing a change request form, seeking Change Control Board (CCB) approval, getting acknowledgment, etc.), a team opts for placing the software under CM control once the major part of development is complete. But if the change request procedure can be done on-line with little time and effort expended by the users, CM will be used at an earlier part of the lifecycle.

In theory, CM is applicable throughout the product's lifetime—from creation, development, product release, customer delivery, customer use, through maintenance. Ideally, CM systems should support this with minimum overhead possible, thereby allowing CM to be applied as early as possible on a project. Existing CM systems, however, tend to focus mostly on a particular phase of the lifecycle so users are limited by that.

2.3. Levels of Configuration Management Control

A number of procedures, policies and tools combine to assist in carrying out CM. They will provide varying degrees of control over the users and evolution of the product. For instance, they may require an engineer to submit a formal, written change request. This is followed by a Change Control Board (CCB) evaluation and authorization of a change. The configuration manager then sets up a workplace for the software engineer. Particular files are extracted by the CM manager from a guarded repository and placed in that workspace solely for that engineer. On the other hand, different procedures, policies and tools may actually allow the engineers to electronically mail their need for changes to the CM manager and other members of the CCB. The members mail their approval immediately. The change request is assigned to an engineer who extracts the pertinent files directly from a repository and makes the changes. All this is done without any manual intervention. Since the CM system would automatically log all accesses, an official record of the change process is created.

The first scenario can be considered to have tight, active control over any action, but the latter process has loose, passive control over actions. Frequent changes are discouraged in the first scenario because of all the manual overhead, whereas in the latter scenario frequent change is encouraged since it is easy to do. These different levels of control may be more appropriate at certain phases of the product's lifecycle, for example, the first one is suitable for maintenance but the second for development.

Whatever CM system is used, it will have a certain level of control over the timeliness of the product's evolution. Existing CM systems provide their own level of control and a few are flexible enough to allow the user to pick the kind of control.

2.4. Distinguishing Between Process and Product Support

CM involves a process and a product. A CM process represents the sequence of tasks needed to carry out CM. Essentially, the process is a plan that defines what needs to be done, who does it and how it is be carried out. Supporting the process is a management function rather than an engineering one. The process model takes into account policies and procedures of the organization and its software development lifecycle model. The CM product is the result of the process and represents an engineering task. A CM system needs to provide functionality for both aspects (as shown in Figure 2-2). Existing systems mainly support the product side of CM, but the trend is for systems to support the management side.

2.5. Amount of Configuration Management Automation

At present, CM is generally a combination of manual and automated procedures. It is possible to perform CM without any kind of on-line assistance. But that is recognized as being inefficient. The goal is to automate as many as possible of the non-creative parts of CM. For instance, written change request forms and the protocol of responding to them are generally documented in an organization's policy folder rather than captured and enforced on-line. But there are systems that can provide for change request forms, as well as electronic mail for shipping the forms and authorizing the changes between engineer and manager.

Each CM system automates some function of CM to a different degree. Users need to use manual procedures to make up the support that does not exist on-line. Most existing CM systems require tailoring with scripts to make them amenable to the project team.

2.6. Configuration Management System Functionality

CM systems exist that provide some of the required functionality for CM. The next chapter gives examples of some interesting features in CM systems. In relation to CM requirements as shown in Figure 2-2, existing CM systems provide support for all the functionality areas, but more in some areas than in others. The support is mostly for components, structure, status, auditing, construction, and accounting; controlling, process and team support, which are more management functions, have limited support.

The notions exist to support most of the CM requirements, but not all notions are in the same CM system, and perhaps not to the degree of thoroughness that users want. This is likely to improve with time as environment builders understand the needs of CM users and the capabilities of environment architectures.

3. Spectrum of Functionality in CM Systems

The previous chapter explained the breadth of issues concerning users of CM systems. This section gives details about specific features in CM systems. In particular, it examines features that support a few of the functionality areas identified in the previous chapter. The features can be viewed as a spectrum, with each feature building upon the other.

It should be noted that the systems and features discussed are meant to be representative of what exists, rather than a complete summary or evaluation of what exists. Features are taken (rather than generalized) from specific CM systems since there is no common terminology when dealing with automated CM functionality. That is, each CM system has its own semantics for its features.

The functionality areas of interest for the CM system features to be discussed concern component, process, team, and a combination of structure and construction features. The following sections discuss each feature in a simplified manner. Each section has its own picture of how the features build on each other and this leads into a final picture that ties all the features together. The topology in each picture represents features that build on one another or, in other words, features that are an extension or generalization of one another. The final picture (Figure 3-5) gives a summary of all the features and systems discussed. Note that the description of features is simplified in order to focus on a certain aspect. It is realized that this may not highlight the full capabilities of features nor of systems. But, for the sake of presenting a spectrum, simplification is used since most CM systems tend to overload the purpose of various features. A brief overview of each CM system referenced in this report is given in Appendix A.

3.1. Component Features

Figure 3-1 is a snapshot of some CM systems and CM features that cater to identifying and accessing components of a software product. In particular, Revision Control System (RCS) [rcs 85] provides a classic file repository feature and Design Management System (DMS) [dms 90] extends that notion by catering to distributed components. The features are described below.

3.1.1. Repository

RCS provides the notion of a repository for source code ASCII files. In effect, the repository is a centralized library of files and provides version control. Most CM systems use some kind of notion of a repository. Any component in the repository is considered to be under a form of CM. All the CM information about files and the content of the files are kept in the repository. Hence, any CM controls pertain to what is in the repository. Users access the repository to glean information about the files. To work on the file, users withdraw (that is, check out) a particular file into their working directory, perform any work on the file, and, at any point in time, replace it in the repository (that is, check it back in). Replacing the file means creating a new version of that file. A version number is automatically associated with

Figure 3-1: Component Features

each replaced file; consequently, users can withdraw any file with a particular version number at any time. So that users cannot simultaneously withdraw the same file and work on it, once the file is withdrawn, it is automatically locked until it is replaced.

