29 Configuration management

Objectives

The objective of this chapter is to introduce the process of managing the code and documentation of an evolving software system. When you have read this chapter, you will:

- understand why software configuration management is important;
- have been introduced to four principal configuration management activities – configuration management planning, change management, version and release management and system building;
- understand how CASE tools for configuration management may be used to support configuration management processes.

Contents

Configuration management (CM) is the development and application of standards and procedures for managing an evolving system product. You need to manage evolving systems because, as they evolve, many different versions of the software are created. These versions incorporate proposals for change, corrections of faults and adaptations for different hardware and operating systems. There may be several versions under development and in use at the same time. You need to keep track of the changes that have been implemented and how these changes have been included in the software.

Configuration management procedures define how to record and process proposed system changes, how to relate these to system components and the methods used to identify different versions of the system. Configuration management tools are used to store versions of system components, build systems from these components and track the releases of system versions to customers.

Configuration management is sometimes seen as part of a more general software quality management process as discussed in Chapter 24. The same manager may share quality management and configuration management responsibilities. Software is released by the developers to a quality assurance team who are responsible for checking that the system is of acceptable quality. It is then passed to the configuration management team who become responsible for controlling changes to the software. Controlled systems are sometimes called *baselines* as they are a starting point for controlled evolution.

There are many reasons why systems exist in different configurations. Configurations may be produced for different computers, for different operating systems, incorporating client-specific functions and so on (Figure 29.1). Configuration managers are responsible for keeping track of the differences between software versions, for ensuring that new versions are derived in a controlled way and for releasing new versions to the right customers at the right time.

The configuration management process and associated documentation should be based on standards. An example of such a standard is IEEE 828-1983, which defines a standard for configuration management plans. Within an organisation, these standards should be published in a configuration management handbook or as part of a quality handbook. External standards may be used as a basis for more detailed organisational standards that are tailored to a specific environment. It doesn't really

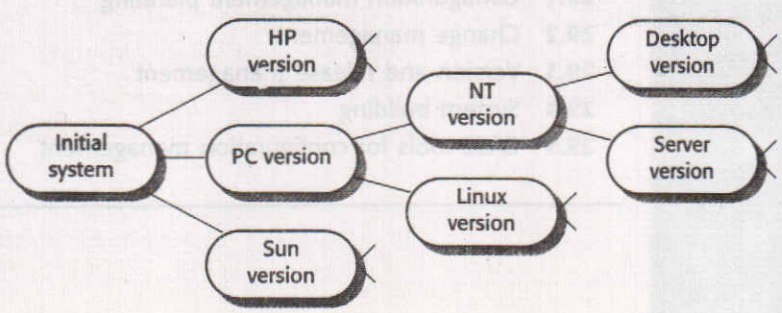

Figure 29.1
System families

matter which standard is taken as a starting point as all of them include comparable processes. In both the ISO 9000 quality standards (Ince, 1994) and the SEI's Capability Maturity Model (Paulk *et al.*, 1993), organisations must define and follow formal CM standards for quality certification.

In a traditional software development process based on the 'waterfall' model (see Chapter 3), software is delivered to the configuration management team after development is complete and the individual software components have been tested. This team then takes over the responsibility for building the complete system and for managing system testing. Faults which are discovered during system testing are passed back to the development team for repair. They then fix the fault and deliver a new version of the repaired component to the CM team.

This approach has influenced the development of configuration management standards and these standards have an embedded assumption that a waterfall model of the software process will be used for system development (Bersoff and Davis, 1991). This means that they cannot be readily applied to alternative approaches to software development such as evolutionary prototyping or incremental development. To cater for this style of development, some organisations have developed a modified approach to configuration management which supports concurrent development and system testing. This approach relies on a very regular (often daily) build of the whole system from its components:

1. The development organisation sets a delivery time (say 2 p.m.) for system components. If developers have new versions of the components that they are writing they must deliver them by that time. Components may be incomplete but should provide some basic functionality which can be tested.
2. A new version of the system is built from these components by compiling and linking them to form a complete system.
3. This system is then delivered to the testing team who carry out a set of predefined system tests. At the same time, the developers are still working on their components, adding to the functionality and repairing faults discovered in previous tests.
4. Faults that are discovered during system testing are documented and returned to the system developers. They repair these faults in a subsequent version of the component.

The main advantages of using daily builds of software is that the chances of finding problems that stem from component interactions early in the process are increased. Furthermore, daily building encourages thorough unit testing of components. Psychologically, developers are put under pressure not to 'break the build', i.e. deliver versions of components which cause the whole system to fail. They are therefore reluctant to deliver new component versions which have not been properly tested. Less system testing time is spent discovering and coping with software faults that should have been found during unit testing.

The successful use of daily builds requires a very stringent change management process to keep track of the problems that have been discovered and repaired. It also leads to a very large number of system and component versions that must be managed. Good configuration management is therefore essential for this approach to be successful.

