Technical Report
CMU/SEI-92-TR-031
ESC-TR-92-031
November 1992

Analysis of a Software Maintenance System: A Case Study

Howard M. Slomer
Alan M. Christie

Technical Report
CMU/SEI-92-TR-031
ESC-TR-92-031
November 1992

Analysis of a Software Maintenance: A Case Study

Howard M. Slomer
U.S. Department of Defense

Alan M. Christie
CASE Environments Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center / 8725 John J. Kingman Road / Suite 0944 / Ft. Belvoir, VA 22060-6218. Phone: (703) 767-8222 or 1-800 225-3842.]

Table of Contents

List of Figures

Analysis of a SoftWare Maintenance System: A Case Study

Abstract: To design, implement, and operate a successful software development process, exposure to similar existing systems is invaluable. The objective of this paper is thus to document and analyze an existing, moderate size, software maintenance project. The project, which supports the maintenance of a software environment has, through incremental improvement, become very effective. However, this effectiveness has only been achieved through struggle, compromise, and creativity. The paper documents the evolution of the project, providing insights into how change was managed, and defines and formally models the project as it existed until recently. The project's process is still evolving, and recent changes, while not formally modeled, are also described.The results of this modeling are applied 1) to compare the project's practices from a perspective of the SEI Capability Maturity Model (CMM) [Paulk 91, Weber 91] and 2) to address briefly the issue of process reuse. Comparison to the CMM resulted in an identification of strengths and weaknesses of the project's software process. In the examination of reuse issues, three hypothetical examples of process reuse are examined.

1 Introduction

This report describes how software maintenance is performed within a project supported by the U.S. Department of Defense[1]. The software environment, which is supported and maintained by the project, is designed to process large amounts of textual information and to retrieve information from dissimilar remote systems. It provides a set of Unix tools which includes: a high-performance distributed editor, a data base, a forms generation package, a mail system, application programs and standard user interface. The project also supports a large collection of library modules (about 1000) that are used both internally and by other software development organizations.The project maintenance system is now quite mature, supporting approximately 350,000 lines of code and having processed to date approximately 2000 change requests.

As background, this report first provides a history of the project from its inception, identifying issues that arose and describing how those issues were resolved (Section 2). The report then describes the project environment both textually and using a more formal model. The textual description is useful for gaining an overall qualitative understanding and for providing insights into the project's operation (see Section 3). The formal model of the project software process (described in Section 4) provides three benefits. First, it graphically defines roles, activities, products and so forth and their relationships. It also defines communication links in the process and conditions under which activities can occur. Such a formal model provides a level of

[1] For security reasons the project cannot be named, and hence in this document it will simply be referred to as "the project."

precision and rigor for defining the process which is difficult to attain through textual description. Second, such a model is useful as a basis on which to start process improvement. Often what is ineffective in a process is clear, but having an agreed upon definition of the process provides a means for communication about change. Finally, defining a formal model of the process provides a vehicle for training new members and providing a corporate memory in “how things are done.” It provides a means for new project members to rapidly understand and visualize the environment in which they will be working, and it establishes continuity of process when experienced members leave the project. Some insights into the operation and the directions where these insights are currently leading the project are also discussed (see Section 5).

As a by-product of developing a detailed project process model, it was evident that it would be a simple but effective exercise to contrast the configuration management practices of the project with those of the Capability Maturity Model [Paulk 91]. This comparison resulted in some interesting conclusions with respect to the strengths and weaknesses of the project’s process (see Section 6). Through a formal model, parts of the process may be identified as candidates for reuse. These process building blocks whose success has been established (at least within the project) may be useful to others who are considering the installation of a similar environment. Some of these process elements may come with supporting scripts, thus allowing rapid tailoring of a new, automated environment (see Section 6).

The project software environment was built in-house on top of the Unix operating system [Sobel 89]. One issue which is often encountered is whether to build such an in-house system or to buy an external product and tailor it to the needs of the organization. Having experienced the “build” option, the project can lend insight into the pros and cons of this approach (Section 7).

Appendix A provides a brief description of the graphical terminology used by the process models shown in Section 4. The tools and artifacts which were developed to support the smooth functioning of the process are also described in some detail. Details of these artifacts and outlines of how these tools were implemented, through Unix scripts, are also given. (See Appendices B through F.)

2 A Brief History of the Project

2.1 The Early Years

The project was originally designed to provide a set of tools (editor, mail system, database, forms package and others) to process large amounts of textual information and to process and retrieve information from dissimilar remote systems. To provide this capability, multiple requirements were initially imposed on the system design. Some of these are listed below:

- The application programs were required to provide multiple language support. All project programs, including the text editor, had to support the extended ASCII character set.
- The project text editor, which is the common user interface to many of the other programs, was required to provide eye-blink response (less than one second) without being constrained by file size.
- The project was required to provide an environment which isolated users from the operating system and, when necessary, from each other.
- The project hardware systems were required to transfer files in both directions to other project systems and between the project systems and mainframe systems such as the Cray and IBM 360.

The multiple language requirement was satisfied by using the eighth bit of each byte to extend the supported character set to 256. This performance requirement was met by designing the editor to a) distribute the processing between the terminal and the computer, and b) to read ahead in either direction anticipating the user's need, and transferring the next block of text to the terminal before it is required. The security requirement was met by replacing the Unix shell (command interpreter) with one which provided a limited view of the system to the user, based on the security requirements. All tools enforce a multi-layered security algorithm based on the user's "need-to-know" and on the terminal's location. Commercial products were not available at the time which met any of the operational requirements.

The project was first developed on a DEC PDP-11/70 running a modified Unix kernel and Delta Data 3000 terminals. The software was logically divided into two major functional sections: terminal software and host computer software.

For several years the project did not implement formal configuration management procedures. Even simple Unix tools like SCCS[2] and MAKE were not used. The source files, compile scripts, and install scripts were stored in a single tree structure of Unix directories. The developers had complete access to all sources and made changes in place.

A new release was generated by copying the current state of the development tree to an empty file system and then compiling and linking everything. When the entire set of programs was compiled and built successfully, a tape was made. This was designated as the new "release." Backups were made daily to prevent potential disasters.

[2] SCCS is the name of the source code control system which runs under Unix.

When serious problems were discovered between releases, they were fixed immediately and the new versions of the programs distributed to users. New software and upgrades to existing programs were advertised and distributed between releases. No attempt was made to employ version control or to provide a means of identifying which versions of sources were used to produce a given executable. Problems were fixed by installing the latest versions of the program in development and working from that point.

This “system” worked well for several years. Problems were fixed quickly, new releases were generated at will, and in general the project’s customers were happy.

There were other factors which contributed to the success of the project:

- The project leader was a highly skilled computer scientist who worked with the development team and was personally responsible or coding many of the most complex modules.
- The team was small and worked well together to solve difficult problems.
- Everyone informally knew which of the project files the other team members were changing and automatically avoided potential CM problems.
- Technical decisions were openly discussed and everyone had a chance to contribute.

As the project continued to grow and expand, however, the lack of formal configuration management started to cause real problems.

New host computers (AT&T 3b20 and 3b15) and terminals (IBM AT) were introduced, while the old systems (PDP-11/Delta Data 3000) continued to be used by one of the major operational groups. The developer faced the problem of porting the software to new hardware platforms while continuing to support the old systems. Enhancements and problem fixes had to be applied to multiple versions of the software. The situation improved somewhat when life cycle support for the old systems was transitioned to the operational organization. The old and new systems were tightly coupled across a network, which meant that many of the software changes had to be coordinated. The problem changed from maintaining two sets of software on dissimilar platforms to that of coordinating changes between two separate organizations.

New network technology (TCP/IP) was introduced with the new hosts. Since several of the original systems were still in use and needed to interface with new systems, the old network software was ported to the new systems. Both sets of network code and the application programs that interfaced to them had to be maintained.

As high performance workstations became available, many of the IBM AT terminals were replaced with terminals from Sun Microsystems. The developers were not only faced with porting to a new variation of the Unix operating system but also to one that was a moving target. Unlike other computer manufacturers, Sun frequently introduced new, incompatible versions of their operating system directly to the user community. The team now faced the problem of maintaining multiple versions of software on the same hardware platform. Introduction of high performance Sun 3 and SPARC workstations blurred the distinction between the host and terminal functions. project users now wanted the traditional host software (data base, forms generation package, mail and other applications) ported to the new work stations.

Several of the original developers resigned from the project and were replaced by less experienced people. The size of the development team was increased to try to cope with the new development and maintenance problems. As a result, communication between developers started to become a problem.

The project was doomed unless a way to manage change could be found.

2.2 Configuration Management Arrives

The development team and management held a series of meetings to determine what could be done to solve the problems. All agreed that a formal system of managing change and controlling software build and release must be introduced. A fundamental decision was made that the developers would not exercise configuration management on their own software. A new work center was established to manage configuration control. After several commercial and government systems were examined, the new CM group decided that none was suitable and that they would have to develop their own programs and procedures. The CM system now in place evolved over several years and continues to evolve. The basic requirements for the new CM system were not formally documented. They evolved during a series of heated meetings between the new work center chief and the development team. The following list represents some of the important requirements of the proposed system:

- The CM system must be as non-intrusive as possible; development and maintenance cannot stop while it is being installed.
- A way of identifying programs must be part of the new system. All programs must be traceable to a formal release-version number. Additionally, all programs must be traceable to the versions of the sources, include files and library modules which were used to generate them.
- A configuration control board (CCB) must be established and meet weekly to approve all changes. A formal, automated method of submitting change requests must be implemented.
- Changes approved by the board must be traceable from executables to specific versions of all files (source files, include files, makefiles, compile scripts, external #ifdefs, etc.).
- An independent test team must be assigned to do quality assurance. A new release would not be distributed without their approval.
- A CM baseline repository containing all approved items must be established. All new baselines had to be generated from the repository and not from the development system.

2.3 CM Continues to Evolve

The CM software and procedures are not static but are continually evolving to adapt to the changing needs of the project. Prior to suggested changes, CM and development managers carefully evaluate possible alterations to the CM system. Some of the developers view many of the CM practices and procedures as a hinderance to development and tend to resist changes. Modifications which do not directly benefit the developers or ones which divert significant development resources tend to be resisted.

One issue which arose between the developers and configuration managers was whether or not to drop the old compile and build scripts in favor of the Unix MAKE utility. The developers argued that MAKE was not consistent across platforms and that to adopt MAKE would add unnecessary maintenance problems. The compromise reached was to adopt MAKE but retain all compile and build functions in the compile/build scripts. MAKE was thus used to determine dependencies and to start the compile scripts which then performed the software build function.

A second issue which arose between the two teams involved the format of the files delivered to the STAGE tree (see Section 3.4 for a description of this tree). The CM manager wanted the items delivered to STAGE as ASCII files that the CM team would then check in to their version of the tree in the CM repository. The development manager argued that the delta function was too important to delegate and should remain a developer function, and that the updated SCCS files should be delivered to the STAGE tree. The option proposed by the CM organization was adopted. After a short period of time mistakes made by the CM team caused inconstancies to develope between the repository and development SCCS files. The original decision was reversed; the SCCS versions of the files are now delivered to the CM team.

In summary, the project evolved from a single platform, single customer system into areas that the original developers could not have predicted. The customer base grew as more organizations adopted the system. The number of supported platforms grew with the increased customer base and with the evolution to inexpensive computer hardware. As the distinction between terminals and computers became blurred by the introduction of high performance workstations, much of the original software had to be redesigned. All of these changes highlighted the need for improved change management. The configuration management system adopted was designed to support current needs and to be flexible enough to grow with the project. The crucial decision which led to the current effectiveness of the process was the separation of configuration management activities from those of the developers. Without this reorganization, the complexity of the process would probably have led to the demise of the project.

3 An Overview of the Project Software Process

The current project software process has several major features. First, requested changes, change authorization, change status, and technical approach are all documented in the change request form (see Appendices E and F). The CR form thus provides an “information backbone” to the whole change process. Second, a change control board is central in authorizing all proposed changes. Third, configuration management of all revised software is the responsibility of an independent CM group, rather than the software development group. Finally, the build process is performed in the same manner by both the development and CM groups using a consistent set of Unix scripts which allow for a high degree of automation and portability (see Appendices B, C, and D).

3.1 The Change Request Cycle

Problem reports and requests for new features usually originate with the users. A developer is assigned to work with the user to determine if the problem is real and if it is a software related problem. The developer then initiates and submits a formal change request (CR) to the CCB. Change request generation is automated through a series of programs and shell scripts. The change request program provides a template containing a series of fields which must be filled out by the developer. The software assigns a unique number to the change request which will be used to track the CR through its entire life cycle. The CCB change request then provides two-way formal communication between the board and the developers. The software is implemented by two programs:

- ccbnew (generate an empty template and assign a unique number to it)
- ccbold # (retrieve for modification a previously generated CCB change request)

The CCB meets once a week to consider all new requests received during the week and to take action on old requests which have not been finalized. CCB change requests may cycle between the developer and the board several times before final disposition. A typical cycle will start with a report of a problem with a proposed solution. The board may reject the change request because it does not contain sufficient information or because the board doesn’t agree with the solution. The developer may then re-submit the request and provide additional information or possibly an alternate solution. When the board finally approves the change, the status will change to *be done*.