The repository stores file history information which includes the different versions of the files, the reason for a change, who replaced that version of the file and when. Note that the complete code for the different versions is not stored. Rather, only the actual difference between each version is stored; this is known as the *delta*. This assists in space savings and access time to a particular version.

3.1.2. Distributed Components

DMS provides a repository that controls files distributed on different hardware platforms. The repository is centralized, but the data from the repository can be distributed. DMS is aware of the distribution and carries out its CM taking that into account, such as by providing some fault tolerance facilities along with the necessary translations of file formats. So, to the users, the distribution is transparent. Users carry out their work on the repository as though all the files were located on their own workstations. When they are ready to return the changed files to the main repository, they use the DMS facilities.

A team of users geographically dispersed can be working on the same configuration of files. Multiple copies of files can exist on different workstations. Any changes to files in the repository can result in the local copy on the distributed workstations being updated since the system knows where all the local copies are. In effect, distributed users have access to a centralized repository, and to them the CM facilities seem to span the network of heterogeneous workstations.

3.2. Process Features

Figure 3-2 is a snapshot of CM systems that incorporate process related features. The CM systems are Change and Configuration Control (CCC) [ccc 87], PowerFrame [powerframe 89] and ISTAR [ISTAR 88]. The features are described below and are an extension of the repository feature of a system such as RCS, which was described in the previous section.

Figure 3-2: Process Features

3.2.1. Domain

PowerFrame is a system designed for the computer-aided engineering/design (CAE/CAD) field and essentially shields its user from any notion of a file system and configuration management. Users see only their domain-specific world and any CM is transparent to them. All they may see includes the appropriate tools for a particular task, and certain forms of presentation such as a logic-schema or a layout design, data that are pertinent to a particular task, and the form of commands that are pertinent to that domain. The user does not have to worry about such tasks as version control or relationships between files, since the system handles that behind the scenes. The users' working setup looks just like their domain's working environment. In effect, the CM features are hidden under the guise of domain-specific concepts.

3.2.2. Lifecycle Phase

The Change and Configuration Control (CCC) system provides some notions for supporting a particular lifecycle model in the sense of supporting phases in the lifecycle. In particular, CCC separates out development, product testing and releasing a product. This separation allows different kinds of users such as software engineers and testers to work on apparently the same code at the same time. This is achieved by passing the code through to separate directories that represent each phase. Work can continue independently in each directory, but, in essence, progresses from phase to phase, from person to person. As the independent work is completed, all changes are merged into a final product in the repository, tested and approved. A new release can be created and the cycle can begin again. In effect, the lifecycle model is achieved via parallel activities on versions of configurations.

3.2.3. Change Requests

In CCC, a change-request is a documented request to make a change and is presented as an on-line form with fill-in-the-blank screen panels. The request is for changes to be made to a section of code or to many related parts of the product such as a configuration. The scope of the change is user-defined, but CCC keeps a record of the request and what is affected by the change. The request is assigned a unique name. Once the form is completed with details of all ramifications (such as which components will be affected), the form can be electronically mailed to the CM manager who can mail back an approval or rejection to the software engineer. If approved, engineers are assigned to make the change, and those engineers are given access to the repository based on the change request assignment. The change request is resolved by making the changes. Once the changes are tested and approved, the affected components can then be passed to the approved configuration. Change requests provide a history of all changes to the repository, a status report for changes in progress, and an audit trail of changes done and changes rejected.

3.2.4. Contracts

The ISTAR environment provides for modeling some parts of a software development process. It models information flow and the completion of tasks. It does this via contracts. Contracts incorporate information flow, roles, tasks and components of the product and are "exchanged". A contract is fulfilled by the "passing" of a particular version of the product (or part thereof). The components are passed to certain elements of the process model such as a person or, logically, to a new phase, that is, the repository of part of the product is passed on.

3.3. Structure and Construction Features

Figure 3-3 is a snapshot of several CM systems that incorporate features for describing structure of a product and for constructing the product. The systems are Aide-De-Camp (ADC) [adc 89], Cedar [cedar 83], Rational [rational 88], Domain Software Engineering Environment (DSEE) [dsee 85], Adele [adele 85], and Configuration Management Assistant (CMA) [cma 89]. The features are described below. They are an extension of the repository notion of a system such as RCS (which was discussed in Section 3.1.1).