29.1 Configuration management planning

A configuration management plan describes the standards and procedures which should be used for configuration management. The starting point for developing the plan should be a set of general, company-wide configuration management standards and these should be adapted as necessary for each specific project. The CM plan should be organised into a number of chapters and should include:

1. the definition of what entities are to be managed and a formal scheme for identifying these entities;
2. a statement of who takes responsibility for the configuration management procedures and for submitting controlled entities to the configuration management team;
3. the configuration management policies that are used for change control and version management;
4. a description of the records of the configuration management process which should be maintained;
5. a description of the tools to be used for configuration management and the process to be applied when using these tools;
6. a definition of the configuration database which will be used to record configuration information.

Other information such as the management of software from external suppliers and the auditing procedures for the CM process may also be included in the CM plan.

An important part of the CM plan is the definition of responsibilities. It should define who is responsible for the delivery of each document or software component to quality assurance and configuration management. It may also define the reviewers of each document. The person responsible for document delivery need not be the same as the person responsible for producing the document. To simplify interfaces, it is often convenient to make project managers or team leaders responsible for all of the documents produced by their team.

29.1.1 Configuration item identification

In the course of developing a large software system, thousands of documents are produced. Many of these are technical working documents which present a snapshot of ideas for further development. These documents are subject to frequent and regular change. Others are inter-office memos, minutes of group meetings, outline plans and proposals, etc. These documents may be of interest to a project historian. However, they are not needed for future maintenance of the system.

During the configuration management planning process, you decide exactly which items (or classes of item) are to be controlled. Documents or groups of related documents under configuration control are formal documents or *configuration items*. Project plans, specifications, designs, programs and test data suites are normally maintained as configuration items. However, all documents which may be necessary for future system maintenance should be controlled.

The document naming scheme must assign a unique name to all documents under configuration control. There are always relationships between these documents. For example, design documents will be associated with programs. These relationships can be recorded implicitly by organising the naming scheme so that related documents have a common root to their name. This leads to a hierarchical naming scheme where examples of names might be:

PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE/CODE
PCL-TOOLS/EDIT/HELP/QUERY/HELPFRAMES/FR-1

The initial part of the name is the project name, PCL-TOOLS. In this project, there are four separate tools. The tool name is used as the next part of the name. Each tool is made up of different named modules. This decomposition process continues until the base-level formal documents are referenced (Figure 29.2). The leaves

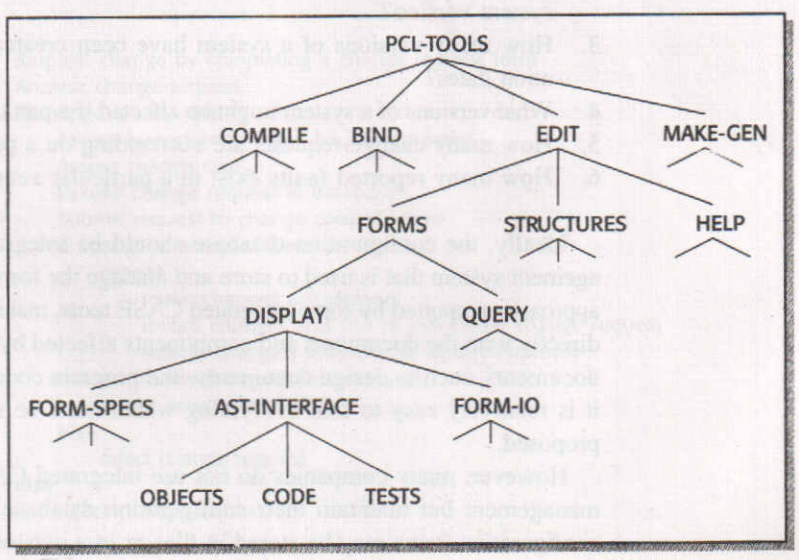

Figure 29.2 Configuration hierarchy

of the documentation hierarchy are the formal project documents. Figure 29.2 shows that three formal documents are required for each managed entity. These are an object description (OBJECTS), the code of the component (CODE) and a set of tests for that code (TESTS).

The problem with naming schemes of this sort is that they are project based. Identifiers associate components with a particular project. This may reduce the opportunities for reuse. Copies of reusable components should normally be taken out of such a scheme and renamed according to their application domain. Other problems may arise if the document naming scheme is used as a direct basis for designing a storage structure for storing the managed components. Users of the documents then have to know their name to find them and all documents of the same type (e.g. design documents) are not held together in one place. There may also be problems in relating the naming scheme to the identification scheme used by a version management system.

29.1.2 The configuration database

The configuration database is used to record all relevant information relating to configurations. Its principal functions are to assist with assessing the impact of system changes and to provide management information about the CM process. As well as defining the configuration database schema, procedures for recording and retrieving project information must also be defined as part of the CM planning process.