Once the CCB request is approved for development, a developer is identified by the project chief to make the changes. The developer uses the Unix **get** command to **checkout** the items needed from the development tree, makes and tests the changes, and then uses the Unix **delta** command to **checkin** the changes to the SCCS version of the file(s). The original number assigned to the change request by the ccbnew program must be entered as the Modification Request (MR) number required by SCCS to do the check in. This feature provides traceability between change requests and versions of the software.

On completion of the changes, the developer updates the CR form by supplying the list of modified items and any other missing information. The developer then requests the CCB to change the status of the CR from *to be done* to *include*. Once approved, the developer moves the new SCCS files to a holding directory (called STAGE).

The CM branch picks up the modified SCCS files from STAGE and does extensive accounting checks. They make sure that all items in the change request have been moved to STAGE and that the version numbers of the SCCS files match those specified in the CCB change request. They also verify the MR numbers in the delta(s) match the CCB change number associated with the request.

Finally the SCCS files are moved 1) to the CM baseline repository, where files under configuration management are stored and 2) to a directory called BASE on the development system which allows the development team to examine, in read-only mode, the files under CM control. This latter feature allowed the developers to cross check that released software is consistent with their intentions.

3.2 The Change Control Board

A configuration control board was established with permanent members consisting of the deputy division chief, work-center chiefs, and project leaders. The CCB reviews all change requests, determines which will be accepted, assigns priorities, and monitors progress by the development team. The meetings are open to all, and developers are encouraged to attend in order to defend proposed changes. The results of the meetings are e-mailed to all staff members. The CM team uses the published results of the meeting to audit and control the items which are moved from development to the baseline system.

3.3 Configuration Management

The CM team uses the published results of the CCB meetings to determine which items should be moved from the development system to the CM baseline repository. They make sure that all items listed in the CCB change request have been moved to the STAGE area of the development system. They audit the SCCS files to assure that the correct versions have been provided and that the SCCS Modification Request (MR) numbers match the CCB change request number. When the auditing checks have been completed, the new items are installed on the CM machine and inserted into the CM baseline repository. The items are then moved back to the BASE tree on the development system. Summary files are produced which indicate the status of all CRs.

All application source code, library source code, include files, makefiles, and compile scripts are under SCCS control. All items contain an imbedded string which will be expanded by SCCS to produce the following:

- SCCS release, level and branch
- Delta (check-in) date
- Formal release number associated with this delta

During the build process the above information is carried from items to objects to executables. When applied to an executable, the Unix **what** command produces a list of all items that make up the program. The version numbers of each item can then be used to trace back to the CCB reports which identify why and when changes were made. The CCB report will also yield a cross-reference list of all other related items which were modified to satisfy the same change request.

The tools (**what**, **grep**, **sh**, **awk** etc.) used for identification and tracking are available on all Unix platforms.

The CM baseline repository is stored on a separate baseline machine. Access to this machine is limited to those responsible for maintaining and updating the baseline. Software changes must be approved by the CCB and audited by the CM team prior to their movement to the baseline machine. All software released for testing and distribution is built from the sources on the baseline machine.

3.4 Unix Directory Structure

All items used to generate the project programs are stored in a single tree structure of Unix directories. A subset of a typical tree is shown in Figure 4.17. The components stored in the tree include: makefiles, compile scripts, and source code. In addition to these basic components, each node in the tree contains a SCCS directory. The SCCS directory holds the SCCS files for all the items stored at the node. The tree structure is used to separate items into logically related groups of project programs. Large programs like the database and distributed editor are stored in major branches of the tree, while a number of smaller related programs might be siblings of a node at the same level. The executables at any point in the tree can be generated by invoking the Unix **make** command at that point in the tree. The entire system can be rebuilt by invoking **make** from the root of the tree.

The top level of this tree is used to divide the programs into broad subsystems (network, editor, library, system etc.). Each additional level is used to further divide these subsystems. Programs may be generated from single or multiple source files. All sources for a program may be at one level of the tree or may be stored at multiple levels at or below the same level. Each node of the tree contains at least one file (makefile) which is responsible for making executables at that node and/or starting the make procedure at all levels immediately below it.

The entire system can be build by running the Unix **make** command from the top of the tree. (See Appendix C for an example of a make parameter file.) Each **make** is responsible for building items at the node in which it is installed and for starting the **make** process at all nodes immediately below it. Individual items can be made by running **make** at the appropriate node.

Compile scripts (see Appendix D) located at each node are executed by the makefiles. The compile scripts determine the platform by consulting a single data file at the root directory (Appendix B) and adjust platform-dependant parameters (compile flags, ifdef parameters, link flags, location of executable, ownership of executable, mode of executable, etc.) which control the build and installation of executables.

A new release on a given hardware platform is generated by moving the baseline tree structure to the target machine and doing a make from the top of the tree. To build the system on a different platform, the entire tree is moved to that platform and the process repeated.

3.5 System Structure and Components

There are multiple instances of the tree structure described on Section 3.4. These trees are called DEVEL, STAGE, BASE and the CM baseline repository (see Figure 3.1).

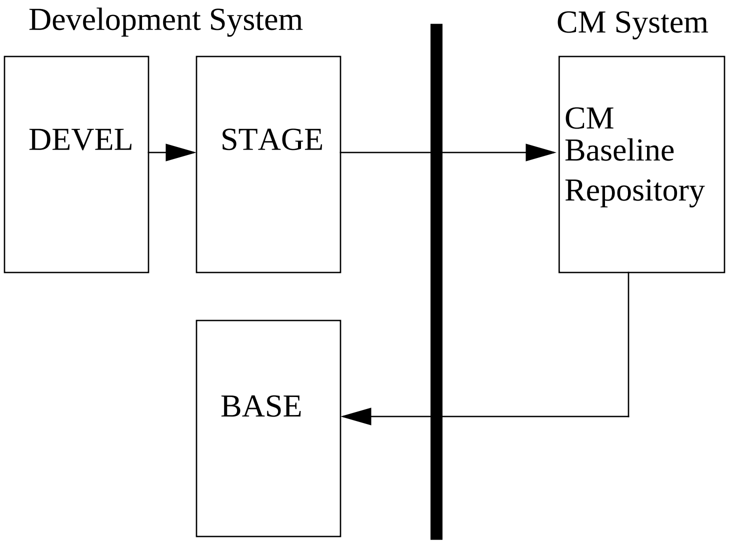


Figure 3-1 Relationship between Project Databases

All modification and testing are carried out in the DEVEL tree. Communication between team members working in the DEVEL tree is generally very good; each developer knows where other team members are working and in general overlapping development is not a problem. Since development is accomplished directly in the tree and not in separate work spaces, two techniques are employed to help synchronize change. First, when developers are assigned work they immediately change the ownership of the items under development to themselves. Ownership by another developer indicates that someone else is looking at a problem involving the items. However, this does not prevent others from accessing these items; it is only a form of notification. Second, SCCS provides a primitive form of file locking by not allowing a second *get for update* when a change is pending. An SCCS lock indicates that the code is actually *checked out* for modification.

The second instance of the tree (STAGE) is usually empty -- it is used as a staging area between the developers and the CM team. STAGE acts as a buffer between the teams and allows development to continue while the CM managers audit and move completed items to the CM repository. The developers move completed items to STAGE where the CM team, using the completed CCB change request as a guide, audit and then move the newly approved items to the CM machine. Auditing is done to check consistency between the contents of the CCB change request and the software delivered by the developer.

The third instance of the tree (BASE) contains a copy of the current release plus all approved changes. Once the CM team audits a change and moves the new versions to the CM repository, the items are moved back to this tree. The developers can verify that their changes were successfully transferred to the repository and that the development tree is current before they proceed with additional changes.

The forth instance of the tree is the CM baseline repository which is kept on the baseline machine. The BASE copy on the development system is a mirror image of the baseline repository.

3.6 User Support

3.6.1 Upper Management

The CM system provides management with an overview of the status of all change requests. A summary program scans the CCB change request forms and provides a formatted output of the current state of all change requests. The agenda for the weekly CCB meetings is determined from the list. All new items and those which require CCB action are scheduled for discussion. The list of topics is e-mailed to CCB members.

3.6.2 Project Management

Project managers also have access to the CCB summary files, and they can use them to determine the current state of all outstanding change requests. Details of a particular change request can be determined from the specific CCB change request.

3.6.3 Developer

The CM system provides developer support in several key ways.

Communication between developers and program managers is automated by the CM system. A copy of each CR form, when submitted or updated, is automatically mailed to everyone involved with the project.

Maintenance has been improved substantially by the combination of automatically embedding SCCS identification strings in the programs and by the tight linkage between CCB change requests, the versions of the source files, and the executables.

The build procedures have been completely automated by makefiles and compile scripts. The build procedures determine which machine platform the item is being built on and automatically makes adjustments to the compile, build, and install parameters.

Good communication and co-ordination between developers and program managers is provided via e-mail by the CM system. Communication between developers and the CCB is through the CCB change request. Copies of all transactions are mailed to everyone on the project.

3.6.4 Tester

Formal testing of all changes is deferred until a new release is ready for distribution. The testing organization uses information derived from the CCB change requests (test scenarios and release notes) to unit-test all changes. Copies of the new release are then given to operational elements to beta-test prior to distribution.

3.6.5 Customer Representative

Release notes which are derived from the CCB change requests are distributed with new versions of the system. The history provided by the CCB change requests supports those responsible for customer interaction.

4 The Change Cycle Illustrated

This section provides a detailed formal definition of the project software process and uses the graphical modeling representation described in Appendix A. While reading the process diagrams is not difficult, a better understanding of them will result if Appendix A is consulted first. In addition, because the focus of this section is a formal and detailed definition of the process, a review of the process overview in Section 3 is advisable. In building the model, the intent was to capture both the human and machine processes and the interaction between them. This intent defines the model's scope.

4.1 How to Read the Model

The following diagram (Figure 4.1) gives a roadmap through the project process model. Since the model is multi-leveled, it can become confusing without such an overview. Each box in Figure 4.1 provides both the title of the process component and the figure number of the detailed process model corresponding to that component. This road map thus guides the reader through the diagrams and descriptions in the rest of this section.

Before beginning the detailed descriptions, some conventions should be discussed. First, any entities which are to the extreme left edge of a diagram provide information which comes from outside the diagram. In a similar way, any entities which are to the extreme right edge of the diagram send information out of the diagram, and by implication, on to other diagrams. This *parameter passing* is analogous to the way in which variables get passed through subroutine parameter lists. Using this subroutine analogy further, there is one entity type whose value is assumed to be globally visible. This is the "Unix tree." Instances if this entity type are, for example, the DEVEL tree and the CM baseline repository tree. Any changes that are made to the content of these trees are assumed to be globally visible without the corresponding variable being explicitly passed through the left or right edge of a process diagram. In reading the diagrams, it is best to start with these entities linked to the left hand edge. These entities will initiate certain activities, which themselves will generate new exit entities (e.g., products, conditions). A review of the formalism in the diagrams is provided in Appendix A.

Two other conventions should be noted. First, it is sometimes necessary to specify the same entity twice on the same diagram, otherwise the clutter of links between entities would reduce clarity. For example, in a diagram, the agent *developer* is commonly used, and it may be necessary to define this agent in two separate locations as *developer/1* and *developer/2*. Second, in the text below, the names of activities defined in the process diagrams are in a bold, italicized font (e.g. *i**nstall modifications***).