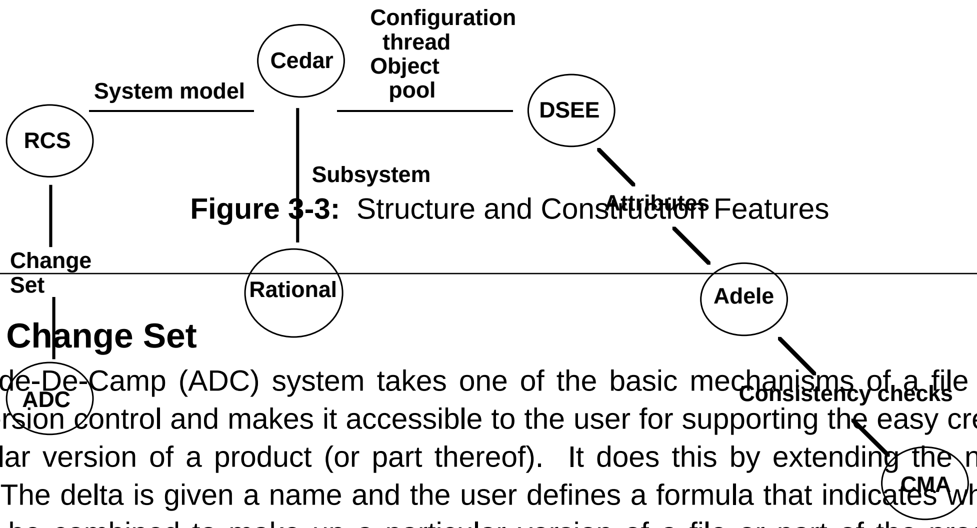

Figure 3-3: Structure and Construction Features

3.3.1. Change Set

The Aide-De-Camp (ADC) system takes one of the basic mechanisms of a file repository with version control and makes it accessible to the user for supporting the easy creation of a particular version of a product (or part thereof). It does this by extending the notion of a delta. The delta is given a name and the user defines a formula that indicates which deltas should be combined to make up a particular version of a file or part of the product. The combination of the deltas and the files to which they are applied and the reasons for each delta make up the *change set*. That is, the user can designate a change set that represents all the changes to a group of files (such as a configuration) and the reasons for each change (which could be the same across the whole group); ADC keeps track as to which delta is associated with which file in the group. The user can then treat this group as a whole and perform operations on the group. At any time, the user can create a version of a product by defining a base version plus a combination of change sets. The change sets reflect an audit trail of changes to a configuration of files as well as a means of accessing any version of a configuration that is not dependent on the latest version of that configuration.

3.3.2. System Model

The Cedar System Modeller was one of the first systems to explicitly define the structure of a configuration, i.e., a set of related components that make up the product under development. The System Modeller tracks the changes to components and controls the compiling and loading of configurations. The system model makes it clear to the environment what components are relevant and what relationships they have. It has a readable textual representation that can be edited by a user at any time and can be used by tools such as browsers, debuggers, and inter-module analyzers. In fact, the environment can use this model to perform some analysis on the components and make sure that the product is in a sensible state. For instance, a user changing a file may invalidate several other related files. The System Modeller recognizes this automatically and can automatically recompile the related (out-of-date) files, thereby creating a new latest version of the product. By defining the static structure of a product, the System Modeller allows the environment to assist in maintaining the integrity of the product.

3.3.3. Subsystems

The Rational environment adds to the notion of a system model and repository by incorporating features for dealing with a very large product. Rational provides for partitioning a large Ada product into parts, allowing for confining the scope of the effects of changes. The parts are called *subsystems*. Subsystems have interface specifications and represent a configuration item; therefore, they can be treated as a whole. Engineers can work on a subsystem making changes and as long as none of the interface specification is changed, any recompilations will happen only to components within that subsystem. Changes to the interface will possibly require the whole product to be recompiled. Subsystems have version control on their components, and subsystems themselves can be of a particular version. Subsystems thus minimize the need for recompiling the complete product. Also, due to the interface specification, the user can mix-and-match subsystems to make up a particular version of the product. The Rational environment checks that only compatible versions of subsystems are combined. Subsystems represent a way for having users limit the scope of their changes and for the environment to check the validity of combined configurations.

3.3.4. Configuration Thread

The Domain Software Engineering Environment (DSEE) takes the notion of a system model and enhances it by incorporating derived objects (i.e., object files as well as source files) and tools that do the derivation to the model. This allows the user to capture more precisely the static components involved in building a product. These components include: the versions of tools, the parameters used to invoke the tools, and the versions of components, along with other specific parameters that are worthwhile recording. As the product is compiled, DSEE automatically captures all the time and the versions of the components used and keeps this information in the repository in an object known as the *configuration thread*. Each configuration thread is given a unique identifier by DSEE. This identifier can then be used by DSEE as the permanent, all-encompassing history of the components necessary to build the product. At any point in time, the user can simply request that a particular version of the product be regenerated. Users no longer need to keep multiple copies of the same version of files and tools and any intermediate files in some central location: DSEE has a complete inventory of all components, their versions and their timestamps, thereby reducing the need for multiple copies and, hence, wasted space. Much of the management of derived objects as well as source objects is under the control of the CM system.

3.3.5. Object Pool

Using its notions of system model and configuration thread, DSEE has all the necessary information to recognize what is required to generate a particular version of a derived object. By associating this information with each derived object, all the objects can be put into a pool. Thus, whenever a user needs a particular derived object (or a compatible one), DSEE can automatically check whether one already exists in the pool, thereby obviating the need for regenerating the object. The user need not know that that derived object exists: DSEE does all the checking based on both the generation information the user gives and on existing configuration threads. This saves on the amount of compiling time and space required, and reuses work already done. In effect, the CM system has knowledge of how to optimally and exactly regenerate a particular version of the product.

3.3.6. Attributes

The Adele System generalizes upon the repository, system model and configuration thread notions by using a database and data modeling capabilities. Thus the repository becomes more object oriented and the user can describe a product in terms of a data model. Essentially, the components of a product are represented as database objects with attributes and relationships. Attributes are user-defined information about an object, for example, a certain object is a debugging object and its target system is VMS. Relationships define dependencies between objects, such as object A depends on object B. Hence, the user can describe a configuration in terms of characteristics of objects, rather than in terms of a composition of specific versions of objects. That is, by using selection rules and constraints centered around the attributes, the system can compose an appropriate configuration. The user can define any structure to a product (rather than just a hierarchical structure) and can designate a product in terms of desired characteristics. Thus, the user can describe a product at a higher level of abstraction (instead of in terms of a long list of instances of files).

