Software Configuration Management Issues in Product Development

Annie Ibrahim, Muhammad Waseem
(National University of Computer and Emerging Sciences, Lahore)

Abstract

After more than 20 years of research and practice in software configuration management (SCM), constructing consistent configurations of versioned software products still remains a dispute. Software Configuration Management (SCM) is viewed as an important key technology in software development. In general the configuration management task is about the description, instantiation and building of systems on the basis developed modules. Despite recent advances in software configuration management, constructing consistent configuration of large and complex versioned software products is a challenge. SCM related issues like version control, product model, release management and change management just to name a few are discussed in detail and the solutions to these issues are discussed in the paper.

Keywords:

Software Configuration Management, SCM Issues, version control, software component, configuration item, revision, release, variant

Software Configuration Management

According to CMM, "The purpose of SCM is to establish and implement the integrity of the products of the software project throughout the project's software life cycle". This definition shows that SCM involves in each phase (Requirements, Design, Coding, Testing, and Deployment) of the software development life cycle. Product development is a challenging task that needs comprehensiveness in terms of processes, and environment for its development. Product(s) tend to evolve with many man years of effort. Its life time is more than a project's life. That's why SCM activities have become vital in Product Development while looking at all of the issues supposed to occur during the development process, and it's a challenging task to handle all those product related SCM issues.

1. History

The term configuration management (CM) has been around since the forties. It started in the defense industry environment as a management technique and a discipline to resolve problems of poor quality, wrong parts ordered and parts not fitting, which were leading to inordinate cost overruns [1].

Software configuration management (SCM) has its roots in configuration management, but it differs from the management of physical product in several aspects [1].

- Software is quickly changed but unfortunately, the effects of the change(s) on the whole product may be complex.
- Software can be duplicated almost in no time, which easily leads to the existence of several copies of software components for example during the development of the product. Without decent coordination, these copies soon diverge and confusion arises.
- The fact that software components and often software documentation is stored on electronic media offers possibilities to extensive automation of their management. So normally SCM process is aided through computer aided software tools.

SCM is different in many aspects as above stated from ordinary configuration management process.

2. Introduction

The terminology in the area of configuration management is neither well defined nor consistent. There are different terms meaning the same issue and some terms having several meanings depending on the writer presented in the literature. In this section the principal SCM terms are presented according to IEEE standards. The IEEE standards, as well as a great deal of SCM literature are written mainly for large projects and impose a lot of bureaucracy on development process. Let's have an overview of SCM and SCM related terminologies.

- ***Configuration*** is the arrangement of a computer system or component as defined by the nature, number and interconnections of its constituent parts.
- ***Component*** means one of the parts that make up the system. Components may be further subdivided to other components resulting in a hierarchical representation of the system.
- ***Configuration Management*** is a discipline applying technical and administrative direction and surveillance to: identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements.
- ***Configuration Item*** (CI) is an entity treated separately in the configuration management process.
- ***Version*** is instance of a system which is functionally distinct in some way from other system instances.
- ***Revision*** is a version that supersedes a previous version.
- ***Variant*** is instance of a system which is functionally identical but non-functionally distinct from other instances of a system.
- ***Delta*** is the difference between two versions in which a delta (difference) and base version (common) make one complete version, each version has different deltas but have same base version.
- ***Release*** is instance of a system which is distributed to users outside of the development team.

The definition of configuration means that in order to specify a configuration we must know all the parts, identified, e.g., by their name and version number, and relationships between the parts belonging to the desired configuration.

3. Activities

From the IEEE's definition of configuration management, we can extract the four main SCM activities: identification, control, status accounting, and auditing. Some new activities have been suggested by Jari Vanhanen [1]. All of the main activities are described below.

3.1. Identification

Identification selects the configuration items to be managed. This may include, e.g., program files, source code files, design documents, management documents, and tools. Actually, every identifiable item related to the product should be considered a candidate for configuration management. Appropriate baselines for the project should be defined.

A baseline is a set of items that has been formally reviewed and agreed upon. It is strictly controlled so that it can serve as the basis for further development. Identification involves the definition of naming standard for the items and related to this the numbering system, which distinguishes the separate versions of an item from each other. Identification activity should also specify the software libraries where items are stored and how to retrieve and reproduce items from the library.

3.2. Control

One of the most important goals of configuration management is to maintain the integrity of the product. This is achieved by controlling changes to all configuration items. Control is placed on the whole change process from the initiation of a change request, through its evaluation, approval or disapproval to the implementation and verification of the changes.

Change requests contain information such as the Configuration Items (CIs) to be changed, the originator, the date, the urgency level, the need for the change, and the description of the change. Different levels of authority are needed to make changes depending on the involved items, life cycle states of the items, and baselines affected. Changes to a released item need much higher level of authority than changes during the development.

The body, which controls the changes, is usually called configuration control board (CCB). It consists of one or several people, depending on the size of the project, organizational structures, and bodies involved in the project. The composition of the CCB may change during the project reflecting the different levels of authority needed in changing different baselines.