4.2 The Top-Level View

Figure 4.2 shows a top-level view of the project process. A change (bug fix or software enhancement) may be initiated either by external users of the software environment or by project support staff (primarily project developers or the project chief). Throughout the change

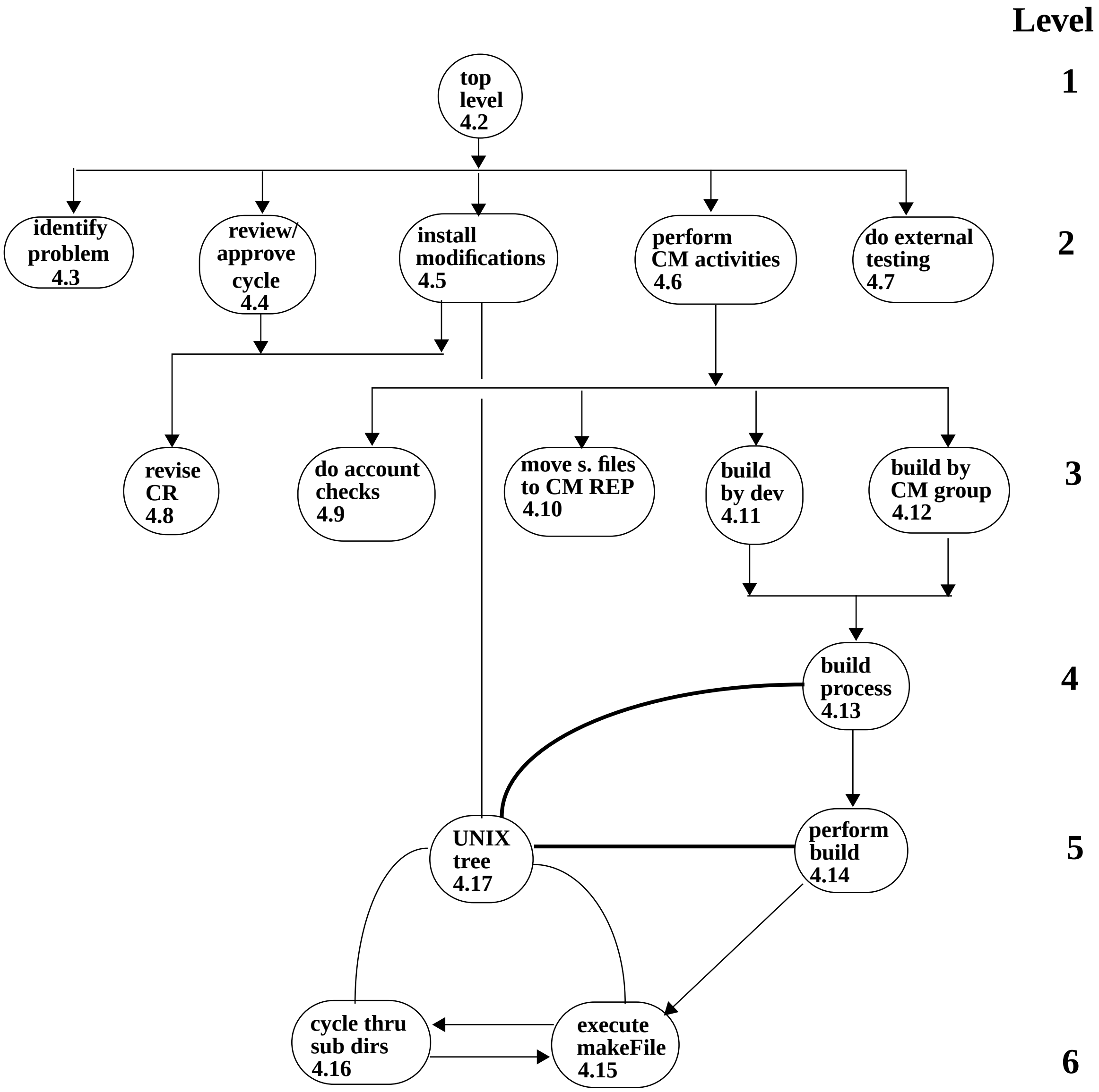
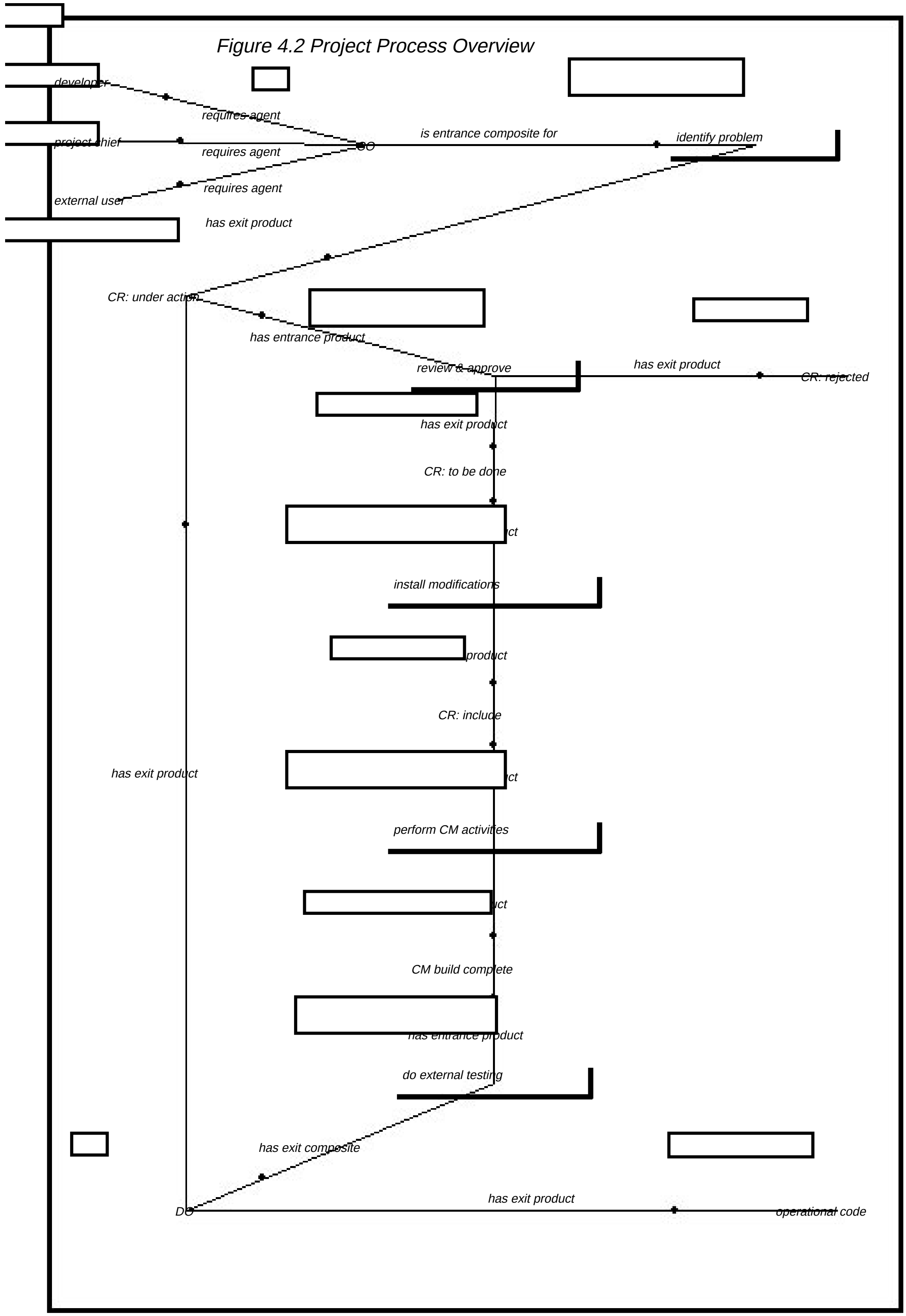


Figure 4-1 Roadmap Through the Process Model

process, the change request form is the main conduit for supplying information and history on the current change request (CR), and it is the ‘‘glue” which ties the process together. Figure 4.1 shows the five major steps in the process of developing and releasing a product. These steps are:

- **Identify problem**. This activity focuses on identifying either an external (user) problem or an internal (developer) problem and describing it in the CR.

Figure 4.2 Project Process Overview



- **Review/Approve Cycle.** Once the problem has been documented, the responsible developer proposes and records a solution. This proposed solution is added to the CR.
- **Install Modifications.** This activity focuses on obtaining approval for the proposed changes to the software (through the change control board) and modifying the software to reflect these changes.
- **Perform CM Activities.** On completion of the changes, the modified files are moved to the configuration management group. The CM group checks for the consistency and completeness of the files. A system build to obtain updated executable code is performed after multiple CRs have been collected and validated.
- **Do External Testing.** The final stage is to transfer the executable code over to an independent test group who verify that the code is ready for release. Note that as a result of testing, errors may result in a new CR being developed. This second CR then follows the same procedure as the former CR.

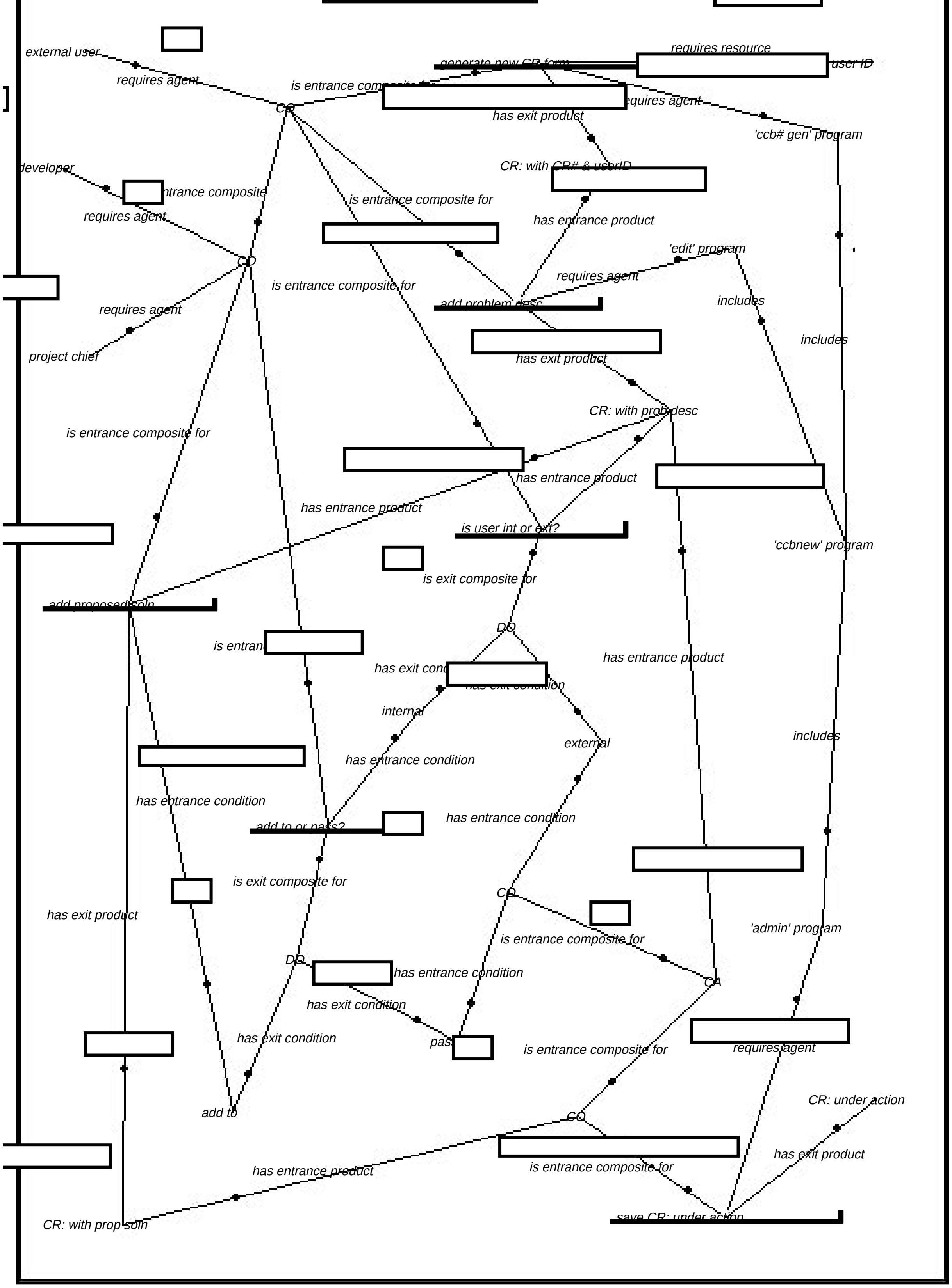
There are three formal CR status values: under action, *to be done* and *include*. In addition to these, a variety of other status values are used throughout the process model described in this section. These have no official authorization; they are used here simply to differentiate between different process states.

4.3 Identify Problem

To initiate a revision to existing software, a blank change request form is generated. The CR form is on-line and is managed by a suite of programs, coordinated through the ccbnew program. For example, when a new CR is being established, the ccbnew program brings up a blank form, inserts a unique CR number, adds the user ID of the person initiating the effort, and provides an editor to insert text describing the problem. The ccbnew program also allows saving of the newly filled-out CR form.

In Figure 4.3, activity ***generate new CR form*** automatically inserts the user ID and a unique change control board number (ccb#) into the form. The user then adds a description of the problem or enhancement *(**add problem description**)*. If the CR initiator is an internal user (i.e., either a developer or the project chief), then that user may go on to suggest how the change should be implemented. In Figure 4.3, there are two decision “activities” associated with this process. The first (***is user int or ext***) separates internal from external users. The second (***add to or pass***) discriminated between these internal users who decide to add a proposed implementation and those who do not. The final action *(**save CR: under action**)* saves the CR with the status *under action*.

Figure 4.3 Process Flow for Identify Problem



4.4 Review and Approve

Upon completion of the CR form, change control board approval is required before implementation of the changes can occur. Figure 4.4 illustrates this approval process. The review panel consists of the change control board and may involve the developer who is responsible for managing the CR prior to the changes being implemented. (A different developer may make the actual changes.)