3.3.7. Consistency Checking

The Configuration Management Assistant (CMA) enhances data modeling for CM with the addition of certain classes of attributes and relationships. Based on the meaning of the attributes and relationships, CMA can check to determine whether a configuration (which is a set of instances of objects) is valid. To be valid means that all the objects in the configuration make up a workable set; they are consistent with each other. It is the semantics of the attributes and relationships that permits CMA to determine the consistency. The classes of attributes represent user-defined characteristics including constraints, typing, and versions of configurations. Classes of relationships represent logical dependency, consistency, compatibility, component, instance, and inheritable dependency. These allow CMA to ensure that all instances of objects have compatible attributes and relationships in the formation of a configuration. In particular, CMA checks to determine whether configurations are complete, consistent, unambiguous, and have no version skews. With the formation of a configuration, CMA can add that configuration's characteristics to its database. Then, for subsequent use of that configuration or parts of it, CMA has enough information to check their consistency. The user can rely on the system to identify any inconsistencies and to preserve consistencies in creating and re-using configurations.

3.4. Team Features

Figure 3-4 is a snapshot of CM systems that support teams of software engineers working on a product. The systems are shape [shape 88], Software Management System (SMS) [SMS 88] and Network Software Environment (NSE) [nse 89]. The features which are described below represent extensions to the structure features of the Domain Software Engineering Environment (DSEE), which were described in the previous section.

Figure 3-4: Team Features

3.4.1. Workspace

The shape System combines the notions of a repository, system model, configuration thread and object pool, and adds the notion of a workspace. Workspaces pertain to each user and are designed to prevent users from interfering with one another's work. The user designates a workspace and imports files from the central repository. The workspace is intended to represent a certain status of a configuration (i.e., a certain version of part of the product). All work such as editing and compiling can be done in that workspace without affecting anything being done by other users in their workspaces. All by-products of the work remain in the workspace without affecting the central repository. When the user is finished with changes to a configuration, it is exported back into the central repository (after being approved).

The workspace is more than just a directory since version control capabilities are provided within the workspace, rather than just at the global level of the public repository. Also, only the user or designated users associated with that workspace can access it. A configuration can be accessed only from within a workspace, rather than from any directory. The CM system provides a team with more prudent access to parts of the products and prevents them from accidentally interfering with each other.

3.4.2. Transparent View

The Software Management System (SMS) enhances the notion of a workspace with a transparent view repository. This means that only the versions of the files in which the user is interested will be seen in the workspace; that is, version selection occurs transparently. All other versions are hidden from view (although they physically exist). The workspace then gives the appearance of a specialized repository for the user. Files that are checked out from the main repository are checked out to the workspace, and users are assigned access to that workspace. Files in the workspace effectively belong to that workspace instead of to a user. A history of all the changes made in the workspace is captured. The user can take a snapshot of versions of the files local to that workspace. Naming of files is local to the workspace rather than globally unique. The transparent view acts as a local repository and as a protection mechanism against information overload and unauthorized access to non-local versions of files.

3.4.3. Transaction

The transaction notion of NSE is an enhancement of the workspace and transparent view notions. It can represent a unit of work and supports: isolation of work between users; interactions between users; merging of changes based on the structure of the product. The transaction consists of an *environment* and a set of commands. A transparent view and workspace is provided by the environment. Commands between environments provide the interactions. The commands represent a protocol used to coordinate the actions between users and to represent the communication of the actual changes. Users work independently in their environments, changing the same or different parts of the product. The engineer can update the repository with a new version of a file changed in the environment. NSE assists in merging the changes into the repository. But it checks that what exists currently in the repository (which could have been placed there by some other engineer) does not conflict with the new changes coming. If there is a conflict, NSE notifies the engineer about the merging problems and provides assistance in eliminating the conflicts.

Just as users can update a repository by merging changes from the workspace into it, users can request that any changes made by another user (who has already updated the repository) be incorporated into the users' local workspaces. Thus, the transaction supports teams changing the same or related parts of the product.

3.5. Summary of the Spectrum of Functionality

Figure 3-5 represents a summary of the features discussed, along with their representative system. The topology of the spectrum is intended to show that the features build upon one another. That is, features are extensions or generalizations of one another. Figure 3-5 shows the following: given a basic file repository, distributed components are an enhancement for supporting component functionality; change requests, lifecycle phase, domain and contracts are enhancements for supporting process functionality; and a system model, change set, subsystem, configuration thread, object pool, attributes and consistency checks are enhancements for supporting structure and construction functionality. A workspace, transparent view, and transaction represent enhancements for supporting team functionality.

Probably the most difficult CM functionality concerns support for multiple engineers creating and maintaining a large product. Therefore, the spectrum highlights mostly features to support the software engineer. Some features support project management functions, but most of the progress is oriented towards team support and a simple form of process support.

Only a few CM systems are shown in Figure 3-5. There are many more with very similar functionality. Yet no single CM provides all the features presented in the spectrum of functionality. The "arms" of the spectrum in Figure 3-5 indicate that progress occurs in varying areas of CM.