A configuration data base must be able to provide answers to a variety of queries about system configurations. Typical queries might be:

1. Which customers have taken delivery of a particular version of the system?
2. What hardware and operating system configuration is required to run a given system version?
3. How many versions of a system have been created and what were their creation dates?
4. What versions of a system might be affected if a particular component is changed?
5. How many change requests are outstanding on a particular version?
6. How many reported faults exist in a particular version?

Ideally, the configuration database should be integrated with the version management system that is used to store and manage the formal project documents. This approach, supported by some integrated CASE tools, makes it possible to link changes directly with the documents and components affected by the change. Links between documents, such as design documents, and program code may be maintained so that it is relatively easy to find everything which must be modified when a change is proposed.

However, many companies do not use integrated CASE tools for configuration management but maintain their configuration database as a separate system. The configuration items may be stored in files or in a version management system such

as RCS (Tichy, 1985), a well-known version management system for Unix. This configuration database stores information about the configuration items and references their file names in the version management system. While this is a relatively cheap and flexible approach, its disadvantage is that configuration items may be changed without going through the configuration database. Therefore, you can't be sure that the configuration database is an up-to-date description of the state of the system.

29.2 Change management

Change is a fact of life for large software systems. As discussed in earlier chapters, organisational needs and requirements change during the lifetime of a system. This requires corresponding changes to be made to the software. A defined change management process and associated CASE tools ensure that these changes are recorded and applied to the system in a cost-effective way.

The change management process (Figure 29.3) should come into effect when the software or associated documentation is put under the control of the configuration management team. It may be initiated during system testing or after the software has been delivered to customers. Change management procedures should be designed to ensure that the costs and benefits of change are properly analysed and that changes to a system are made in a controlled way.

The first stage in the change management process is to complete a change request form (CRF) where the requester sets out the change required to the system. As well

Figure 29.3 The change management process

```
Request change by completing a change request form
Analyse change request
if change is valid then
    Assess how change might be implemented
    Assess change cost
    Record change request in database
    Submit request to change control board
    if change is accepted then
        repeat
            make changes to software
            record changes and link to associated change request
            submit changed software for quality approval
        until software quality is adequate
        create new system version
    else
        reject change request
else
    reject change request
```

Change Request Form

Project: Proteus/PCL-Tools **Number:** 23/94
Change requester: I. Sommerville **Date:** 1/12/98
Requested change: When a component is selected from the structure, display the name of the file where it is stored.

Change analyser: G. Dean **Analysis date:** 10/12/98
Components affected: Display-Icon.Select, Display-Icon.Display

Associated components: FileTable

Change assessment: Relatively simple to implement as a file name table is available. Requires the design and implementation of a display field. No changes to associated components are required.

Change priority: Low
Change implementation:
Estimated effort: 0.5 days
Date to CCB: 15/12/98 **CCB decision date:** 1/2/99
CCB decision: Accept change. Change to be implemented in Release 2.1.
Change implementor: **Date of change:**
Date submitted to QA: **QA decision:**
Date submitted to CM:
Comments

Figure 29.4 A partially completed change request form

as recording the change required, the CRF records the recommendations regarding the change, the estimated costs of the change and the dates when the change was requested, approved, implemented and validated. It may also include a section where the maintenance engineer outlines how the change is to be implemented. Change requests should be registered in the configuration database. The CM team can therefore track the status of change requests and the change requests which are associated with specific software components.

An example of a change request form, which has been partially completed, is shown in Figure 29.4. The change request form is usually defined during the CM planning process. For some contracts, however, particularly government contracts, the change request form must conform to a specified client standard.

Once a change request form has been submitted, it is analysed to check that the change is valid. Some change requests may be due to misunderstandings rather than system faults; others may refer to already known faults. If the analysis discovers that a change request is invalid, duplicated or has already been considered, the change should be rejected. The reason for the rejection should be returned to the person who submitted the change request.

For valid changes, the next stage of the process is change assessment and costing. The impact of the change on the rest of the system must be checked. A technical analysis must be made of how to implement the change. The cost of making the change and possibly changing other system components to accommodate the change is then estimated. This should be recorded on the change request form. This assessment process may use the configuration database where component

interrelationships are recorded. The impact of the change on other components may then be considered.

Unless the change simply involves correcting minor errors on screen displays or in documents, it should be submitted to a change control board (CCB) who decide whether or not it should be accepted. The change control board considers the impact of the change from a strategic and organisational rather than a technical point of view. It decides if the change is economically justified and if there are good organisational reasons to accept the change.

The term 'change control board' sounds very formal. It implies a rather grand group which makes change decisions. Formally structured change control boards which include senior client and contractor staff are a requirement of military projects. For small or medium-sized projects, however, the change control board may simply consist of a project manager plus one or two engineers who are not directly involved in the software development. In some cases, there may be only a single change reviewer who gives advice on whether or not changes are justifiable.