3.3. Status Accounting

Recording and reporting the information on executing SCM activities is called status accounting. This activity includes identifying what information is needed, how the information is obtained, and what kinds of reports are produced.

3.4. Auditing

Auditing is a way to verify that configuration items match the requirements and the package being reviewed is complete. Configuration audits can be divided in functional and physical configuration audits. Functional configuration audit is executed to verify that a configuration item agrees with its specification documents. It is conducted by reviewing the test report data and comparing the statements in test reports to the specifications. Physical configuration audit is performed to verify that correct versions of all the items belonging to the configuration are present.

3.5. Process Management

Organizations often have own procedures, policies and life-cycle models defined and process management supports in ensuring their correct execution. It may, for example, enforce each source file to go through the predefined states like development, testing and quality assurance.

3.6. Team Work

The work and interactions between multiple developers of a product should be controlled. For example, procedures are needed to confirm that locally made changes are merged into the new release of the product.

4 SCM Process Areas

Configuration Management becomes more vital for products than for the projects. SCM has been identified as one of the challenges of Product development. Some of the issues faced during performing SCM are discussed in below.

4.1. Configuration Management

There is a need to store different modules of a software product and all of their versions details carefully. This topic includes version management, product modeling and complex object management.

4.2. Release Management

Software Release Management is a process through which software is made available to and obtained by its users. Software is constructed via the assembly of pre-existing, independently produced and independently released components. Those dependencies should be properly tracked for every component in a release.

4.3. Change Control & Managements

Change management is the truth regarding software. Change control should be properly managed. Change requests and Problem reports for every product involved should be initiated, recorded, reviewed, approved and tracked.

4.4. Version Control & Management

While revision management shows development files in progress, a version is a particular instance of an entire development project. A version is usually thought of as all the functions, features and complete builds you can use right now. Files managed, version-labeled and promoted by the revision management system are used to build the version.

4.5. Product Model

Since the beginning of SCM, the focus has always been given to file control / management. It is of no surprise to see that even today, the data model resemble a file system. This is archaic and contrasts with today's data modeling.

4.6. Revision Management

This is the core function of SCM and the foundation upon which other functions build. Revision management is the storage of multiple images of the development files in an application, which can be shared by multiple developers. With revision management, you can get an image a snapshot in time of the development process and can re-create any file to the way it was at any point in time.

4.7. Build management

When the code files, compilers, development tools and other components needed to create an application are ready, developers make an application build. Build management imposes automated, repeatable procedures that speed the build process, improve build accuracy and make it easier to create and maintain multiple versions of an application. An SCM toolset can become a vital integrative resource for building an application across multiple platforms. A strong cross-platform build management capability also eliminates the need to redefine the build process (the script) for each platform you are targeting. Configuration Management becomes more vital for products than for the projects. SCM has been identified as one of the challenges of Product development. Some of the issues faced during performing SCM are discussed in below.

5 Prior Literature

Product development needs to make use of reusability a practice. Different components need to be reused in different products. Component-based Software development (CBSD) is an emerging technology making reusability a reality. The traditional SCM methods cannot support CBSD effectively and efficiently. Traditional SCM tools consider file as the basic configuration item. For CBSD the basic constituent of the system is component rather than a file [2].

During the Product development of large and complex software systems, it is often necessary to recompose it in many different ways. Both because its components evolve and because there may be the need to configure the system differently for different hardware, operating systems, or clients [3].

Consistently configuring a large product existing in many versions is a difficult task. Many constraints on combining revisions and variants must be taken into account. Frequently, these constraints are neither documented properly nor specified formally. Then, it is up to the user of a SCM system to select consistent combinations of versions. Furthermore, dependencies between changes must also be taken into account, e.g. bug fixes may require other bug fixes to be included [4].

SCM tools are too often monolithic and their capability to inter-operate is limited. SCM tools should provide support for data modeling of complex objects, configuration

control with automatic selection and consistency criteria. A growing number of tools include a process, in an explicit formalism or not; they are called Process Sensitive Systems (PSS). Planners, mail systems, projects management tools are all examples of PSS. The data and concepts on which PSS and SCM work are roughly the same. Companies will require covering the complete process spectrum, with minimum redundancy and overhead, but still using the specialized tools they are used to. SCM is nothing else than one of these [5].

A software architecture description defines the structure of a software system in terms of components and relationships. Software Configuration Management provides version and configuration control for all software objects created throughout the software life cycle. These disciplines overlap considerably. In particular, both software architectures and software configurations describe the structure of a software system at a coarse level [6].

At the heart of software release management is the notion of dependence, by which we refer to a component's need for a set of other components in order for that component to properly function. Understanding a component's dependencies is complicated by the fact that components may be developed by different organizations, that those organizations autonomously control the release of new versions of their components, that each version of a component may have different dependencies, and that dependent components may themselves be complex component-based systems [7].

6 Product Development related SCM Issues

Software Configuration Management (SCM) is parallel activity to product development [8]. . The issues have been categorized into different phases of Software Development life cycle. These issues are discussed and then analyzed against different models studied to see which model helps to handle which issues.