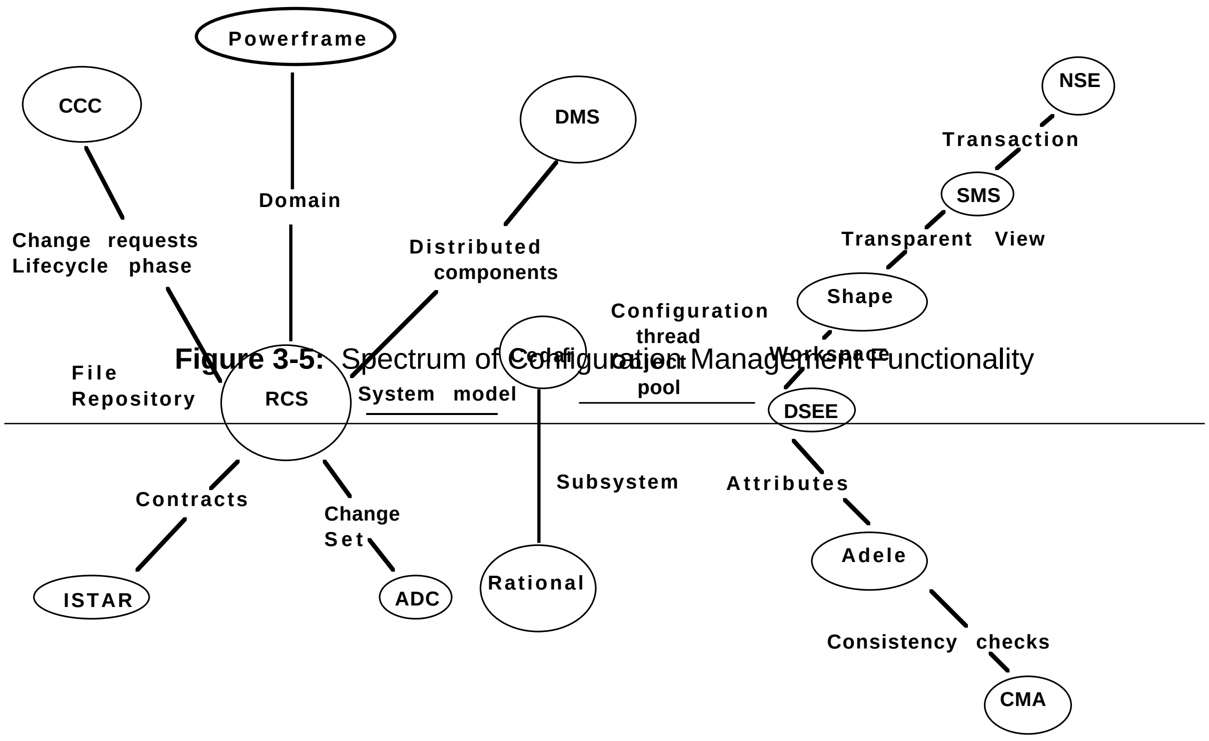

Figure 3-5: Spectrum of Configuration Management Functionality

4. The Future of CM Systems

The spectrum of functionality in Figure 3-5 is intended to be a high-level snapshot of state of the art in CM features. Several "arms" in the spectrum indicate possibly different CM areas, but it is envisioned that, as experience in CM systems is gained, these "arms" will join. This means that there is probably a fundamental CM model that encapsulates all the functionality presented in the spectrum. Further analysis work is needed to validate this probability.

But regardless of whether every CM system designer is trying to implement the same features, there are political and technical issues that do affect the future of CM systems. (Political issues relate to marketing and standardization; technical issues concern the feasibility of implementing certain mechanisms.)

A major political issue concerns the evolution of Computer Aided Software Engineering (CASE) tools. For instance, should CASE tool vendors ignore CM within their tools and assume that environment vendors will provide the CM support in their frameworks? Or should CASE tools builders provide CM support in their tools? If CASE vendors incorporate their own CM support, users will have to solve the problem of integrating different CM systems when they install their (different) CASE tools. Also, from the vendors' viewpoint, will they essentially be duplicating much of the work that has already been done or is now being attempted for environment frameworks?

On the other hand, if CASE vendors do not incorporate CM into their tools, can they rely on environment architects to provide a suitable framework to integrate CASE tools and simultaneously provide some sort of global CM capability? The answers to these questions are not known. In either case, there is the implication that some kind of standardization would probably be needed for CM systems in relation to environments, or vice versa.

Many technical research issues will affect the capabilities of CM systems. They include:

- What is the appropriate technology on which to base a CM system? Is an object-oriented database with persistency notions for immutable objects the most suitable?

- In what layer of an environment's architecture does CM fit? Should it be at the base level in the database, making it an integral part of an environment framework? Or is it all a matter of specifying CM as a process at a higher level in the architecture?

- Can the mechanisms for CM be separated from all the CM functionality, that is, are there "standard" CM primitives that could be used in any environment to support all the CM functionality? Is there a common CM model?

- Is it possible to provide distributed CM support? Can geographically dispersed software teams use the same CM system for local CM and for system integration? This is a major problem in industry, particularly for Department of Defense contractors.

- Is it possible to support cross-development of software? Can engineers developing a product on a host machine easily move it to the target machine, while still maintaining CM control over the product?

- Is scale a limiting factor for CM systems? Is the CM support for a million line product the same as that for a 100 million line product?

- Is it possible to model all aspects of the CM process, including the people-intensive parts, into a CM system?

Answers to the above questions are not yet obvious. It is likely that progress will come from various sources—from CM system vendors, environment architects, tool integrators, the software process modeling forum, and particularly from the computer-aided design/engineering (CAD/CAE) and computer-integrated manufacturing (CIM) worlds.

5. Conclusions

CM is management of the evolution of a software product. It is one area in software engineering environments where progress has been made. That is evident from the spectrum of functionality, as well as the number of existing CM systems and their capabilities. The CM spectrum shown in this report represents a snapshot of many features implemented by various CM systems. The spectrum is intended to indicate progress by showing that a number of features are extensions of other features. It is hoped that presenting the spectrum of functionality may aid in understanding state-of-the-art in CM systems and in providing a common framework for discussing CM.