When a set of changes has been approved, the software is handed over to the development or maintenance team for implementation. Once these have been completed, the revised software must be revalidated to check that these changes have been correctly implemented. The CM team, rather than the system developers, then builds a new version or release of the software.

When new versions of the system are created though daily system builds, a simpler change management process is used. Problems and changes must still be recorded but changes that only affect individual components and modules need not be independently assessed. They are passed directly to the system developer. They either accept them or make a case why they are not required. Changes which affect system modules produced by different development teams should be still be assessed by some kind of change control authority who decides if they should be implemented.

As software components are changed, a record of the changes made to each component should be maintained. This is sometimes called the derivation history of a component. The best way to maintain such a record is in a standardised comment prologue kept at the beginning of the component (see Figure 29.5). This should

Figure 29.5
Component header information

```
// PROTEUS project (ESPRIT 6087)
//
// PCL-TOOLS/EDIT/FORMS/DISPLAY/AST-INTERFACE
//
// Object: PCL-Tool-Desc
// Author: G. Dean
// Creation date: 10th November 1998
//
// © Lancaster University 1998
//
// Modification history
// Version   Modifier   Date        Change       Reason
// 1.0       J. Jones   1/12/1998   Add header   Submitted to CM
// 1.1       G. Dean    9/4/1999    New field    Change req. R07/99
```

reference the change request associated with the software change. Specialised tools may be used to process the derivation histories and produce reports about component changes.

29.3 Version and release management

Version and release management are the processes of identifying and keeping track of different versions and releases of a system. Version managers devise procedures to ensure that different versions of a system may be retrieved when required and are not accidentally changed. They may also work with customer liaison staff to plan when new releases of a system should be distributed. New system versions should always be created by the CM team rather than the system developers, even when they are not intended for external release. This makes it easier to maintain consistency in the configuration database as only the CM team can change version information.

A system *version* is an instance of a system that differs, in some way, from other instances. New versions of the system may have different functionality, performance or may repair system faults. Some versions may be functionally equivalent but designed for different hardware or software configurations. If there are only small differences between versions, one of these is sometimes called a *variant* of the other.

A system *release* is a version that is distributed to customers. Each system release should either include new functionality or be intended for a different hardware platform. There are always many more versions of a system than releases as versions are created within an organisation for internal development or testing that are never released to customers.

Version management is now always supported by CASE tools as discussed in section 29.5. These tools manage the storage of each system version and control access to system components. They must be checked out from the system for editing. When re-entered in the system, a new version is created and named by the version management system.

29.3.1 Version identification

Within a large software system, there are hundreds of software components each of which may exist in many different versions. Procedures for version management should define an unambiguous way of identifying each component version. Specific versions of components may then be recovered as required for further change.

There are three basic techniques which may be used for component identification:

1. *Version numbering* The component is given an explicit and unique version number. This is the most commonly used identification scheme.
2. *Attribute-based identification* Each component has a name (which is not unique across versions) and an associated set of attributes which differs for each version of the component (Estublier and Casallas, 1994). Components are therefore identified by the combination of name and attribute set.
3. *Change-oriented identification* Each system is named as in attribute-based identification but is also associated with one or more change requests (Munch *et al.*, 1993). The system version is identified by associating the name with the changes implemented in the component.

Version numbering

In a simple version numbering scheme, the component or system name is augmented by a version number. Therefore, we can talk about Solaris 2.6 (version 2.6 of the Solaris system) and version 1.4 of component getToken. The first version may be called 1.0, subsequent versions are 1.1, 1.2 and so on. At some stage, a new release is created (release 2.0) and the process starts again at version 2.1, 2.2, etc. The scheme is a linear one based on the assumption that system versions are created in sequence. Many version management tools (see section 29.5) such as RCS support this approach to version identification.

This approach and the derivation of a number of different system versions from previous versions of the system is illustrated in Figure 29.6. The arrows in this diagram point from the source version to a new system version that is created from the source. Notice that the derivation of versions is not necessarily linear and versions with consecutive version numbers may be produced from different baselines. This is shown in Figure 29.6 where version 2.2 is created from version 1.2 rather than version 2.1. In principle, any existing version may be used as the starting point for a new version of the system.

This scheme is simple but requires a good deal of associated information management to keep track of the differences between versions and the relationships between system change proposals and versions. It may therefore be difficult to find

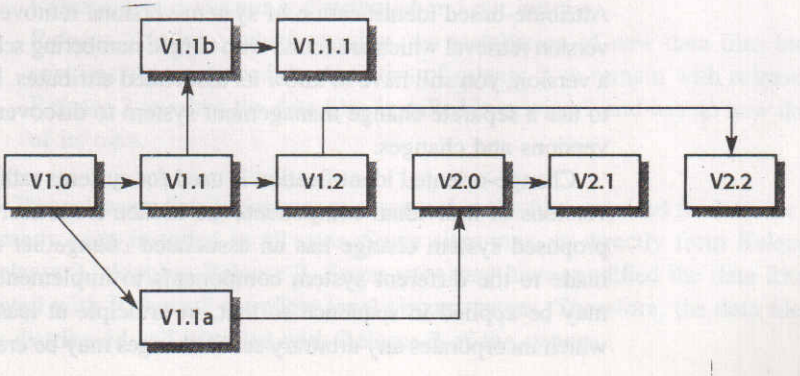

Figure 29.6 Version derivation structure

specific system or component versions when they are required, especially if there is no integrated link between the configuration database and the stored versions of the system.