6.1. Requirements Phase

The first law of systems engineering is that no matter where we are in the system life cycle, the system/software will change, and the desire to change it will persist throughout the life cycle. Dealing with this change represents a major management challenge. The essential role of requirements phase is to ensure that the users' needs are properly understood before designing and implementing a system to meet them. The requirements also provide part of the basis for system and acceptance testing. There is a requirement uncertainty principle that states that the greater the functional change, the less accurate the requirements.

Issue 1: Minimal Requirements

During requirement elicitation, most emphasis is given to gather requirements, and requirements are not managed properly e.g. No formal documentation for requirements is done. Due to this issue, most requirements may be missed and may result in minimal requirements. As continuous changes evolved but due to lack of any formal process to manage the requirements, most of the requirements are dropped or missed.

Issue 2: Traceability

During requirement engineering, requirements are gathered and placed in traditional fashion e.g. most requirements are documented in ordinary text files, no hierarchical way to place the requirements. There different levels of requirements, Business Requirements, User Requirements, Functional Requirement, Non Functional Requirement and System Requirement. If no relations or links are kept among different levels of requirements then this issue can result in poor validation process for requirements and ultimately poor testing of all required features.

Issue 3: Pre-Mature Requirement

If design is started early while the requirements are not firmed, a premature requirements freeze will likely result, which leads to poor design based on minimal

requirements. And excessive late changes may need to be made at later stages. If changes are made in hap hazard way with out any formal change mechanism, there is a 100% chance of rework.

Issue 4: Concurrency/Revisions

If two groups are working on software requirement specification (SRS) document and changes are made in SRS document from both groups, now if changes of one group are not visible to other group, it may possible that one may overwrite others changes. Or more over they may continue with two copies of SRS and other different groups start working on different copies of SRS.

Issue 5: Variations

For a product, there may be a vast pool of customers having different requirements, if there is no proper requirement management process, requirements of different customer or users can be mixed up.

6.2. Architectural / Design Phase

A software architecture description defines the structure of a software system in terms of components and relationships. Software architecture evolves as products become mature with the passage of time. Product development spans over a long time. During this time period, technologies emerge, the change of technologies / industry standards introduce the change in software architecture. Software architecture and Software configuration, both deals with the organization of software but their focuses differ: software development and software management respectively [9].

Architectural designs make use of case tools. Case tools if not integrated with the SCM tool produces overhead on the part of the developer. The decision of the case tool(s) need to be taken keeping in mind the SCM tool used.

Issue 1: Integration

If more groups are designing different modules or components, at the end or parallel to designing, the design items (modules or components) of different teams will be integrated. Now if changes are made to any design item, it should be integrated with the design. But due to lack of change control process, latest design item may not be integrated and faulty design may proceed to development.

Issue 2: Traceability

Design should be traceable in terms of requirements. If a change is made to any requirement, it should be reflected in design too. But some times due to lack of traceability, many requirement changes may not appear on the design side.

Issue 3: SCM & Case tools integration

Since vendors of SCM systems strive for reusability, they make virtually no assumptions concerning the application tools to be supported. Vice versa, the vendors of application tools e.g., architectural design tools usually do not want to become dependent on a specific SCM system. The architectural design tool need to be extended according to the SCM system used if tight integration is required among case tool and SCM tool. Presently there is no standard interface to SCM services.

Issue 4: Version Difference

Working with case tools produce different versions of the design documents. Generally SCM systems provide a mechanism to look at the differences among different versions of source files. But generally this facility is not being provided to the design documents produced using cast tools through SCM system used. The example is of Rational Rose and Visual Source Safe.

6.3. Implementation Phase

Issue 1: Version Control

The source code is versioned. Each module should be able to compile with other modules, to link them with base lined object programs and to use them in private workspaces. There should be a facility so that

not only the source code but the executable should also be versioned and associated with the source code for tracking purposes.

Issue 2: Sharing

If multiple products are under development and products are using underlying facilities of each other. The module may be shared among different products. The change in the module should be made after consideration of both the products. If for some change the change is only required in one of the products. It should be made possible through the SCM system.

Issue 3: Merging

If parallel development is going on a single file. The difference versions of the file need to be merged. This merge mechanism should be properly handled through the SCM system. Merging is a very critical task and need to be controlled properly so that no in-consistencies are introduced to any module.

6.4. Testing Phase

Testing is major software quality assurance technique and it plays a vital role in product quality. There are certain issues which must be considered in order to improve the quality of product.

Issue 1: Derivations

A costly fixed bug reappeared after a minor change.

6.5. Release Management

Software release is a process through which software is made available to, and obtained by, its users. There are certain issues associated with the release.

Issue 1: Dependency Relationship

It is critical for a developer to be able to easily and accurately document dependencies as part of the release process. Moreover, once recorded, those dependencies should be directly usable and traceable through the SCM tool being used for this purpose. This traceability is required to keep track of the modules dependencies in different releases. Moreover, newer versions of components should not replace the older versions.