Users of CM systems need to be aware of various issues that will affect their expectations of their CM system. The issues pertain to: the effect of user roles and their subsequent requirements on CM systems, when to start using a CM system; the levels of control incorporated into the system; support for management of both the CM process as well as the product under CM; how much of their CM functions can be automated; and the kind of functionality in the CM system. Existing CM systems differ as to how these issues are addressed.

No single system provides all the spectrum of functionality to suit different users. Overall, CM systems provide features for identifying components, designating structure, assisting in the construction of the product, auditing, accounting, controlling, process, and team support. Although the depth of the support may not be ideal, existing CM systems solve many of the CM problems.

Acknowledgements

Many thanks to the reviewers of this paper, in particular, Peter Feiler, Grace Downey, Kurt Wallnau, Ed Morris, Bob Ellison, Chuck Weinstock, Mario Barbacci, Rick Green and Jim Tomayko, and to technical writer Marie Elm.

References

[2167a 87]	U. S. Department of Defense. *DoD 2167a Standard* 1987. DoD.
[adc 89]	*Aide-De-Camp Software Management System, Product Overview* Concord, MA, 1989.
[adele 85]	Estublier, J. A Configuration Manager: The Adele Data Base of Programs. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 140-147. June 1985.
[babich 86]	Babich, W. *Software Configuration Management.* Addison-Wesley, 1986.
[bersoff 80]	Bersoff, E. H., Henderson, V.D., and Siegel, S. G. *Software Configuration Management.* Prentice-Hall, 1980.
[ccc 87]	Softool. *CCC: Change and Configuration Control Environment. A Functional Overview* 1987.
[cedar 83]	Lampson, B. and Schmidt, E. Practical Use of a Polymorphic Applicative Language. In *POPL*. ACM, 1983.
[cma 89]	Ploedereder, E. and Fergany, A. A Configuration Management Assistant. In *Proceedings of the Second International Workshop on Software Version and Configuration Control*, pages 5-14. ACM, USA, October 1989.
[dms 90]	Deitz. Daniel. Pulling the Data Together. *Mechanical Engineering* (), February 1990.
[dsee 85]	Leblang, David B. and McLean, Gordon D., Jr. Configuration Management for Large-Scale Software Development Efforts. In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June 1985.
[ISTAR 88]	Graham, Marc and Miller, Dan. *ISTAR Evaluation*. Technical Report CMU/SEI-88-TR-3, ADA201345, Software Engineering Institute, July 1988.

[nse 89] Courington, W.
The Network Software Environment.
Technical Report Sun FE197-0, Sun Microsystems Inc., February 1989.

[powerframe 89] Johnson, W.
Bringing Design Management to the Open Environment.
In *High Performance Systems*, pages 66-70. June 1989.

[rational 88] Feiler, Peter H., Dart, Susan A., and Downey, Grace.
Evaluation of the Rational Environment.
Technical Report CMU/SEI-88-TR-15, ADA198934, Software Engineering Institute, Carnegie Mellon University, July 1988.

[rcs 85] Tichy, Walter F.
RCS: A System for Version Control.
In *Software: Practice and Experience*, pages 637-654. July 1985.

[scm 87] *IEEE Guide to Software Configuration Management*
1987.
IEEE/ANSI Standard 1042-1987.

[shape 88] Mahler, Axel and Lampen, Andreas.
shape—A Software Configuration Management Tool.
In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 228-243. Siemens Germany, January 1988.

[SMS 88] Cohen, Ellis et al.
Version Management in Gypsy.
In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 210-215. November 1988.

[Taborda 90] Taborda, L. J. M.
Development of a Configuration Management Toolset.
In *Proceedings of the Fifth Australian Software Engineering Conference*. ACS, May 1990.

Appendix A: Overview of CM Systems

This appendix gives a snapshot of the capabilities of the various CM systems mentioned in the earlier chapters of this report. Neither an evaluation nor a complete description is given of the systems. Rather, the intent is to give the reader some information about the CM capabilities of the following systems.

- Adele
- Aide-De-Camp (ADC)
- Cedar
- Change and Configuration Control (CCC)
- Configuration Management Assistant (CMA)
- Design Management System (DMS)
- Domain Software Engineering Environment (DSEE)
- ISTAR
- Network Software Environment (NSE)
- PowerFrame
- Rational
- Revision Control System (RCS)
- shape
- Software Management System (SMS)

A.1. Aide-De-Camp (ADC)

Aide-De-Camp (ADC), from Software Maintenance and Development Systems, Inc., consists of the basic ADC system and a turnkey system that enhances ADC with a configuration notion, trouble reports, work lists and workspaces.

Basic ADC provides change sets for distribution of change. Essentially, the user groups deltas together to reflect associated changes in a product. The change set can be applied to another variant. The change set can represent a change to a file or a change to a configuration.

ADC provides a repository approach to version control and has database facilities with attributes, browsing, and relationships. Users decide whether to use one ADC database or several.

The turnkey system integrates problem reports and change requests with the assigning of work orders and setting up of a local workspace ("cleanroom"). This means that when a change request is sent to the CM manager and approved, the manager assigns a work item to a software engineer. When the engineer activates that work item, the workspace is created. As soon as the engineer is finished with that work item, the workspace is automatically deleted. The workspace is a set of directories and files; essentially, it is a local copy of the files in a configuration. Upon leaving the cleanroom, all the local copies are deleted and changes are added back to the main database. There is the notion of a configuration manager; this person assigns the work tasks. A workorder is assigned to a particular change request.