As a result of the review, *(**review CR**)*, the CR may be approved and given the status *to be done*, it may be rejected (too costly, low priority, etc.) or it may require revision. There are a variety of reasons for making revisions. First, if the CR was filled out by an external user, the approach to implementation has not yet been defined. Second, a proposed approach may not be sufficiently detailed or be too vague. In these cases the review board will provide guidance, and this is documented through the meeting minutes. These minutes are distributed to all interested parties in electronic form.

The proposed changes are incorporated into the CR *(**revise CR**)*. The review/approve/revise cycle can be performed as many times as necessary until one of the exit criteria have been met (i.e. the CR has either the status *to be done* or *rejected*.)

4.5 Install Modifications

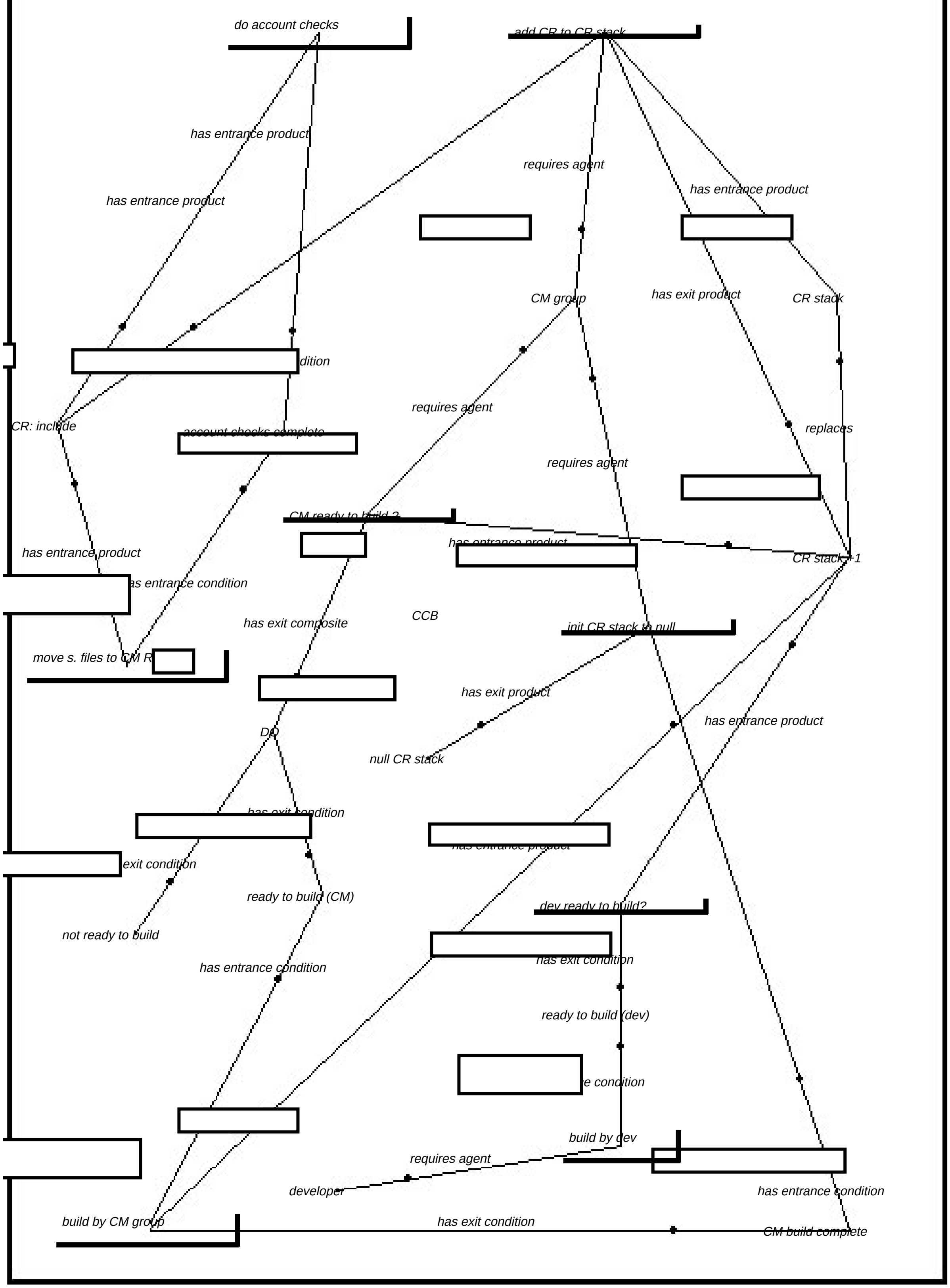
After change request approval has been granted, work on implementing the changes starts (see Figure 4.5). The project chief first identifies which developer will perform the work *(**assign developer(dev)/1**)*. That developer then makes copies of the appropriate files from the tree in which the developers work *(**check out files**)*. These files are stored in the DEVEL tree in archival *s.* format so that they have to be extracted before use. The resulting source files are placed in another branch of the DEVEL tree for private developer modification. The *s.* files automatically lock to prevent clashing. Once checked out, the code changes are made. The development sequence consists of making changes *(**make changes per CR)***, developing appropriate tests (***develop test suite***), and testing the code *(**test changes locally**)*. This cycle is repeated until the modifications have been successfully incorporated.

Once the updates have been completed, the CR is revised to reflect the changes (***revise CR***) and the revised files are incorporated back into the archival *s.* files *(**update s. files in DEVEL**)*. While saving these files, the system requests that the developer type in an *MR number*. To this, the developer inserts the number of the CR used to authorize the modifications (see ***generate new CR form*** in Figure 4.3).

4.6 Perform CM Activities

The CM group is responsible for 1) assuring the completeness and correctness of all files which have been provided by the developer group for the current CR and 2) building the code to be released. These activities are illustrated in Figure 4.6. In the first activity *(**do account checks**;* see Section 4.9), the CM group uses the change request to assure adequacy of the

Fig w for: Perform CM Activities



files in the STAGE tree. Having resolved any problems (usually through interaction with the developers), the files in the STAGE tree are transferred over to the CM repository, which has a structure identical to that of the trees previously discussed.

The software build performed by the CM group involves modifications due to many CRs. As CRs are processed, they are added to a stack which is managed by the CM group (***add CR to stack***). The CM repository is incrementally updated each time a new CR is added to the stack *(**move s. files to CM REP**;* see Section 4.10). The update cycle is continued until the CM group is ready to release the next version of the software, at which time a build is performed (***build by CM group***; see Section 4.11). A build may be initiated, for example because of number of CRs currently processed, or to release an upgrade containing a particularly important modification *(**ready to build?**)*. At any point during the accumulation of CRs, the developers may also decide to perform an informal build on the DEVEL tree *(**build by dev**;* see section 4.12), primarily to assure themselves that the CM group will not confront major problems when they perform their "official" build.

4.7 Do External Testing

The final step in the modification process is to perform independent testing before release (see Figure 4.7). Upon completion of the build, the executable code is shipped to the test group. Because the test group have other responsibilities, a significant delay can result from waiting for completion of the test phase.

The first activity performed in the test phase is the actual testing *(test code)*. If the tests succeed, then the existing code built by the CM group is ready for external delivery. If the tests fail, then a new change request must be prepared, *(develop new CR)*, which is then reviewed with the change control board, *(review with CCB)*. If the delays incurred in implementing the proposed changes are significant, and the impact of these changes is serious then the CR bypasses the normal lengthy review process and follows the accelerated process shown up the right hand side of Figure 4.7. The CR (CR: quick fix) is used by the developers to revise the code, *(develop revisions)*. The CM group then build a patch, *(build patch)*, which is added to the tape containing the original (faulty) code, *(add patch to built code)*. For less urgent fixes, the *(CR: to be done)* which exits this task, is injected into the normal CR cycle (see Figure 4.2).

4.8 Revise Change Request

A change request may be require modification as a result of a CCB review or to reflect changes in code during the development cycle. These changes are made using the "ccbold" program, which supports a variety of functions. Figure 4.8 illustrates the activities involved in revising a CR. The following is a list of these functions:

- Retrieve a copy of an existing CR form from its s. history file.
- Add the ID of the user who is doing the modification to the CR.
- Edit the CR.

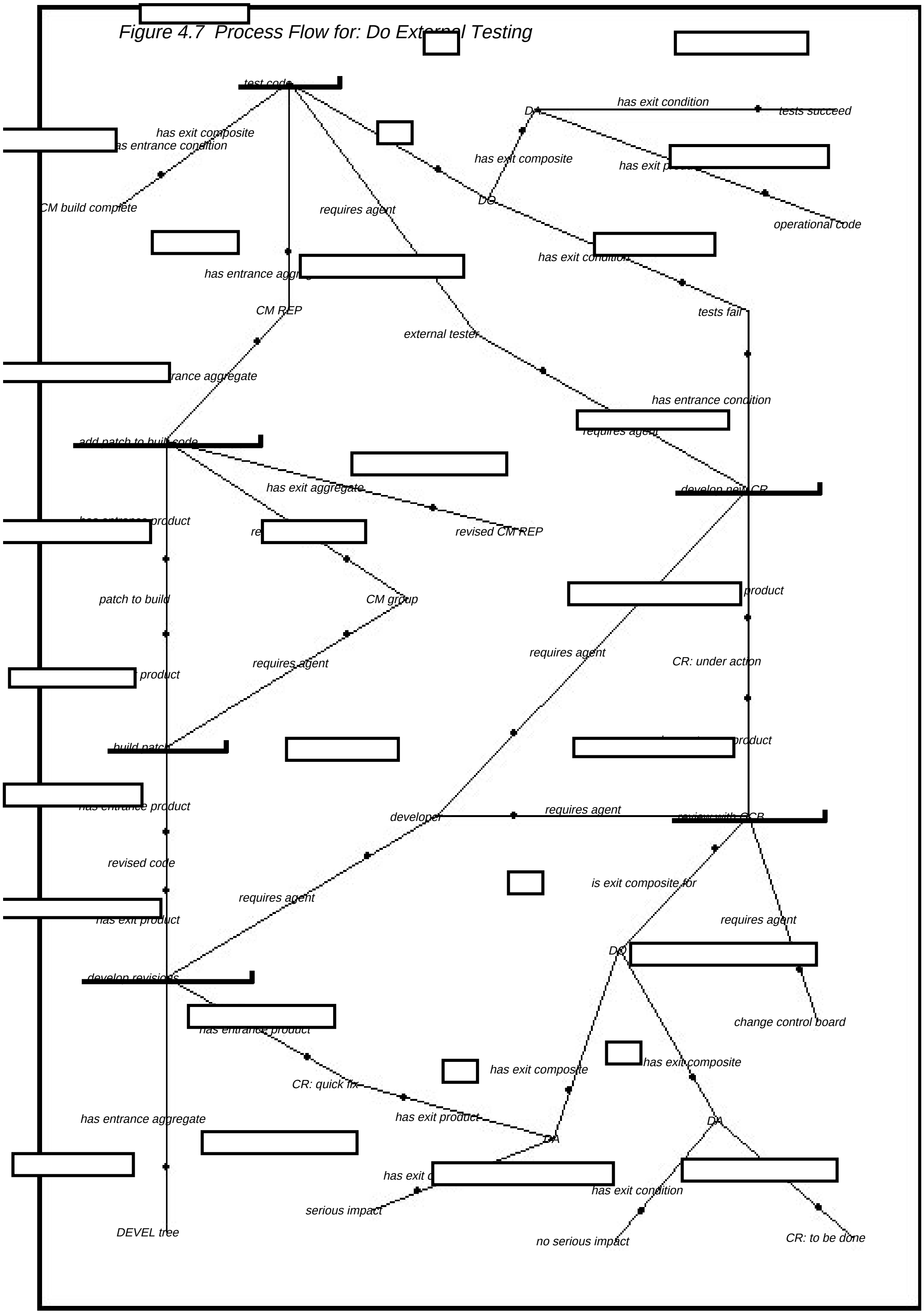


Figure 4.7 Process Flow for: Do External Testing

- Generate a *diff* file which contains only the updates to the CR.
- E-mail to staff members a file containing the modified CR plus a ‘diff’ list.
- Store the updated CR back in the s. history file.

4.9 Do Account Checks

After the development group has completed a set of modifications as defined in the CR, the appropriate files are transferred to the STAGE tree (see Section 4.5). At this point the CM group checks to make sure that the files are complete and correct. Figure 4.9 shows this process. Three common types of error may occur. The first type of error is simply that the files requested in the CR are missing *(**check s. files exist**)*. The second and third types of error are related. These have to do with:

- cross checking that the version numbers of the files to be modified (as specified in the CR) are consistent with the latest file version numbers in the *s.* history files *(***check version #’s***)*, and
- cross-checking that the MR number in the *s*. history file is consistent with the CR number of the change request under consideration *(**check MR & CR #’s match**)*. (Recall from Section 4.5 that the MR# was attached to the file after it was modified.)