Attribute-based identification

A fundamental problem with explicit version naming schemes is that they do not reflect the many different attributes which may be used to identify versions. Examples of these identifying attributes are:

- Customer
- Development language
- Development status
- Hardware platform
- Creation date

If each version is identified by a unique set of attributes, it is easy to add new versions that are derived from any of the existing versions. These are identified using a unique set of attribute values. They share most of these values with their parent version so relationships between versions are maintained. Versions can be retrieved by specifying the attribute values required. Functions on attributes support queries such as 'the most recently created version', 'the version created between given dates', etc. For example, the version of the software system AC3D developed in Java for Windows NT in January 1999 would be identified

AC3D (language = Java, platform = NT4, date = Jan1999)

Attribute-based identification may be implemented directly by the version management system. More commonly, however, it is implemented on top of a hidden version naming scheme and the configuration database maintains the links between identifying attributes and underlying system and component versions.

Change-oriented identification

Attribute-based identification of system versions removes some of the problems of version retrieval which are found with simple numbering schemes. However, to retrieve a version, you still have to know its associated attributes. Furthermore, you still need to use a separate change management system to discover the relationships between versions and changes.

Change-oriented identification is used for systems rather than components so that versions of individual components are hidden from users of the CM system. Each proposed system change has an associated *change set* that describes the changes made to the different system components to implement that change. Change sets may be applied in sequence so that, in principle at least, a version of the system which incorporates any arbitrary set of changes may be created. Therefore, no explicit

version identification is required. The CM team interacts with the version management system indirectly through the change management system.

In practice, of course, it isn't possible to apply arbitrary sets of changes to a system. Different change sets may be incompatible so that applying change set A followed by change set D may create an invalid system. Furthermore, change sets may conflict in that different changes affect the same code of the system. To address these difficulties, version management tools which support change-oriented identification allow system consistency rules to be specified which limit the ways in which change sets may be combined.

29.3.2 Release management

A system release is a version of the system that is distributed to customers. System release managers are responsible for deciding when the system can be released to customers, managing the process of creating the release and the distribution media and documenting the release to ensure that it may be re-created exactly as distributed if this is necessary.

A system release is not just the executable code of the system. The release may also include:

1. *configuration files* defining how the release should be configured for particular installations;
2. *data files* which are needed for successful system operation;
3. *an installation program* that is used to help install the system on target hardware;
4. *electronic and paper documentation* describing the system;
5. *packaging and associated publicity* which have been designed for that release.

Release managers cannot assume that customers will always install new system releases. Some system users may be happy with an existing system version. They may consider that it is not worth the cost of changing to a new release. New releases of the system cannot, therefore, depend on the existence of previous releases. Consider the following scenario:

1. Release 1 of a system is distributed and put into use.
2. Release 2 follows which requires the installation of new data files but some customers do not need the facilities of release 2 so remain with release 1.
3. Release 3 requires the data files installed in release 2 and has no new data files of its own.

The software distributor cannot assume that the files required for Release 3 have already been installed in all sites. Some sites may go directly from Release 1 to Release 3, skipping Release 2. Some sites may have modified the data files associated with Release 2 to reflect local circumstances. Therefore, the data files must be distributed and installed with Release 3 of the system.

Figure 29.7 Factors influencing system release strategy

Factor	Description
Technical quality of the system	If serious system faults are reported which affect the way in which many customers use the system, it may be necessary to issue a fault repair release. However, minor system faults may be repaired by issuing patches (often distributed over the Internet) that can be applied to the current release of the system.
Lehman's fifth law (see Chapter 27)	This suggests that the increment of functionality which is included in each release is approximately constant. Therefore, if there has been a system release with significant new functionality, it may have to be followed by a repair release.
Competition	A new system release may be necessary because a competing product is available.
Marketing requirements	The marketing department of an organisation may have made a commitment for releases to be available at a particular date.
Customer change proposals	For customised systems, customers may have made and paid for a specific set of system change proposals and they expect a system release as soon as these have been implemented.

Release decision-making

Preparing and distributing a system release is an expensive process, particularly for mass market software products. If releases are too frequent, customers may not upgrade to the new release; if they are infrequent, market share may be lost as customers move to alternative systems. This, of course, does not apply to customised software developed specially for an organisation. However, for this type of software, infrequent releases may mean increasing divergence between the software and the business processes that it is designed to support.

Decisions on when to release a new version of the system should be governed by technical and organisational considerations, as shown in Figure 29.7.