A Software Problem Report (SPR) or change request is a logged description of a bug or additional functionality being requested; the SPR documents why a change is made and must be specified before a work order can be written against it.

Basic ADC provides a list processing language that effectively allows the user to work on one file, a group of files, or a configuration. Linear and parallel versions of files can be created by the use of change sets. ADC supports merges independent of the latest version.

A.2. Adele

Adele is a research configuration manager system from the University of Grenoble. Its basic features are data modeling, interface checking and representing families of products. Adele represents more of a CM system kernel which is used as a basis for a CM system.

Adele’s attributed database allows for defining of objects which can be interfaces and their realizations (instances of bodies), configurations and families. Objects have attributes that describe their characteristics and relationships that describe object dependencies. Since the system knows of the dependency graph, it can assist in composing a configuration. The user can designate a configuration based upon desirable attributes. Attributes can be user-defined and system-defined. The user can designate selection rules based on attribute values, constraints and preferences. Adele can detect incomplete and inconsistent configuration descriptions.

A.3. Change and Configuration Control (CCC)

Softool’s Change and Configuration Control Tool (CCC) provides a repository and a turnkey system. CCC provides version control with certain conventions that suit the waterfall lifecycle model. Its CM tool supports 2167a documentation standards and change request forms on-line in the database. The native CCC includes a command language consisting of macros, and a facility to call operating system procedures for tailoring version control.

An application can be a baseline that is split into four parallel versions: production, development, test/integration and approved. Components are passed among these after certain approval such as change request completed.

Five classes of users form a hierarchy of access rights to information in the database. They are the database administrator, the CM manager, the project manager, the developer, and the test manager. Several levels of access controls exist such as access based on passwords, user classes and specific data item or change request assignments.

The CCC database hierarchy, which represents the product’s structure, comprises multiple levels of data structures including the database, a system, a configuration, a module, and text.

When a change is made, CCC records time, who, what changed and why it changed, and assigns a name to that change. CCC allows several change management strategies: changes may be recorded (1) as they occur, (2) on a regular basis during development and maintenance, or (3) at baseline release.

Parallel versions of code can be used for simultaneous development via virtual copies. These can be merged or selected and changes can be applied across configurations. Conflicts will be detected during a merge.

The native CCC has a macro facility and a build facility. The build facility allows change tracking, routing of forms, and notification to personnel when specific predefined events occur. Users can define a depends-on relationship. Change impact analysis is provided.

A.4. Cedar

The System Modeller for Cedar was part of the research work done at Xerox. The system model is a description of how to compose a set of related product configurations from their components. Users define various system models that make up the product. The System Modeller automates the program development cycle by tracking the changes to components (in this case modules) and controlling the compiling and loading of configurations.

A.5. Configuration Management Assistant (CMA)

The Configuration Management Assistant (CMA) from Tartan Laboratories is a mechanism for creating a CM system. The mechanism involves a very general data model with classes of attributes and relationships. Each class has its own semantics. The classes of attributes are partition, rendition, and version, and the classes of relationships are logical dependency, consistency, compatibility, component, instance, and inheritable dependency. CMA provides for recording and retrieving descriptions of configurations and the set of entities which comprise, record, and retrieve information about known (in)consistencies and dependencies between entities in a configuration, and predict the completeness and consistency of newly formed configurations. The database acts as a centralized repository and is subject to access control restrictions. Any change to the database occurs via the commitment of a simple "transaction." Each configuration can have its own access control mechanisms. Name clashes between configurations are avoided through use of name spaces.

A.6. Database Management System (DMS)

The Design Management System (DMS) from the Sherpa Corporation is geared for the computer-aided design/engineering market.

DMS provides a centralized repository with transparent distributed data base handling for files. Files can contain any kind of information, for example, ASCII, graphics, design data. User-defined monitors can be defined on repository events. DMS access control is based on class of user and promotion level of file, and file names can be encrypted. Versions are maintained via VMS file versioning scheme. All information (product structure, release procedures, alerts on events) is centralized and a hierarchy structure is assumed. DMS has a kernel database facility with user-defined attributes and relationships. Each modification to a structure results in a new version of all components in it. There is no concept of a "release," although it is implicit as part of a release procedure via promotion levels (promotion levels represent stages through which a project passes). The user incorporates review stages, who has access to what kind of data, the grouping of data, who should be notified of status changes and what approvals and checks are required for sign-off and promotion. DMS allows for promotion levels that represent company policy for approval or sign-off on files.

For distributed files, automatic update synchronization is carried out. Changes can be communicated to team members. The latest version of a file can be located regardless of where it resides in the network. Access control can be defined on files. Virtual teams can be defined(these are users that are geographically dispersed but can share a common database).

Product structure which DMS checks against can be defined and versions of this structure can be saved. Alerts can be triggered based on certain events. Status information can be provided and user-defined reports generated. Change requests are captured on-line and associated documents attached to them. The reviewers for the requests can electronically approve them. Status reports on change requests can be determined. An audit trail is kept.

A.7. Domain Software Engineering Environment (DSEE)

Apollo’s Domain Software Engineering Environment (DSEE) provides derived object code management as well as source version control, system modeling, configuration threads, version selection based on attributes, releases of configurations, system building, (reusable) object pools, task lists for tracking tasks to be done and those completed, and alerts for notifying users of certain events.

DSEE provides version control (using deltas) on a repository of source files. Check out and check in, along with logging facilities, are used for files. Using the system model and configuration threads, derived objects are identified and managed.