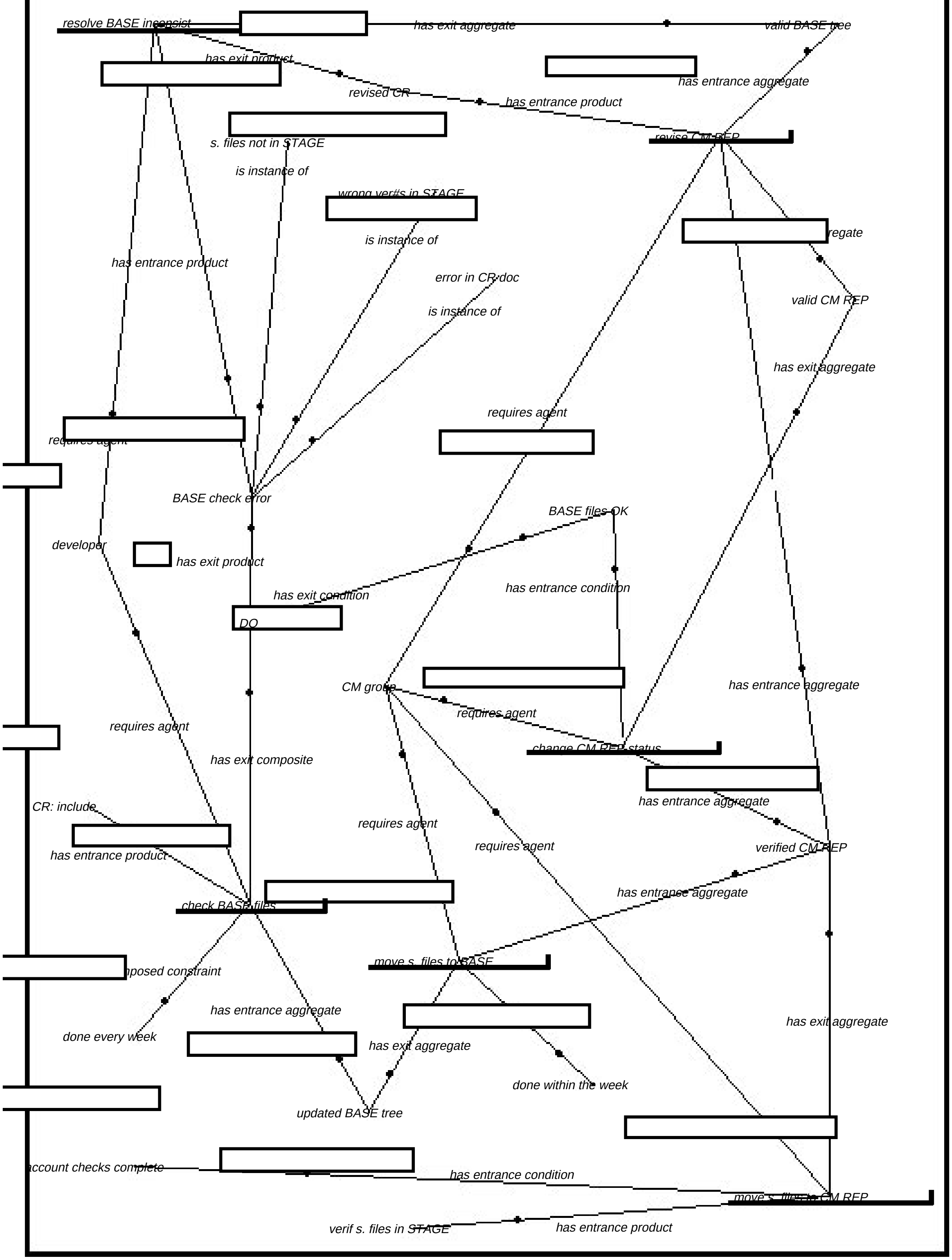
Inconsistencies arising from either of these two checks must then be resolved between the development and CM groups *(**resolve account errors**)*. Correct recording of this file/CR information is important not only for ongoing development, but also to allow future tracking of prior changes in case subsequent operational problems occur.

4.10 Move s. Files to CM Repository

Upon successful completion of the accounting checks, the files are moved from the STAGE tree to the CM repository *(**move s. files to CM REP**;* see Figure 4.10.). As soon as the files have been transferred, they are moved back to a tree called BASE. This tree has two functions. First, the BASE tree provides the developers with access to the current versions of the files under configuration management. Second, it allows developers to assure themselves that the files to be externally released are consistent with their intent (i.e., it is an additional safety check). In fact, the developers check these BASE tree files for correctness *(**check base files**)*. As with the accounting checks, the development and CM groups must resolve any inconsistencies found *(**resolve BASE inconsist)***. The CM group is then responsible for reflecting these changes in the CM repository *(**revise CM REP**)*.

In Figure 4.10, the files initially inserted into the CM repository are called *verified*, since they have gone through the accounting checks. After the BASE tree checks have been performed, and any errors corrected, the status of the files is then upgraded to *validated*. (This terminology is used in this report for clarity; these are not a generally used in the project.)

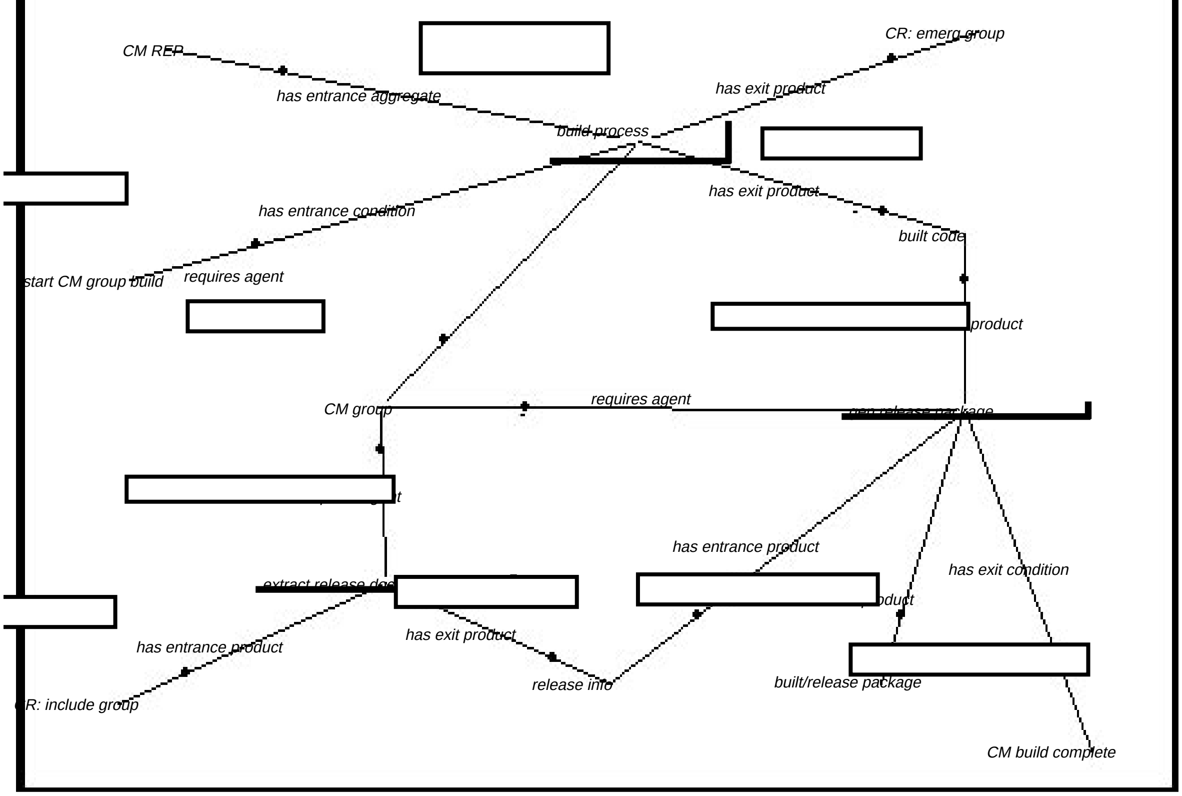
g 10 Process Flow for: Move s. Files to CM REP



4.11 Build by CM Group

The build performed by the CM group involves two activities (see Figure 4.11). First there is the build itself *(build process)*, and second there is the production of the accompanying release package (***gen release package***). During the build process, errors in compiling and so forth may arise, and these errors can result in additional “emergency” change requests being generated ***(CR: emerg group)***. The release package involves, in part, the extraction of information from the group of change requests which are associated with this build.

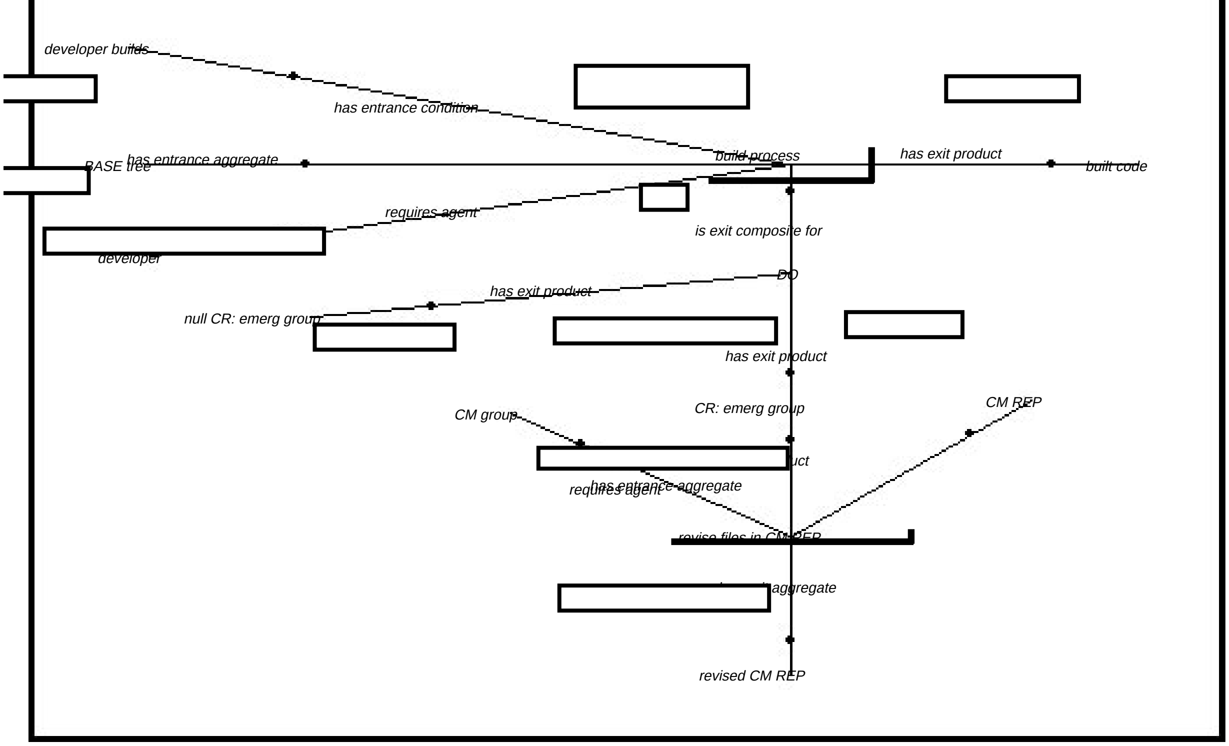
Figure 4.11 Process Flow for: Build by CM Group



4.12 Build by Developers

At any time during the accumulation of CRs and prior to a CM build, the developers may decide to perform a build (see Figure 4.12), using the files in the BASE tree ***(build process)***. A build may be performed for a number of reasons such as to check a particularly complex set of changes or simply because time permits. Errors may result in this build process, in which case a change request will be generated ***(CR: emerg group)***. This change request allows the CM group to revise the CM repository files as necessary ***(revise files in CM REP)***.

Figure 4.12 Process Fow for: Build by Developers



4.13 The Build Process

Surrounding the mechanics of the build are variety of activities related to change request support. This support is illustrated in Figure 4.13.

The build ***(perform build)*** can have two outcomes: a successful build ***(built code)***, in which case this sub-process is complete, or an unsuccessful build from which errors result. These errors must first be resolved ***(resolve build errors)*** and from this resolution, it is determined whether these errors must be processed by generating another change request ***(dev/revise CR: emerg fix)*** or whether the cause was minor (such as a missing file ***(file not found)***). In the latter case, the file is inserted ***(fix file error)***, and a new build is performed. If the error does require a new CR, then this CR is reviewed with the change control board. The board may accept or reject the proposed change, and if rejected, then the revise/review cycle is repeated. If accepted, then the CR is added to the group of changes which exist as part of resolving prior build errors ***(add to CR: emerg group).***

Once all errors are fixed (those requiring new CRs or otherwise) a successful build can take place. The existence of built code fires a pseudo-activity ***(complete CR cycle)*** allowing the complete group of CRs to become an exit product along with the code itself.