Release creation

Release creation is the process of creating a collection of files and documentation which include all of the components of the system release. The executable code of the programs and all associated data files must be collected and identified. Configuration descriptions may have to be written for different hardware and operating systems and instructions prepared for customers who need to configure their own systems. If machine-readable manuals are distributed, electronic copies must be stored with the software. Scripts for the installation program may have to be

written. Finally, when all information is available a master release disk is prepared and handed over for distribution.

The normal distribution medium for system releases is now CD-ROM disks which can store up to 600 Mbytes of data. In addition, many software products are also released by making them available on the Internet and allowing customers to download them. However, many people find the time taken to download large files is too long and prefer CD-ROM distribution.

Release documentation

When a system release is produced, it must be documented to ensure that it can be re-created exactly in future. This is particularly important for customised, long-lifetime embedded systems. Customers may use a single release of these systems for many years and may require specific changes to a particular software release long after its original release date.

To document a release, you have to record the specific versions of the source code components which were used to create the executable code. You must also keep copies of the source and executable code and all data and configuration files. You should also record the versions of the operating system, libraries, compilers and other tools used to build the software. These may be required to build exactly the same system at some later date. In these cases, copies of the platform software and tools may also be stored in the version management system.

29.4 System building

System building is the process of compiling and linking software components into a program which executes on a particular target configuration. When you are building a system from its components, you have to think about the following questions:

1. Have all the components which make up a system been included in the build instructions?
2. Has the appropriate version of each required component been included in the build instructions?
3. Are all required data files available?
4. If data files are referenced within a component, is the name used the same as the name of the data file on the target machine?
5. Is the appropriate version of the compiler and other required tools available? Current versions of software tools may be incompatible with the older versions used to develop the system.

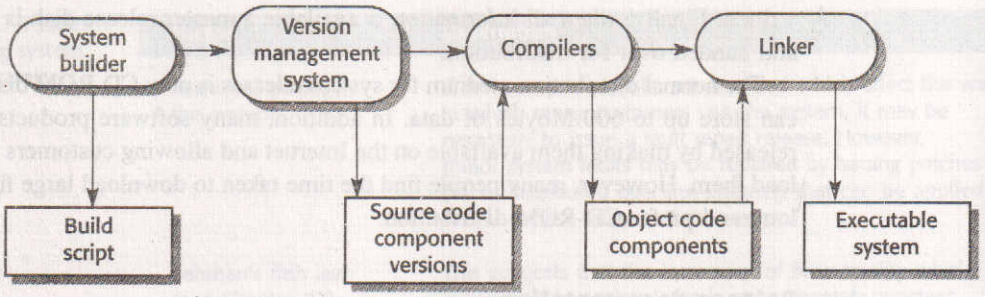

Figure 29.8
System building

Nowadays, CM tools are used to automate the system building process. The CM team writes a build script that defines the dependencies between the different components of the system. It also specifies the tools used to compile and link the system components. The system building tool interprets the build script and calls other programs as required to build the executable system. This is illustrated in Figure 29.8.

Dependencies between components are specified in the build script so that the system building system can decide when components must be recompiled and when existing object code can be reused. Build script dependencies are most commonly specified as dependencies between the files in which the source code components are stored. However, when there are multiple source code files representing different versions of components, it is sometimes unclear which source files were used to derive object-code components. This confusion is particularly likely when the correspondence between source and object code files relies on them having the same name but a different suffix (e.g. .c and .o).

To avoid some of the difficulties of physical file dependencies, some experimental systems, based on module description languages, have been produced (Sommerville and Dean, 1996). These use a description of the logical software structure and a mapping to a storage structure to infer the dependencies between files containing source code components. This approach reduces the scope for error and leads to more understandable descriptions of the system building process.

29.5 CASE tools for configuration management

Configuration management processes are usually standardised and involve the application of predefined procedures. They require careful management of very large amounts of data and attention to detail is essential. When building a system from component versions, a single configuration management mistake can mean that the software will not work properly. Consequently, CASE tool support is essential for

configuration management and, since the 1970s, a large number of different tools which address different areas of configuration management have been produced.

Examples of first-generation CM tools include SCCS (Rochkind, 1975) and RCS (Tichy, 1985) for revision control and make (Feldman, 1979) for system building. These are stand-alone tools that address specific activities in the configuration management process. Second-generation tools such as Lifespan (Whitgift, 1991) and DSEE (Leblang and Chase, 1987) provided some integrated CM process support but did not support all CM activities. At the time of writing, integrated CASE toolsets are available (Leblang, 1994) which support configuration planning, change management, version management and system building. However, these integrated CM toolsets are complex and expensive and many organisations still use first- and second-generation CM tool support.

29.5.1 Support for change management

Each person involved in the change management process is responsible for some activity. They complete this activity, then pass on the forms and associated configuration items to someone else. The procedural nature of this process means that a change process model can be designed and integrated with a version management system. This model may then be interpreted so that the right documents are passed to the right people at the right time.