A DSEE system model represents a description of a product (or part thereof). DSEE is a declarative description defining static and structural properties. It is source-oriented and includes build rules and dependencies for each component. The model is a build-oriented description. It incorporates result and tool dependencies, the hierarchy of components, the build rules (shell scripts), the build order, the bindings of parameters to commands, identification of the library and paths, options for translator tools, and it provides some conditional processing rules.

The configuration thread represents the version selection of components to be used in the building of the product. Versions can be selected based on certain characteristics, equivalencies, or compatibilities. A bound configuration thread represents all the instances of components that were used to build the product, and has a unique identification. DSEE can, at any time, completely regenerate a particular instance of a product due to this configuration thread. Each derived object has a bound configuration thread associated with it.

Optimizations are achieved in space savings by eliminating the need for multiple copies of objects, by using compatible derived objects, and by allowing temporary inconsistencies in the built product. A distributed build improves compile time and shares the load across machines.

A.8. ISTAR

ISTAR from Imperial Software Technology Ltd. is an environment designed to support project management and a certain process model. Relationships between individuals on a software project are modeled as contracts. A contract is theoretically a description of expected products and is implemented as a database. A configuration item is the unit of transfer between contracts and is considered "frozen" when transferred. The transferring of contracts indicates certain tasks or phases are complete. CM exists for items in the contract databases. Successor and variant control is provided for components in the databases. The user can define relationships between CM components and can assign components to a problem report. There is support for system building.

A.9. Network Software Environment (NSE)

The Network Software Environment (NSE) from Sun Microsystems is an environment with a database that manages the UNIX directory structure and derived files in addition to the source code. NSE provides for team support in developing code. Workspaces support nested/recursive transactions with a protocol for merging and updating files between a child and parent workspace. The files in the workspace represent a configuration and can represent multiple versions of the configuration. All but the last configuration is immutable. Multiple users of the same workspace must check out and check in files while working in that workspace. The workspace effectively captures the directory structure used to store the source and derived objects of the product, the build structure and the logical structure of the product. The structure of a product is made visible via the browser.

A.10. PowerFrame

PowerFrame from EDA Systems, Inc. provides configuration management for CAE/CAD work. It shields the user from the operating system and file systems through a uniform, graphical/iconical interface. Design teams have their own design management environment and database. Operationally, users pull down an appropriate tool menu and PowerFrame automatically finds all relevant data, runs the tool, and saves all the changes after the user is finished.

PowerFrame incorporates several ways of organizing data in a product—a *project*, a *vista*, a *view* and a *datapack*. The project is a collection of data that is the subject of the work of a co-operating team (i.e., the product that includes all versions of the files for every phase of the design). A vista is a working set of file versions in use by a particular engineer at any time. A view allows users to concentrate attention on a particular aspect of a design (i.e., information relevant only to a logic-schematic view or a layout view is displayed). A datapack is a logical unit (i.e., in electronic terms, an Arithmetic Logic Unit) that is an abstraction of some component being designed; it allows detailed data—such as that produced by various tools—to be hidden, yet accessed when needed; in effect, PowerFrame groups together all related information regarding that abstraction. Generally, the user does not need to delve into the datapack but just requests that a particular action be performed on it. Note that all these notions are implemented as configurations. Some objects are automatically versioned (via check-in and check-out commands) whereas others are versioned by checkpointing (such as projects and datapacks).

A.11. Rational

The Rational Environment from Rational provides support for teams of programmers working on large Ada products. Rational's CM facilities are based on its subsystem concept. Ada program libraries are integrated with their CM. A subsystem represents a portion of the Ada product. Subsystems can be developed independently of other parts of the product by a single software engineer or by a team in a coordinated manner; a subsystem has a version identifier; a subsystem can be released; different versions can be worked on simultaneously; and subsystem can be combined with other subsystems or merged with others. Rational provides mechanisms that minimize the need for recompilation of Ada units. With the subsystem, Ada units can (optionally) be checked out and in, if that level of control is desired.

A.12. Revision Control System (RCS)

The Revision Control System (RCS) provides simple version control with the simple configuration identification. The repository is a version tree of files. A *branch* represents a variant of a file. RCS uses a numbering scheme to number versions and branches. Only the differences among files (*reverse deltas*) are stored in the repository. This saves space and also allows the latest version in the tree to be accessed most quickly.

Tags on versions are used for tying together file versions into a configuration. The configuration is specified by picking a file version from each line of file history and tagging it. To compose a configuration, the user gathers all the similarly tagged items. A version of a file is checked out of the repository (with a lock) and checked back in when the user has completed the changes. At this time, RCS logs details of the change such as the author, date, time, and reason for change. RCS can automatically incorporate a unique stamp into the actual file if the user wants one. RCS can also compare different versions of files, freeze a configuration, or assist in merging branches.

A.13. shape

The shape System is from the University of Berlin. It provides a repository with an attributed file system using Unix and incorporates a system model with integrated build facilities, workspaces and derived binary pools. Files and configurations can be retrieved via an attribute pattern. Configurations can be immutable or alterable. Attributes are user-defined or system-defined. Linear and parallel versions of configurations support team development. A workspace is a private work area (which changes its state between *busy* and *proposed*). The official public database is the long-term repository (and goes between states *published* to *frozen*). Code that is changed can be submitted for manual inspection before passing it to the official database. A system model and configuration thread (as in DSEE) are incorporated with the build facility.

A.14. Software Management System (SMS)

The Software Management System (SMS) from BiiN provides a version control, workspace management, system modeling and building, derived object management, configuration threads, change detection in the repository, tool interface specification, and attribute-based version selection with automatic expiration of some characteristics. Workspaces are protected. There is authorization and logging. The naming of files is transparent. Locked files are accessible only while attached to a private workspace. The user does not see version control.

Table of Contents

List of Figures