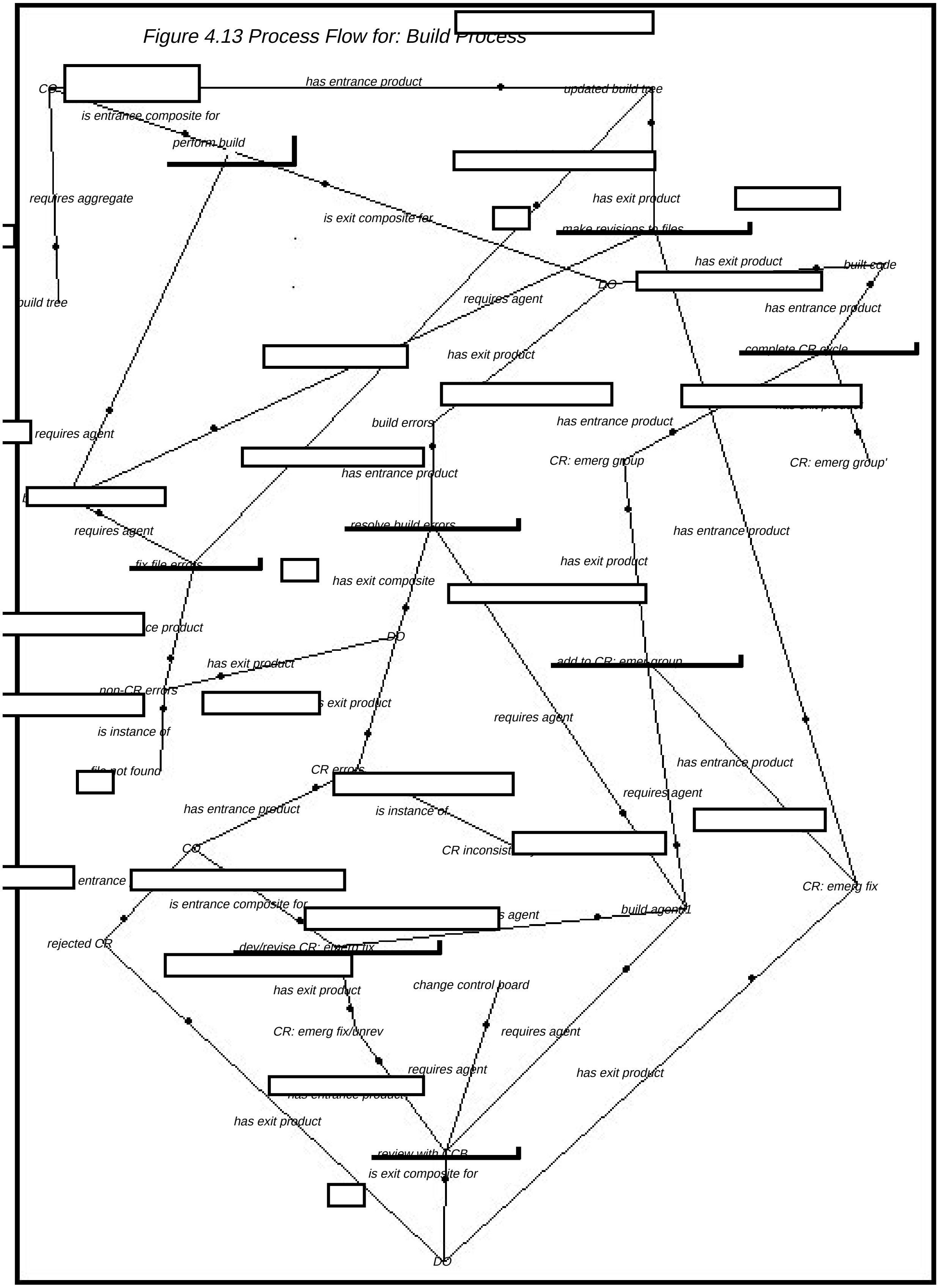


Figure 4.13 Process Flow for: Build Process

4.14 Perform Build

The actual build may be performed on one of many platforms (see figure 4.14). After selecting the target platform for the build ***(select target machine Y)***, either the CM repository tree or the BASE tree is copied over to that platform ***(copy tree X to machine Y)***. In order that the source files can be compiled on this platform, the compile scripts must be configured (through ifdef's) to the local platform characteristics. This is done through the use of a file called compile.env. The initiator of the build modifies this ASCII file, which contains a list of all platforms, by typing an arrow ("->") next to the name of the current platform ***(set pointer to machine Y)***. At this point, the build agent extracts the root make-file from the top node of the tree ***(get makeFile from root)***, and initiates the actual build ***(execute makeFile)***. Execution results in one of two possible products: built code or build errors. Once the build has been executed, and the executable code has been installed, the tree is removed from the platform ***(delete copy of tree X)***.

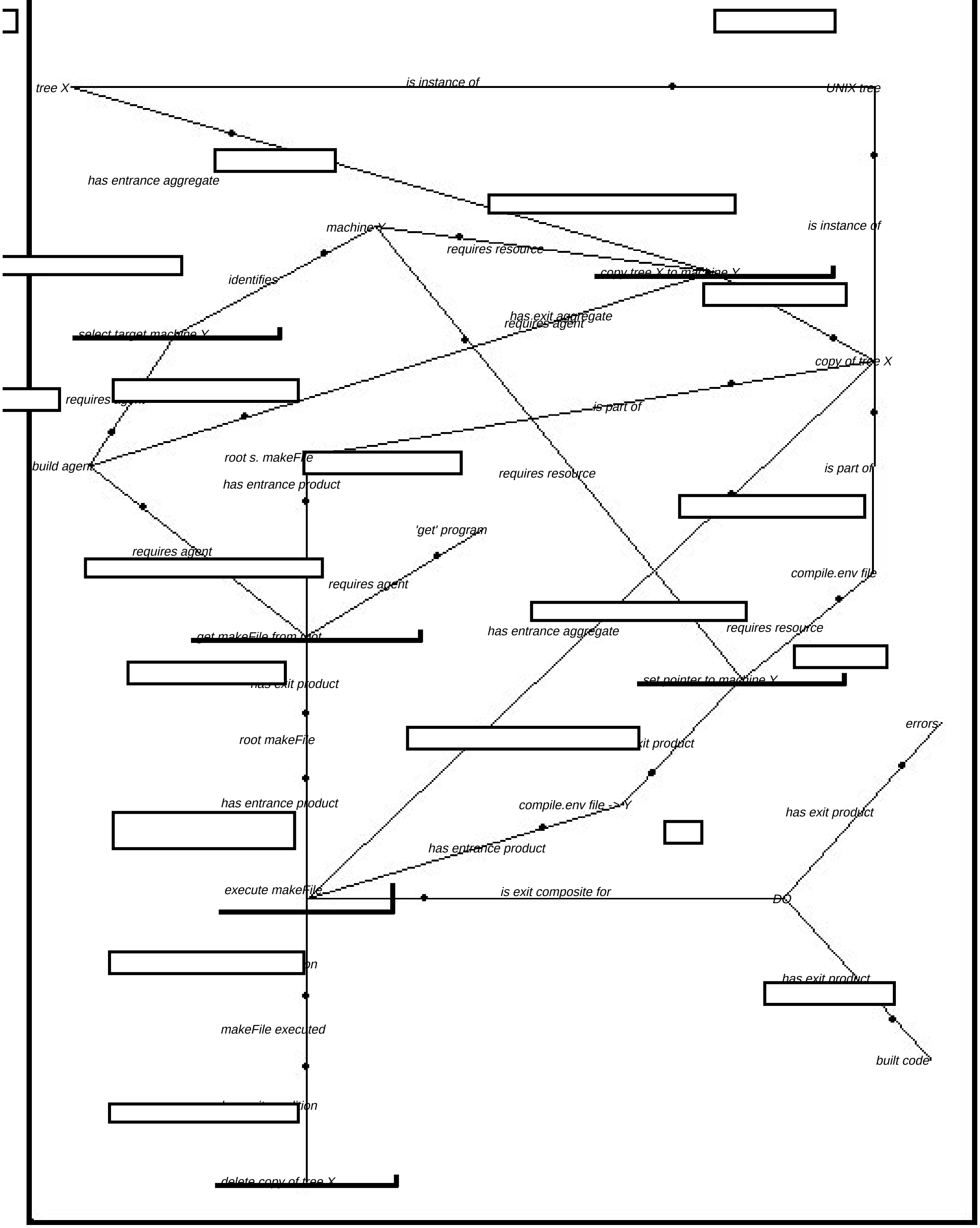
4.15 Execute Make File

The compilation and build process takes place automatically after the build agent has started up the make-file associated to the top node of the tree. The make-file serves two purposes: 1) to compile source code (through a call to a local compile-file) at this level in the tree and 2) to initiate make files in lower level subdirectories. The complete build process is recursive and quite complicated. Before describing the actual build process, an explanation of the tree structure is useful.

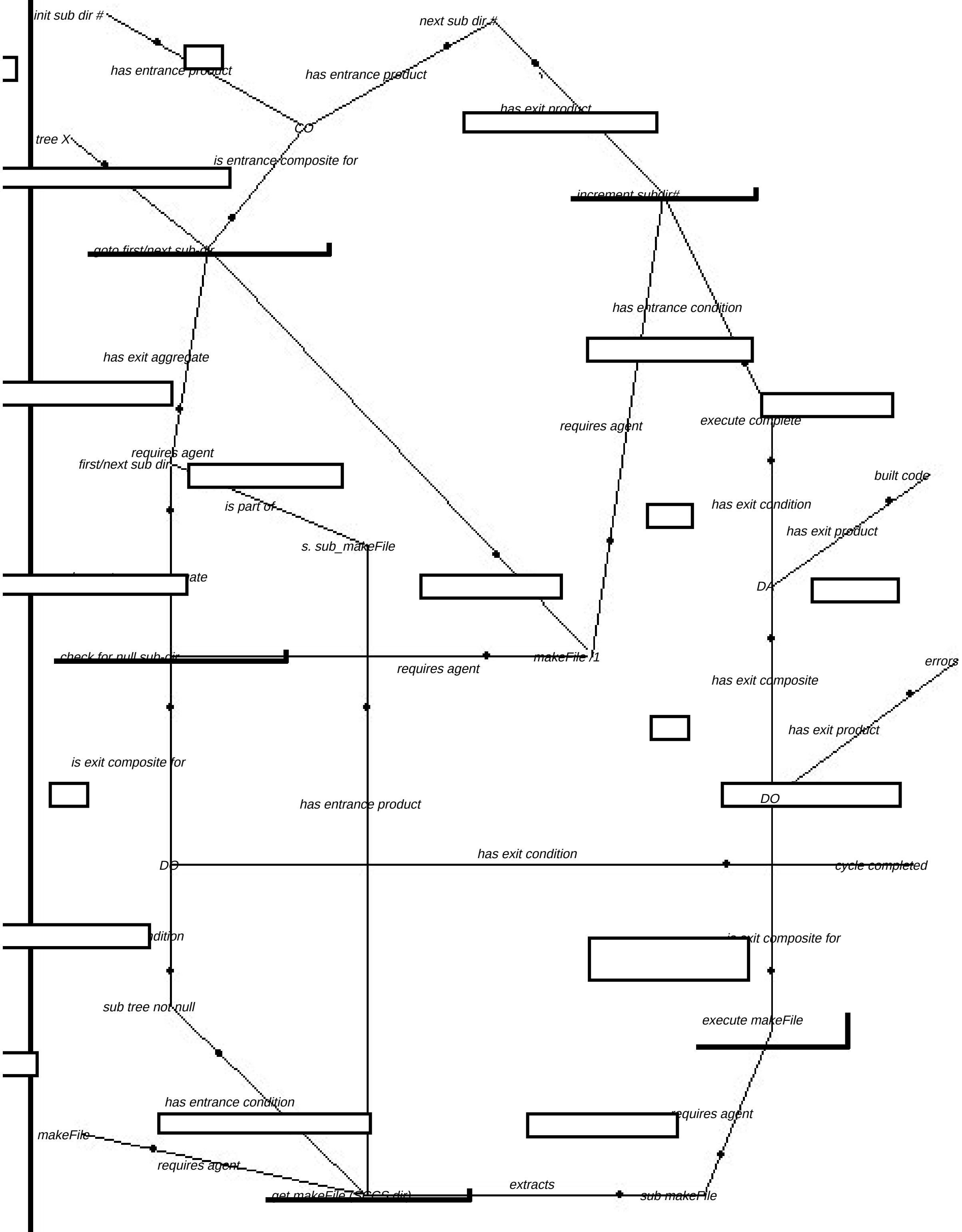
All the tree structures in the project are of identical type, which has been called "Unix tree" in this report (see Figure 4.17). Each node (or subdirectory) in the tree has an SCCS node which includes three file types: a file called *s.makefile* which contains the current makefile and its past history, an *s.compilefile* file which contains the current compile file and its past history, and source files and their past history (e.g., *s.f2writeblk.c*). In order to compile the source code across the tree, the makefile in the root SCCS directory is extracted from its *s.* file. Execution of this makefile in turn initiates makefiles in the SCCS subdirectories one level down from the current level. The process continues recursively down the tree and terminates on leaf nodes. If appropriate, the makefile also initiates execution of the local compile file in order to generate and install executable code from the source code for that node.