Change management tools may therefore provide the following facilities to support the process:

1. *A form editor* that allows change proposal forms to be created and filled in.
2. *A workflow system* that allows the CM team to specify the different people who must process the change request form and the order of processing. This system will also automatically pass forms to the right people at the right time and inform the relevant team members of the progress of the change. Electronic mail is used to provide progress updates for those involved in the process.
3. *A change database* that is used to manage all change proposals and which may be linked to a version management system. Query facilities which allow the CM team to find specific change proposals are usually provided.

29.5.2 Support for version management

Version management involves managing large amounts of information and ensuring that system changes are recorded and controlled. Version management tools control a repository of configuration items where the contents of that repository are immutable (i.e. cannot be changed). To work on a configuration item, it must be checked-out of the repository into some working directory. After the work is complete, it is then re-entered in the repository and a new version is automatically created.

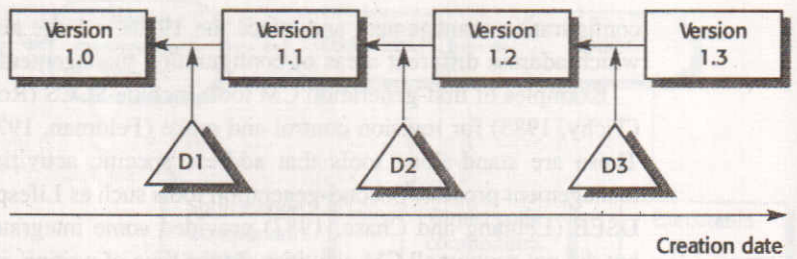

Figure 29.9 Delta-based versioning

All version management systems provide a comparable basic set of capabilities although some have more sophisticated facilities than others. Examples of these capabilities are:

1. *Version and release identification* Managed versions are assigned identifiers when they are submitted to the system. Different systems support the different types of version identification discussed in section 29.3.1.
2. *Storage management* To reduce the storage space required by different versions which are largely the same, version management systems provide storage management facilities so that versions are described by their differences from some master version. Differences between versions are represented as a *delta* which encapsulates the instructions required to re-create the associated system version. This is illustrated in Figure 29.9 which shows how backward deltas may be applied to the latest version of a system to re-create earlier system versions.
3. *Change history recording* All of the changes made to the code of a system or component are recorded and listed. In some systems, these changes may be used to select a particular system version.
4. *Independent development* Different versions of a system can be developed in parallel and each version may be changed independently. For example, release 1 can be modified after development of release 2 is in progress by adding new level-1 deltas. The version management system keeps track of components which have been checked out for editing and ensures that changes made to the same component by different developers do not interfere. Some systems only allow one instance of a component to be checked out for editing; others resolve potential clashes when the edited components are checked back into the system.

29.5.3 Support for system building

System building is a computationally intensive process. If all of the components of a large system have to be compiled and linked, this can take several hours. There may be hundreds of files involved with the consequent possibility of human error if these are compiled and linked manually. System building tools automate the build process to reduce the potential for human error and, where possible, minimise the time required for system building.

System building tools may be stand-alone tools such as derivatives of the Unix make utility (Feldman, 1979) or may be integrated with version management tools. Facilities provided by system building CASE tools may include:

1. *A dependency specification language and associated interpreter* Component dependencies may be described and recompilation minimised. I explain this in more detail later in this section.
2. *Tool selection and instantiation support* The compilers and other processing tools which are used to process the source code files may be specified and instantiated as required.
3. *Distributed compilation* Some system builders, especially those which are part of integrated CM systems, support distributed, network compilation. Rather than all compilations being carried out on a single machine, the system builder looks for idle processors on the network and sets off a number of parallel compilations. This dramatically reduces the time required to build a system.
4. *Derived object management* Derived objects are objects which are created from other source objects. Derived object management links the source code and the derived objects and only re-derives an object when this is required by source code changes.

Managing derived objects and minimising recompilation is best explained using a simple example. Consider a situation where a program called comp is created out of object modules scan.o, syn.o, sem.o and cgen.o. For each object module, there exists a source code module called scan.c, syn.c, sem.c and cgen.c. A file of declarations called defs.h is shared by scan.c, syn.c and sem.c (Figure 29.10). In Figure 29.10 arrows mean 'depends-on'. Therefore, comp depends on scan.o, syn.o, sem.o and cgen.o, scan.o depends on scan.c, etc.