Issue 2: Deltas

When a new version of a component is to be released, the developer should only have to specify what has changed, rather than treating the new version as a completely separate entity. This overhead should be minimized on the developer part.

Issue 3: Conditional Installation

At the time of installation, a customer may not require a specific functionality from the product.

6.7. General Issues

Above issues seem very innocent and minor but are root causes for poor quality of product in terms of delayed time to market, over budget, and least functional. Other general issues [9] are:

Issue 1: Storage Space

Multiple products may be developed with the same development team working on it. Multiple products may share some modules. In such a case, any changes made to the module will be accessed across the products. Such sharing causes a lot of storage space on the part of SCM system used. The storage space increases rapidly that it becomes hard to manage the products. The increases storage space not only causes to enhance the storage space but the efficiency of the SCM system is also affected.

Issue 2: Component Management

Product development involves different modules. No module consists of a single file. That is why a file is not a logical constituent in an application. Usually, a logical constituent is implemented into a set of files like a module. It is very difficult to manage so many files and their relationships. For traditional SCM systems, every thing revolves around the file but the reality is other way round.

Issue 3: Multiple developer syndromes

When you have a project that requires more than one developer, there is the problem with multiple people working on one product base. This could be a test plan, requirements specification, or code. Effort is wasted when two or more people work on same file and then save it. Without SCM control, the last person to save the file has those changes saved. All the other changes are lost. The simplistic method of locking a file while one person reads it prevents others from simultaneously working on the file.

Issue 4: Multiple releases

Enhancements to the base product should result in additional releases of the product containing the latest changes. Once the second release is available, some users are on an earlier release. Having an SCM makes managing those releases possible. When bugs are reported, changes must be made across all impacted releases. As new features become available in the product, they must be made available to all current users, no matter what the release date.

Issue 5: Product family

As products are built that offer the same capabilities across a heterogeneous set of hardware platforms, both the common and the platform-specific software bases must be managed. If a product operates on four versions of Windows, three versions of Unix, Red Hat Linux, and FreeBSD the user manual may be significantly the same. But there is a different installation process for all nine platforms. Without SCM, nine individual manuals must be written and maintained. With SCM, one documentation configuration item with nine versions will suffice, with the differences being only the installation procedure.

Issue 6: Schedule change

As requirements change, so must the schedule. Mapping the feature sets for release to the schedule allows project managers to more accurately estimate the effort required for generating that next release. Having the SCM in place allows the project manager to look at historic effort levels in getting out releases. This is an enormous aid in estimating the "what if" scenarios that result from taking on new product users or providing customized solutions to other clients

Issue 7: Software changes

No product developer has the luxury to write code once and forget about it. Along with requirements and schedules, the software being developed changes in response to those other changes. Software is not static. That is its inherent power. It can be changed, so it will be changed. SCM systems track those changes so that, if the wrong change is made, a previous working version is available. This capability alone has saved enormous amounts of time as developers have tried out specific solutions that did not work in the product environment and were able to rapidly back up to a working version.

Issue 8: Staff changes

In the best of organizations, people get promoted, take other jobs, and leave. When that happens in the midst of a development project, not just the technology knowledge goes out the door. The long-learned knowledge of how things are done is also gone. So when a replacement person is brought on board, they may know the technology, but without a documented SCM process, they will have no real idea how to do product development. SCM provides the framework and knowledge base of what has gone on before in the project. A new staff member has one place to go to understand the "how" of the organization's development process and the "what" of the project to date.

System/user documentation change: No product developer has the luxury to produce in a technology or tool vacuum. All product developers use hardware microcode, operating systems, tools, and documentation that are not under their control. When a major operating system change occurs (e.g., the next "best" release of Windows), an SCM will allow tracing all the CIs, components, and

subcomponents that are impacted by that change. The change is isolated, and the amount of effort required to respond to the change can be estimated. This provides a responsible schedule for an upgrade based on situations beyond the organization's control.

7. SCM Models

All of the figures in this section are available in Figures section at the end. In this term paper, we have presented few fundamental SCM models, which have been deployed for many tools like Source Safe, Clear Case etc. These models has laid rudimentary base for many modern models like Dynamic Model [3]. The presented models are: Check in\Check out Model [1], Composition Model [1], Long Transition Model [1], Change Set Model [1], Dynamic Model [3], CBSD Model [2], and Requirement Processing Model [10].

7.1. The Check-out/Check-in Model

The check-out/check-in model is the traditional model used by such well-known configuration management tools as Source Code Control System (SCCS) and Revision Control System. The central concept is the repository, where all the individual files and all of their versions are stored. Usually, the files in the repository can not be operated directly by developer tools but explicit operations are needed to store a file into the repository (check in) and to retrieve it back to the desired directory (check out).

When a file is checked in, usually after some modifications, a new version of that file is created. When checking a file out of the repository the desired version has to be denoted. The files can be checked out for reading or writing allowing concurrency control actions to avoid undesired concurrent changes to the same version of a file. When a file is checked out for writing, locking mechanism can guarantee that no other person modifies the same version of the file until it is checked back in to the repository.