Figures 4.15 and 4.16 describe the build process in detail. The processes in these two figures are tightly coupled through mutually recursive calls between ***execute makeFile*** and ***cycle thru sub-dirs***. For instance, consider an arbitrary node in the tree. The makefile in the SCCS branch of this tree initiates the calls to the sub-directories directly below this directory. Cycling through the sub-directories requires a counter to be set up and initiated to the first sub-directory ***(init subdir# to 1)***. After cycling through the sub-directories ***(cycle thru sub-dirs)***, one of two outcomes is possible. Either all the code in the sub-tree has compiled, or an error has been generated in some part of the sub-tree. If an error has occurred then this error is simply

Figure 4.14 Process Flow for: Perform Build



igure 4.16 Process Flow f y -Directories



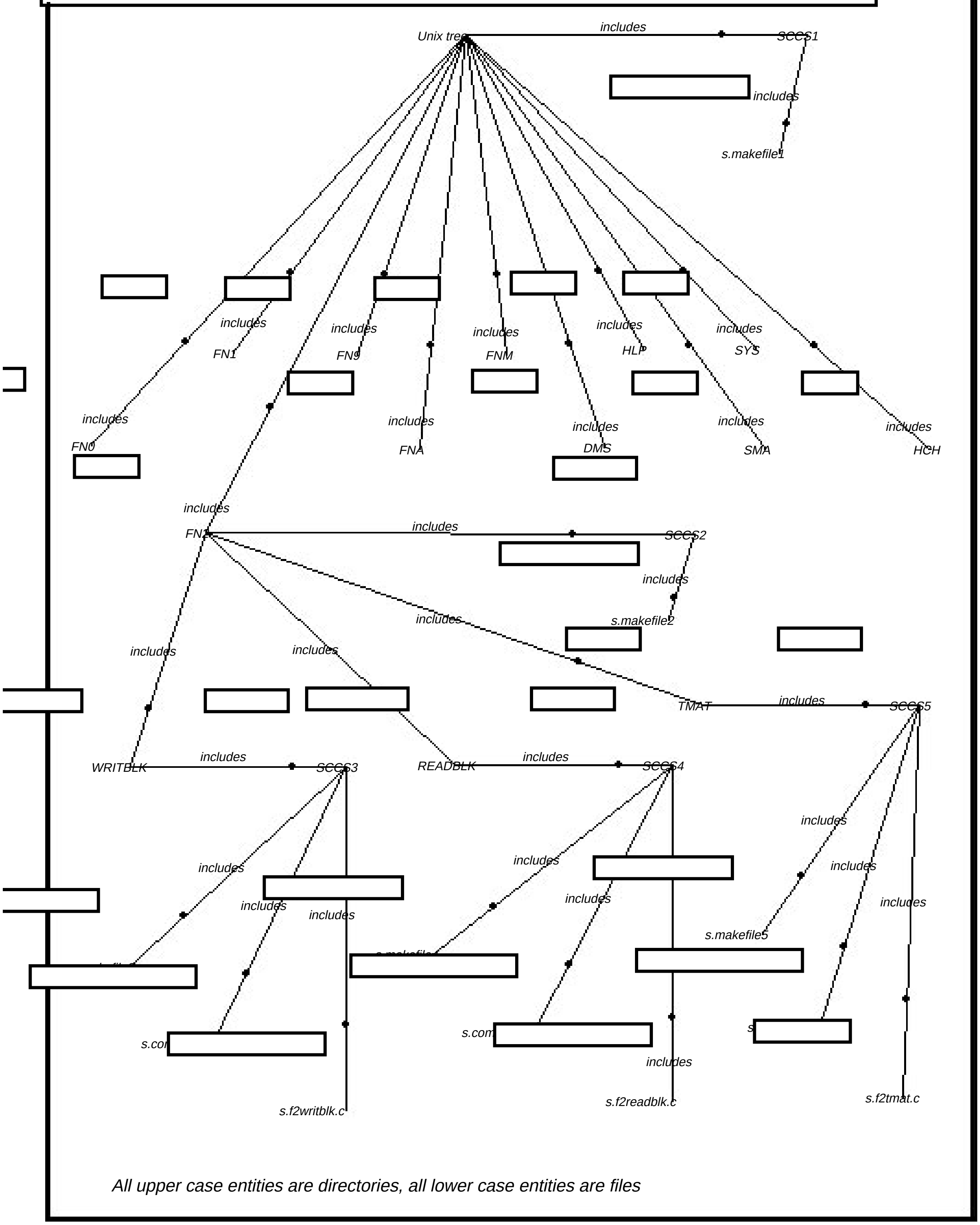
passed on to the next higher level in the tree ***(pass on sub-dir or local)*** bypassing the compilation which would otherwise have been performed at this node.

If no sub-tree errors have been detected, then the compile file is extracted ***(get compile script)***, and the compile sequence is initiated ***(run compile script)***. The compile sequence involves extracting the source code (***extract source***), compiling the source ***(compile source),*** linking compiled objects ***(link objects)***, and installing the resulting executables ***(install exe's).*** Any of these activities can result in errors, and if an error does occur, the sequence terminates with the error recorded ***(record error)***. This action initiates a backtracking up the tree, terminating the build process. If the build succeeds, the executable code becomes part of the accumulating code resulting from the succession of compiles at the nodes currently traversed ***(combine builds).***

4.16 Cycle Through Sub-Directories

Each non-leaf-node makefile initiates the makefiles in all the sub-directories immediately below the current directory. Figure 4.16 shows this cycle of incrementing through the lower level sub-directories (or nodes). Each subdirectory is chosen in turn ***(goto first/next subdir)***. The cycle ends when no more sub-directories exist at this level ***(check for null sub-dir)***. At each lower level node, the make-file (***sub makeFile***) is extracted from its associated *s.* file ***(get makeFile (SCCS dir))***, and this is then executed ***(execute makeFile)***. The output from this execution is either errors or built code. If an error is generated, then execution of subsequent make-files is bypassed, and the error is propagated backup the tree. If the build is successful, the variable *subdir#* is incremented ***(increment subdir#)***, and the next cycle initiated.

Figure 4-17: Unix Tree Structure



All upper case entities are directories, all lower case entities are files

5 Lessons Learned and Future Directions

Although much has been done to improve the project by applying configuration management (CM) principles, several key areas remain a problem. The large number of items[3] (2000), hardware platforms[4] (15) and versions of Unix operating system[5] (10) supported by the project continues to cause CM and testing problems. Also, several of the hardware platforms (Sun, 386 Clones) may have one of several different versions of the Unix operating system installed. The lack of automation of key CM functions continues to cause errors and delays during the system build and testing cycles. The CM system continues to evolve and solutions to these problems are being tested for future releases.

5.1 The Build Process

Building and testing new releases for multiple platforms is very time consuming. The entire tree structure must be installed, built, and tested on each platform. The only way to assure that all platforms are running identical versions is to make changes in a central repository and then move the entire tree structure to each platform. When problems occur in building or testing, the fixes must be installed on all platforms and each rebuilt and tested.

The solution to this problem is currently under test. A set of shell scripts has been developed which will create a new system on any target platform from sources located on one central system. The shell scripts create the tree structure of directories and the makefiles necessary to build and install the entire system on any of the target systems. The new makefiles obtain their input from the central system, and the output of the compile, build and install process are executed on the target system. The advantage of this technique is that changes can be made on a central system, and then all platforms can be built in parallel. Copies of the source tree do not have to be moved to each platform. The entire process is driven by four script files on the source machine (three common and one unique for each platform). Porting to new platforms has been simplified since the sources, the build and the install parameters are now centrally located on a single machine.

5.2 Testing

Once the new release is built, a major problem is the delay caused by testing. The organization chosen by management to test the project software was a Unix application software development office with its own development responsibilities. They are responsible for several prod-

[3] Source files, include files, makefiles and compile scripts (This number does not included customer developed code).

[4] DEC PDP-11/70; AT&T 3b5, 3b15, 3b20,3b2/600;SUN 3, 4; Data General Aviion; IBM AT, Intel 386/486 Clones; Apollo Workstation, DEC Workstation, AMDAHL, CRAY.

[5] Unix Version 6; SVR2; SVR3; SCO XENIX; SCO Unix 3.2;Aviion SVR5.4;Sun OS 3.5x, 4.0.3, 4.1;IBM XENIX.

ucts which duplicate project functionality and are, in a sense, in competition with the project. Within this organization, testing products developed in the project has low-priority. Testing is deferred until the release is ready for distribution and may represent many months of development and hundreds of individual changes.

Development continues during testing, and in some cases critical changes are ready to install before testing of the previous release has been completed. In some mission-critical cases, the software under development must be installed on the customer's system before the official release. This creates a situation where a customer can actually loose a fix or new capability by installing the latest release.

The testing delay increases the number of changes that must be tested for each formal release. A recent release included several hundred separate change requests. The solution to this problem is to divide the software into small separate products which can be tested and released individually. When significant changes have been made to one of the products, it will be tested and released. This process has started, but it is progressing slowly because of the tight coupling between some of the project programs.

5.3 Automation

Much of the CM auditing and baseline updating is done manually. New programs and procedures are being written to address this problem. Several fields in the CCB change request form have been changed (Appendix E) to support automatic scanning of the form. Software is being developed to scan the CR forms in order to automate the auditing and installation procedures performed by the CM team.

5.4 Buy versus Build

The CM system had to be as portable as the application programs that it supported. Since new features and bug fixes are applied to multiple hardware platforms, the CM system had to automatically support product build on all platforms. Since a commercial solution could not be found which met these basic requirements, the development team constructed their own CM system from the basic Unix tools available (**SCCS**, **sh**, **make**, **cc**, **ld**, **mail**, **grep** etc.).

The repository features of a system like Softool's CCC [Softool 91] could be used to store the software items, but little would be gained by doing so. The predefined CM process model imposed by a system like CCC Turnkey would not integrate well with the existing project development process. Adjustments to the CCC Turnkey model to match the current process would be expensive to implement, and it was not obvious what would be gained by doing so. (See also Section 7.2 for a discussion of this topic.)

Because of the tight coupling and dependencies between the various project programs, the project developers felt that easy access to all software was necessary and that sharing a common work space (DEVEL tree) was a benefit rather than a weakness. Commercial systems generally limit access to portions of the code and assume that each developer will work in a separate work space.

The decision to build rather than buy resulted in the following advantages to the project:

- The CM system is easily tailored to changes in the software development process, and development can continue its during any upgrades.
- The project owns the CM system and does not have to depend on the commercial market for support. Revisions and improvements to the development process are easily incorporated.
- The CM system is hardware-platform independent and can be easily ported to new platforms.
- Special training on the CM system is not required since the CM system uses standard Unix tools.
- Maintenance costs are lower. Expensive vendor upgrades are not required and the project does not need a CM “system administrator” to manage, maintain, and modify the system.
- Being vendor-independent, it is immune to companies whose future existence cannot be guaranteed.

6 The Project Software Process and the Capability Maturity Model

This section looks at the project software process from the point of view of the configuration management practices described in the Capability Maturity Model (CMM). Other key practice areas, such as Software Quality Assurance, are not evaluated. The sub-sections below list the CMM practices as provided in [Paulk 91], and then give comments (each comments being in italics and being preceded by a bullet). These comments discuss the project maintenance process in relation to the practices. Some of the practices in the CMM are expanded to provide greater detail; some of this detail has been omitted below if it did not contribute useful insights. For a detailed discussion of the CMM and its underlying rationale, [Paulk 91] and [Weber 91] should be consulted.

There are two motivations behind this comparison. First, the exercise of comparing a process model with the practices of the CMM is useful in answering the question: Do such process models help in identifying process weakness and areas for process improvement? Second, this report defines the project maintenance process in sufficient detail that it could be adopted by others. For an organization wishing to build a maintenance process similar to the project's, it is useful to identify where enhancements might make the process more effective.

6.1 Goals

Goal 1: Controlled and stable baselines are established for planning, managing, and building the system.

- *Yes. Baselining of software components is clearly the objective of the project's process through use of change requests, a change control board and the software support environment.*

Goal 2: The integrity of the system's configuration is controlled over time.

- *Yes. The system (i.e., products) is controlled through the use of the change requests and the close coupling of these CRs to the files in the CM repository.*

Goal 3: The status and content of the software baselines are known.

- *Yes. The baseline content is well controlled through effective use of the Unix file system and the content and history of changes are known and can be tracked through the change requests.*

6.2 Commitment to Perform

Commitment 1: The organization follows a written policy for implementing software configuration management (SCM).