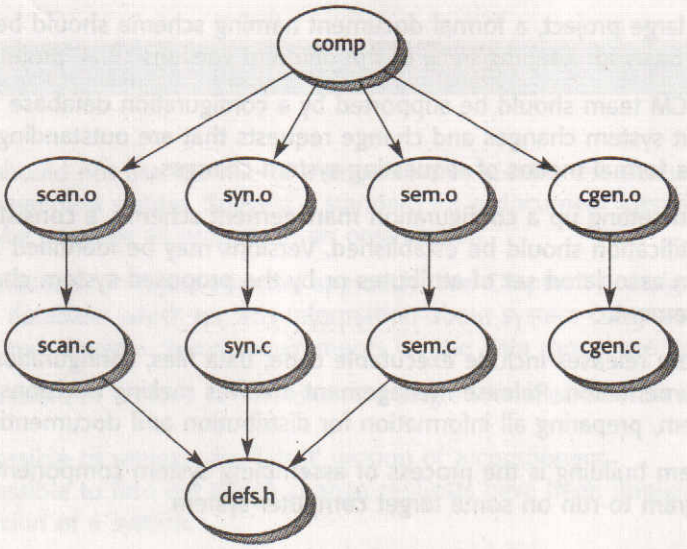

Figure 29.10 Component dependencies

If scan.c is changed, the system building tool can detect that the derived object scan.o must be re-created and calls the appropriate compiler to compile scan.c to create a new derived object, scan.o. It then uses the dependency link between comp and scan.o to detect that comp must be re-created by linking scan.o, syn.o, sem.o and cgen.o. The system can detect that the other object code components are unchanged so recompilation of the associated source code is unnecessary.

Some system builders use the file modification date as the key attribute in deciding whether or not recompilation is required. If a source code file is modified after its corresponding object code file then the object code file is re-created. However, this normally means that there is a derived object only for the most recent source code version. If an earlier version of the source code must be recompiled, then its modification date has to be set artificially to force the system builder to recognise the need for recompilation. Other systems use a more sophisticated approach to derived object management. They tag derived objects with the version identifier of the source code and, within the limits of the storage capacity, they maintain all derived objects. Therefore, it is usually possible to recover the object code of all versions of source code components without recompilation.

KEY POINTS

- Configuration management is the management of system change. When a system is maintained, the role of the CM team is to ensure that changes are incorporated in a controlled way.
- In a large project, a formal document naming scheme should be established and used as a basis for keeping track of the different versions of all project documents.
- The CM team should be supported by a configuration database that records information about system changes and change requests that are outstanding. Projects should have some formal means of requesting system changes.
- When setting up a configuration management scheme, a consistent scheme of version identification should be established. Versions may be identified by version number, by an associated set of attributes or by the proposed system changes which they implement.
- System releases include executable code, data files, configuration files and documentation. Release management involves making decisions on when to release a system, preparing all information for distribution and documenting each system release.
- System building is the process of assembling system components into an executable program to run on some target computer system.

- CASE tools are available to support all configuration management activities. These include tools such as RCS to manage system versions, tools to support change management and system building tools.
- CASE tools for CM may be stand-alone tools supporting change management, version management and system building or may be integrated systems which provide a single interface to all CM support.

FURTHER READING

System Configuration Management. This is the latest proceedings of a series of workshops on software configuration management. It summarises the latest research and industrial practice in this area. (J. Estublier, 1999, Springer.)

Trends in Software: Configuration Management. This is a collection of papers on different aspects of configuration management by authors who are active researchers and practitioners in this field. It's a good introduction for students and practitioners who are interested in reading on advanced CM topics. (W. Tichy (ed.), 1995, John Wiley and Sons.)

Implementing Configuration Management. This book is written from the perspective of a large system supplier. It discusses the problems of system configuration management where hardware, software and embedded firmware must all be controlled. (F. J. Buckley, 1993, IEEE Press.)

EXERCISES

29.1 Explain why you should not use the title of a document to identify the document in a configuration management system. Suggest a standard for a document identification scheme that may be used for all projects in an organisation.

29.2 Using the entity-relational or object-oriented approach (see Chapter 7), design a model of a configuration database which records information about system components, versions, releases and changes. Some requirements for the data model are as follows:

- It should be possible to retrieve all versions or a single identified version of a component.
- It should be possible to retrieve the 'latest' version of a component.
- It should be possible to find out which change requests have been implemented by a particular version of a system.

- It should be possible to discover which versions of components are included in a specified version of a system.
- It should be possible to retrieve a particular release of a system according to either the release date or the customers to whom the release was delivered.

29.3 Using a data-flow diagram, describe a change management procedure which might be used in a large organisation concerned with developing software for external clients. Changes may be suggested from either external or internal sources.

29.4 Describe the difficulties which can be encountered in system building. Suggest particular problems that might arise when a system is built on a host computer for some target machine.

29.5 With reference to system building, explain why it may sometimes be necessary to maintain obsolete computers on which large software systems were developed.

29.6 A common problem with system building occurs when physical file names are incorporated in system code and the file structure implied in these names differs from that of the target machine. Write a set of programmer's guidelines which help avoid this and other system building problems which you can think of.

29.7 Describe five factors which must be taken into account by engineers during the process of building a release of a large software system.

29.8 Describe two ways in which system building tools can optimise the process of building a version of a system from its components.