7.2. The Composition Model

Where check-in/check-out model deals with individual files, the composition model focuses on supporting configurations. In this context a configuration consists of a system model, which lists all the components that make up a system and version selection rules which are applied to the system model in order to choose the desired version of each component.

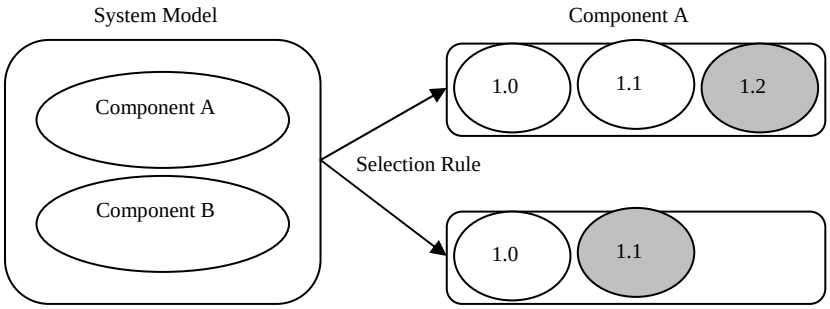

Figure 1: Composition Model

Selection rules may specify either a revision or a variant of the file and thus support management of system variants. In figure 1, a system consists of components A and B. A selection rule choosing the latest version of each component has been applied resulting to version 1.2 of component A and version 1.1 of component B.

7.3. The Long Transaction Model

The long transaction model focuses on supporting the evolution of the whole system as a series of apparently atomic changes, and provides team support through coordination of concurrent change. Developers operate primarily with versioned configurations. In contrary to the composition model they first

select the version of the system configuration, and then focus on the system structure. The selected system configuration determines the versions of the components used.

may be visible to the testing team and so on until the hierarchy ends to the repository.

Three categories of concurrent development are supported:

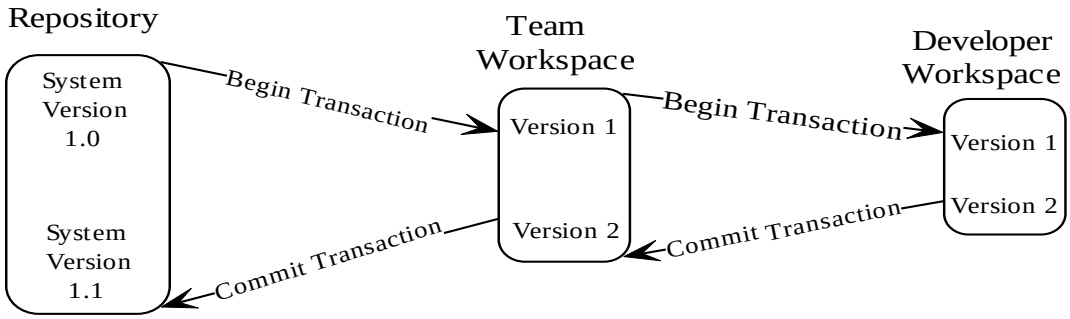

Figure 2: Long Transition Model

When making a change a transaction is started. The change is made in a workspace, which represents the working context and provides local data storage visible only within the scope of the workspace. A workspace may be mapped into the file system allowing transparent access to the repository for the development tools. A workspace consists of a working configuration, where modifications are made and possibly several preserved configurations, which are frozen states of previous working configurations. A workspace originates from a bound configuration in the repository or from a preserved configuration of an enclosing workspace. When the changes are finished, the transaction is committed, which effectively creates a new version of the configuration in the repository or enclosing workspace and make the changes visible outside the workspace. Finally the workspace may be deleted or it may be used for further changes. If the workspace originates from another workspace, the result is a hierarchy of workspaces. The different levels in the hierarchy represent different levels of visibility.

The bottom workspaces belong to the individual developers, one level up is the workspace for the team and the next level

- concurrency within one workspace,
- concurrency between workspaces requiring coordination, and
- concurrent, independent development.

In the first case concurrent changes are restrained by allowing only one person at a time to change the file. The control may happen at different levels: limiting access to a workspace to one person; allowing only one person at a time to be active in a workspace; or locking individual components for exclusive use of one person at a time. In the second case changes in separate workspaces together evolve the system. Schemes for controlling this concurrency may be conservative or optimistic. Conservative schemes require a priority locking across workspaces. In optimistic schemes conflicts are detected when changes are committed. Third case assumes that system evolves in independent development paths and changes need not be coordinated when created.

Figure 2 illustrates the concepts of the long transaction model using a simple example. In the beginning of the first transaction, a working configuration of system version 1.0 is created into a team workspace. Then a new transaction is started, which creates a new working configuration to a developer

workspace. In the developer workspace some changes are implemented to the system and working configuration version 2 is created, making version 1 a preserved configuration. Committing the second transaction creates a new working configuration to the team workspace and finally committing the first transaction creates a new system version to the repository.

of configurations can be made. These queries include:

- Determining which component has been modified as part of logical change.
- Determining the collection of change sets a particular component is part of.

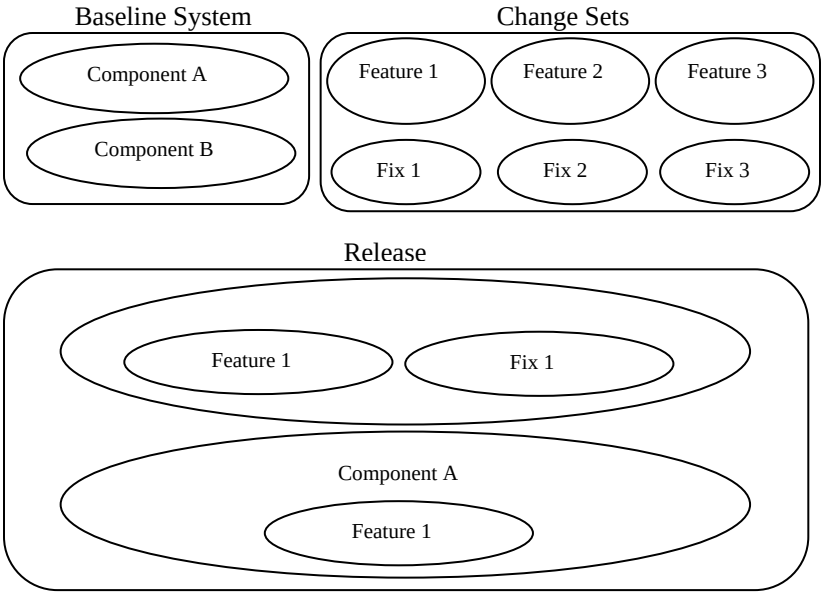

Figure 3: Change Set Model

7.4. The Change Set Model

The main concept in the change set model is the change set, which represents the set of modifications to different components making up a logical change. A typical case is that implementing a requested change to software requires modifications to several components.

Change sets simplify several operations. Developers can work with groups of components belonging to the same logical change instead of dealing with each component separately. Change requests, which are descriptions of the changes to be made, may be easily linked to the actual changes made to the components.

Queries on the dependencies between logical changes, changed components, and versions

- Determining which change sets are included in a particular configuration.
- Determining which configurations include a particular change.

Configurations in this model consist of a baseline and several change sets applied to the baseline. Different configurations can be made by applying different collections of change sets to a baseline. However, all combinations of change sets are not necessarily consistent. Some of them may be dependent on other change sets and some may be in conflict with other change sets. Some method for determining the physical and logical dependencies between changes has to be used.

In figure 3 a system release is constructed applying changes Feature 1 and Fix 2 to the baseline system.

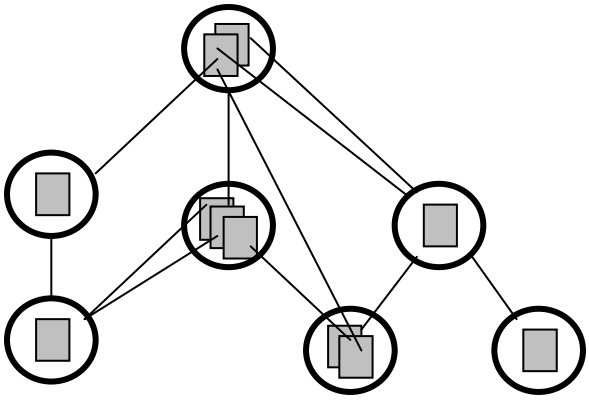

Figure 4: Dynamic Model

7.5. Dynamic Model

Whatever the changes are made, they are always made in the modules of the program. Suppose a version "X" requires a change "abc", and this change should be made in a module named as "mod1". Now let's suppose that the change is commonly required to be made in other versions too, let's say version "Y". Now if we changed the module "mod1" and puts information of usage of module locally for every version, this can resolves our problem for common change required in both versions if both versions use the same module. In this way, we don't need to make common changes separately in all versions as shown in the figure 4.

7.6. CBSD Model

CBSD model is based on Component-based Software Development model. The model is shown in the figure below. The goal of this model is to support CBSD more efficiently, so some concepts in CBSD are still used in this model, and there are a few changes in some concepts concerning the traditional SCM. This model says that Files are the basic physical storage units. Component is an integral logical constituent in a system, which is relatively independent on its surroundings concerning the functions and behaviors. The operations for management of components are:

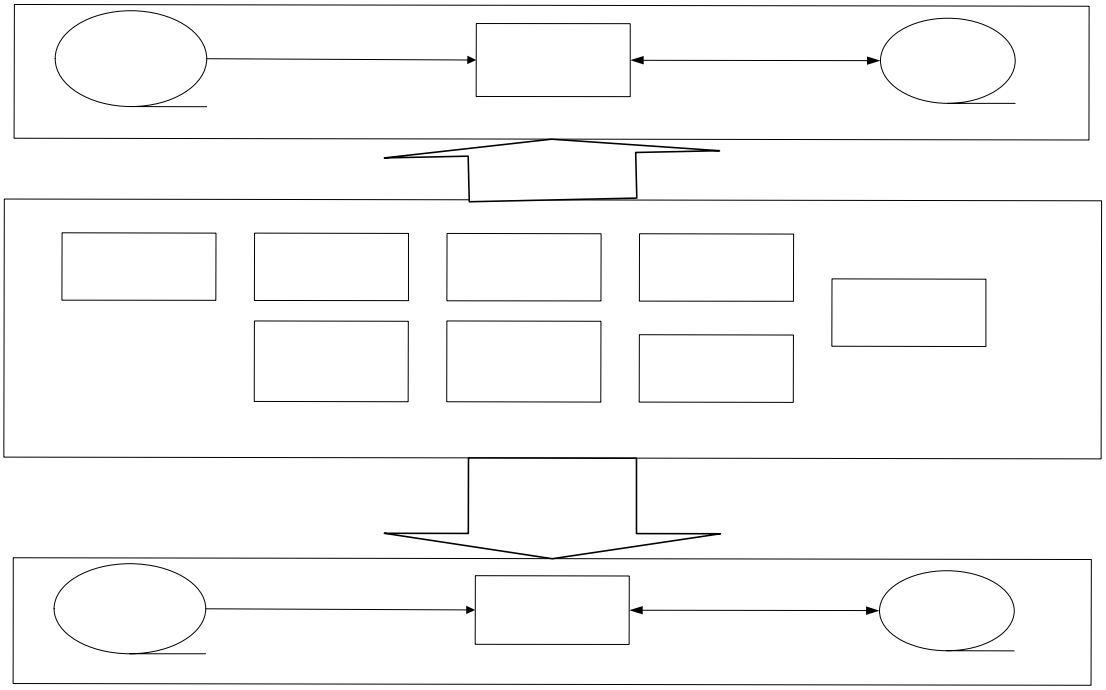

Figure 5: CBSD Model

Component Composition

The checkout-edit-checkin style is used to manage each evolutionary direction of a primitive component. To make a change in a

Compos
Compone

The C

primitive component, it should be checked out first. After the change, it should be checked in and thus becomes a new version.

Primitive components can have several evolutionary directions. By using the branching operation, a new evolutionary direction can be created from any of the existing versions of a primitive component. An evolutionary direction is also called a branch.

not clearly defined, especially in the beginning of the development process when requirements are not clear and not completely understood. In general, we have many-to many-relations between RS Items and system modules1: A module can be affected by several requirements, and, on the contrary, a requirement can be implemented in different modules. These relations are even more complicated if we take into considerations that some requirements are related to other

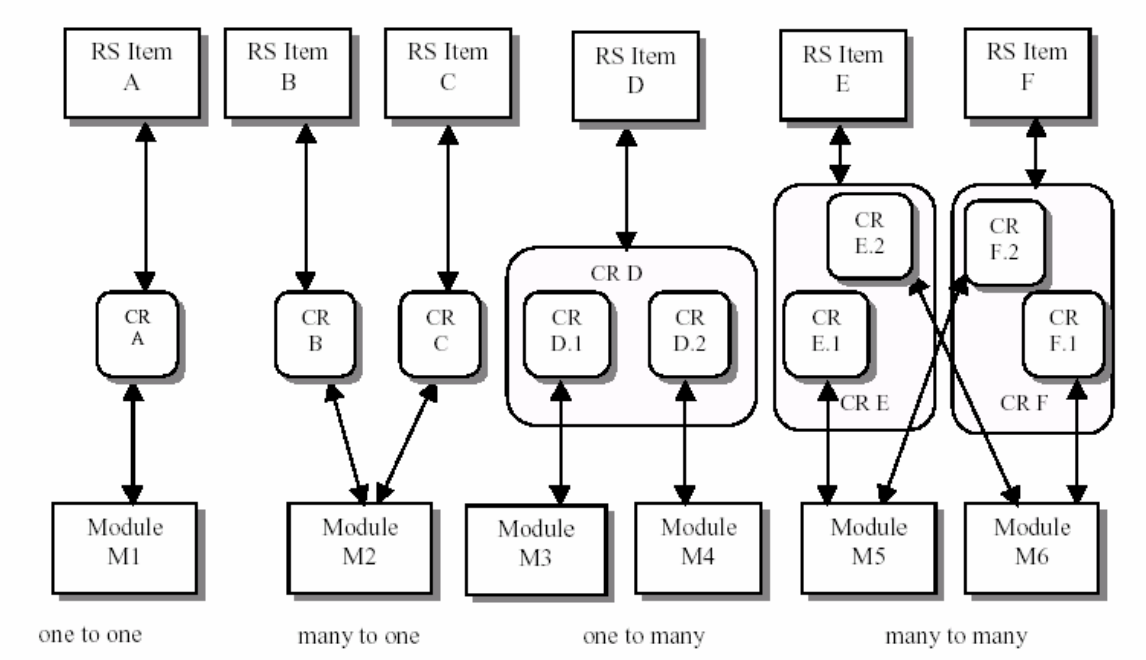

Figure 6: Requirement Processing Model

Different branches can merge together during the evolution of the primitive component.

A primitive component is usually under the development of a team. The concurrency operation is used to coordinate the different modifications from different developers in order to form a primitive component.

7.7. Requirement Processing Model

When we place Requirements Specification under version control, we obtain control over the requirements changes. This control is however not sufficient, we also need a mechanism to relate requirements to the implementation parts. This relation is often

requirements and the same applies to modules. System modules can consist of a number of items, such as documents and source files, and in such a complex relationship keeping track of requirements is a challenge.

We wish to make requirements more visible during the implementation. Developers should be aware of the requirements they are expected to implement, and they should be aware of possible changes in the requirements of which requirements are being implemented and of which new requirements have appeared. One possibility of relating requirements with system modules is to use Change Management support from CM.CM

provides a basic support for change process operations and the CM functions can be utilized for providing links between requirements and the final result of the development process.

Change Requests uniquely address system modules where the requests are to be implemented (Figure 6).

The relation between RS Items and Change Requests is bidirectional. An RS Item refers to the associated Change Request, and, if a Change Request is generated from an RS Item, it contains a reference to the corresponding RS Item.

9. Conclusion

Despite many limitations and expected improvements on the basis of issues identified, SCM has proved one of the few successful software engineering technologies.

It may appear that most of SCM research was performed in the 80's, and only tool improvement can be expected in the future.
The issues highlighted have shown that future work needs to be done for SCM models improvement. This paper shows that much is still to be done both to find new concepts and better implementations.

Issues identified during Product Development

Models	Requirement					Design				Implementation			Testing	Release Issues		
	Issues					Issues				Issues			Issues	Issues		
	1	2	3	4	5	1	2	3	4	1	2	3	1	1	2	3
Check in / Check out				X						X		X	X			
Composition Model				X	X					X	X	X				
Long transaction Model				X	X					X	X	X				
Change Set Model	X	X			X		X			X		X	X	X	X	
CBSD				X		X				X	X	X		X	X	
Requirement	X	X	X		X	X	X									

Table 1: Comparison of models with issues

8. Model Comparison for Product Development Issues

A number of issues have been identified in section 'Product Development related SCM Issues'. The issues have been discussed. SCM models presented above are analyzed in the following to see which issues are handled through which SCM model used. A comparison is performed on the basis of issues and the models presented. An 'X' represents that this issue is being handled by using the model under Model column. General issues have not been compared.

need to be handled through SCM systems under development. These improvements will help product development efficient. SCM concepts should be handled through SCM tools rather than putting more pressure on the side of the developer. Each phase is presented with a list of issues; SCM tools should provide support so that those issues can be handled properly.

Nevertheless, in the near future, provided the number of core topic issues yet to be solved and the efficiency, scalability and usability issues, no one of these evolutions will be seriously addressed by SCM vendors. SCM tools will still grow, propose proprietary

solutions and still consider SCM as an isolated domain.

10. Future Work

Some models are presented. But no single model solves the issues presented for Product Development. One of the reasons is that no standardization exists in SCM realm that is applied for the implementation of SCM systems in SCM tools. Moreover, models need to be well integrated so that future can see standards in SCM tools especially standards are needed to integrate the development/case tools with the SCM tools. It reduces the training time of resources on SCM system. Moreover, it enhances the user friendliness and reduces the hassle on part of the developer. Research should be done to standardize the SCM systems so that different SCM tools can be integrated to get best out of the both worlds.

11. References

[1] "*Improving configuration management processes of a software product*" By Jari Vanhanen, Master's Thesis, Helsinki University of Technology, 1997

[2] "*A Software Configuration Management Model for Supporting Component-Based Software Development*" By Hond Mei, Lu Zhand, Fuqing Yang, ACM SIGSOFT, 2001.

[3] "*An Integrative Model for Configuration Management and Version Control*" By Lars Bendix, 1996.

[4] "*Configuring Versioned Software Products*" By Reidar Conradi and Bernhard estfechtel, 1996.

[5] "*Software Configuration Management: A Roadmap*" by Jacky Estublier, 2001.

[6] "*Software Architecture and Software Configuration Management*" By Bernhard Westfechtel, and Reidar Conradi.

[7] "*Software Release Management for Component-Based Software*" By Andre van der Hock, Alexandar L. Wolf, 2001.

[8] "*Software Configuration Management for Project Leaders*" By Tim Kasse and Patricia A. Mcquaid, SQP Vol. 2, No. 4/ 2000, ASQ.

[9] "*How You Can Benefit from Software Configuration Management*" By Robert Futrell, Linda Shafer, and Donald Shafer. Prentice Hall Publisher, 2002

[10] *"Processing Requirements by Software Configuration Management"* By Ivica Crnkovic , Peter Funk , and Magnus Larsson, euromicro 99, proceedings of the 5th EUROMICRO conference, 1999