- *No. While a written policy for defining configuration management does not exist for the project, practices 1 through 5 below are well understood, strictly observed, and to a great extent controlled through automated procedures.*

This policy requires that:

1. Responsibility for SCM for each project is explicitly assigned.

- *Yes. This responsibility is shared between the developers, change control board and the CM group.*

2. SCM is implemented on products throughout the project's life cycle.

- *Yes. All software products being developed or maintained are controlled throughout their life cycle.*

3. SCM is implemented for externally-deliverable products and for appropriate products used inside the organization.

- *Yes. All deliverable products and associated internal products (i.e., the change requests) are maintained under CM control.*

4. All projects have a repository for storing the key software engineering elements (i.e., configuration items) and the associated SCM records.

- *Yes. A repository for all deliverables is maintained under the control of the CM group.*

5. The software baselines and SCM activities are audited on a regular basis.

- *No. Formal auditing is not conducted. Informal checks of new baseline items may be made the developer group.*

6.3 Ability to perform

Ability 1: A board having the authority for managing the software baselines (i.e., a software configuration control board - SCCB) is established.

- *Yes. The project change control board is central to authorizing and controlling all modifications to software baselines*

The SCCB:

1. Authorizes the establishment of software baselines and their configuration items.

- *Yes.*

2. Represents the interests of the project manager and all groups who may be affected by changes to the software baselines.

- *Yes. The project manager is on the SCCB. All interested parties are invited to attend the SCCB review meetings in order that their opinions are accounted for.*

3. Reviews and authorizes changes to the software baselines.

- *Yes.*

4. Authorizes the changes to software baselines.

- *Yes. Authorization results from the SCCB review meetings and as reflected in the change requests.*

Ability 2: A group that is responsible for coordinating and implementing SCM for the project (i.e., the SCM group) exists or is established.

- *Yes. The project has its own CM group for controlling software baselines.*

The SCM group:

1. Creates the project's software baseline library.

- *Yes.*

2. Develops, documents, and distributes the SCM plans, standards, and procedures.

- *No. Formally documented SCM plans are not developed.*

3. Manages access to the software baseline library.

- *Yes. Only the CM group have access to the repository.*

4. Updates the software baselines.

- *Yes. And it verifies their completeness and correctness.*

5. Creates software baseline products.

- *Yes. It performs the builds which result in the baseline products.*

6. Records SCM actions.

- *Yes. It records actions through the change requests.*

7. Produces and distributes SCM reports.

- *Yes. It does so for release notes and summaries.*

Ability 3: Adequate resources and budget for performing the SCM activities are provided.

- *Yes. Adequate resources and budget are provided.*

1. A manager is assigned specific responsibilities for SCM.

- *Yes.*

2. Appropriate tools to support the SCM activities are made available to the SCM staff and to the software engineering staff.

- *Yes. These include automated support for change control evolution and support for software build, multiple databases for software development and baselining, along with appropriate tools for manipulation of software artifacts. Appropriate hardware to support the software environment is also provided.*

Ability 4: Members of the SCM group are trained in the objectives, procedures, and methods for performing their SCM activities.

- *No. Formal training is not given. Training is on the job.*

Ability 5: Members of the software engineering staff are trained to perform their SCM activities.

- *No. Formal training is not given. Training is on the job.*

6.4 Activities performed

Activity 1: Different levels of SCM are implemented, as appropriate, during the project's life cycle.

- *Yes. Different levels of SCM are implemented (e.g., between the developer and SCM groups). However, the project's life cycle is primarily in the maintenance phase, and the scope of SCM needs is more limited than in other projects.*

1. The level of SCM is different for different products.

- *Not applicable. The product type is quite uniform.*

2. The level of SCM changes during the project's life cycle to provide a balance of control and flexibility that is most beneficial to the project.

- *Yes. Different levels of SCM are implemented (e.g. between the developer and SCM groups). However, the project's life cycle is primarily in the maintenance phase, and the scope of SCM needs are more limited that in other projects.*

3. The software engineering group informally manages its individual products during development.

- *Yes. The development group has control over its products prior to their being baselined by the SCM group.*

4. The software engineering group uses informal SCM for products not under formal SCM.

- *Yes. The development team use informal version and configuration control to manage products under modification.*

5. The project uses formal SCM for control and stability of baselined items when they are needed to coordinate and interact between project groups and with the customer.

- *Yes. Baseline control is strictly adhered to when product is released to external customers. Interaction between project groups is not applicable.*

6. The project's software baseline library and its contents are controlled and available after the project ends.

- *Yes.*

Activity 2: A documented SCM plan exists.

- *No. A documented SCM plan does not exist.*

Activity 3: A documented and approved SCM plan is used as the basis for performing the SCM activities.

- *No. A documented SCM plan does not exist.*

Activity 4: A configuration management library system is established as a repository for the software baselines.

- *Yes. It is.*

This library system:

1. Supports multiple control levels of SCM.

- *Yes. It does in the sense that, while the SCM group have tight control over the repository, access to baseline information can be retrieved by the development group through accessing the BASE tree (which reflects the contents of the CM repository).*

2. Provides for the storage and retrieval of configuration items and their configuration components.

- *Yes.*

3. Provides for the sharing (i.e., read-only) and transfer of configuration items and their configuration components between the software engineering groups' control and the SCM group's control and between control levels within the library.

- *Yes. Sharing of baseline information results from the CM group's maintaining of the BASE tree in the same configuration as the CM repository. The BASE tree is accessible to the developers.*

4. Helps to enforce product standards (e.g., naming and format) of configuration items and their configuration components.

- *Yes. Several databases are used in the project environment. These all have a standard structure. In addition the files within these databases have names which conform to a defined naming convention.*

5. Provides for the storage and recovery of archival versions of configuration items and their configuration components.

- *Yes.*

6. Helps to ensure correct creation of software baseline products.

- *Yes. There are multiple cross-checks within the project 's process which ensure the correctness of baseline products, for example, c ross-checking of change control numbers, cross-checking of file version numbers, and verification by the development group that the baseline product is consistent with their expectations.*

7. Provides for the storage, update, and retrieval of SCM records.

   - *Yes. Control and storage of the change requests allows tracking of any previous changes that were made.*

8. Produces SCM reports.

   - *Yes. These include release notes, installation notes, summaries.*

9. Provides for the maintenance of the library structure and contents

   - *Yes. The system performs regular backups and provides for restoration and recovery from errors and crashes.*

Activity 5: The software engineering products and process specifications (i.e., configuration items) to be placed under configuration management are identified.

- *Partial yes - the answers to items 1 through 7 below provide explanation.*

1. The configuration items are selected based on documented criteria.

   - *No. There are no documented criteria for selection configuration items. Items under configuration control include the software items under modification and change requests.*

2. The characteristics of each configuration item are specified.

   - *No.*

3. The software baselines to which each configuration item belongs are specified.

   - *Yes. Indirectly through the prior change requests, and CR status summaries.*

4. The point in the software life cycle that each configuration item is placed under configuration management is defined.

   - *Yes. This happens after the CCB authorization takes place.*

5. The person responsible for each configuration item (i.e., the owner, from a configuration management point of view) is defined.

   - *Yes. The CM group are responsible for all configuration items.*

Activity 6: A documented procedure is followed for initiating, recording, reviewing, approving, and tracking change requests and trouble reports for all configuration items.

- *No. However, well understood procedures are followed, and much of this process is enforced through automation of, for example, the change request procedure.*

Activity 7: A documented procedure is followed to control changes to configuration items.

- *No. A documented procedure does not exist. However a well-understood procedure exists.*

This procedure requires that:

1. Established controls are followed to ensure that configuration items are checked out and checked in for change in a manner that maintains the correctness and integrity of the software baseline library.

- *No. No documented procedure exists, but strict check-out and check-in procedures are followed.*

2. Reviews and/or regression tests are performed to ensure that changes have not caused unintended effects on the product.

- *No. No documented procedure exists, but testing by an independent group is performed to assure adequacy of the software. (The independent testing procedure could be stronger in that the tests performed originate with the development group.)*

3. Revised configuration items are audited to ensure that they are prepared according to the SCM standards and procedures.

- *No. No written SCM standards or procedures exist, and therefore no audits are performed as prescribed in such documents. However, checking is conducted in several other ways including: reviews of the change requests by the change control board, and reviews of baseline products by the development group (using the BASE tree).*

4. Only configuration items that are accepted by the SCCB are entered into the software baseline library.

- *Yes.*

Check-in procedures include steps to verify that the revisions are authorized, to create a change log, to maintain a copy of the changes, to update the software baseline library, and to archive the replaced software baseline.

- *Yes. All these procedures are followed.*

Activity 8: A documented procedure is followed to create and control the release of software baseline products.

- *No. No documented procedure is followed. However, a well-understood procedure is followed.*

This procedure requires that:

1. The SCCB authorizes the creation of software baseline products.

- *No. No documented procedure exists, but the change control board does authorize the creation of software baseline products.*

2. Software baseline products, for both internal and external use, are built only from configuration items in the software baseline library.

- *No. No documented procedure exists, but software baseline products, for both internal and external use, are built only from configuration items in the CM repository.*

Activity 9: A documented procedure is followed to record the status of configuration items and change requests.

- *No. No documented procedure exists, but the status of configuration items and change requests is strictly controlled through, for example, the change control board.*

This procedure requires that:

1. The configuration management actions are recorded in sufficient detail so that the software baselines' content and status are known and previous versions can be recovered.

- *Yes. This information is recorded in the change request.*

2. The current status and history (i.e., changes and other actions) of the software baselines are maintained.

- *Yes -* s. *files containing the current versions and previous histories of files are maintained and any file version can be accessed through the SCCS configuration management system.*

Activity 10: Standard reports documenting the SCM activities and the contents of the software baseline are created and distributed to affected groups and individuals.

- *Yes.*

These reports include:

1.SCCB meeting minutes

- *Yes. CCB meeting minutes are generated and distributed.*

2. Change request summary and status.

- *Yes. They are included in the change request.*

3. Trouble report summary and status (including fixes).

- *Yes. They are included in the change request.*

4. Summary of changes made to the software baselines.

- *Yes. They are included in the software release notes.*

5. Revision history of configuration items.

- *Yes. They appear in the SCC* s.*files.*

6. Software baseline status.

- *Yes. they are included in the CRs pending revision*.

7. Findings of software baseline audits.

- *No. No formal baseline audits take place. In some cases developers will build perform software builds on code derived from baseline files. This activity provides a cross check on the baseline products.*

Activity 11: A documented procedure is followed to prepare for, conduct, report results from, and track action from software baseline audits.

- *No. No documented procedure is followed to prepare for, conduct, report results from, and track action from software baseline audits. However, as noted under item 6 of activity 4 above, extensive checks are made to assure accuracy of all baseline products.*

6.5 Monitoring implementation

Monitor 1: Measurements are made and used to determine the cost and schedule status of the SCM activities.

- *No. No measurements are made or used to determine the cost and schedule status of the SCM activities.*

6.6 Verifying implementation

Verification 1: The SCM activities are reviewed with senior management on a regular basis.

- *No. SCM activities are not regularly reviewed with senior management (i.e., management above the operation level of the project).*

Verification 2:The SCM activities are reviewed with the project manager on a regular basis.

- *Yes. The project manager is closely involved with oversight of the project activities.*

Verification 3: Periodic audits are performed to assess how well the SCM standards and procedures are being followed and how effective they are in managing the software baselines.

- *No. This is because there are no formal standards or procedures.*

Verification 4: The software quality assurance group reviews and audits the activities and products for SCM, and reports results as appropriate.

- *No. This is because there is no software quality assurance group.*

6.7 Conclusions

The exercise of mapping the characteristics of the project environment to the Capability Maturity Model has provided several insights. First, as might be expected, one does not need a process model in order to compare a project's practices to those to the CMM. The act of building the process model is, however, very helpful in performing this comparison. In addition, it was found, again not unexpectedly, that the CMM refers to some issues which may not be addressed in a process model. For example, a process model is unlikely to incorporate training activities or may not explicitly specify certain documentation (e.g., SCM standards) which are needed for compliance with the CMM.

In summary, the following were found to be the strengths and weaknesses of the project maintenance process, as compared to the practices defined in the Capability Maturity Model:

Strengths:

- The process of assuring correctness and completeness of baselines is effective.
- Change control is well defined and managed through an effective CR process.
- The SCM function is separate and independent of the development function.

Weaknesses:

- Lack of documentation (procedures, plans, etc.).
- Lack of formal training.
- Lack of tracking (e.g., on cost and schedules).
- Lack of independent audits.

Most of the above weaknesses do not deal with the day-to-day process issues as modeled in Section 4, but rather with issues which affect the longer term operation. For example, if there were a major movement of personnel out of the project, the lack of training and documented procedures would become a more vital issue.

7 Reusing Elements of the Project Software Process

The process model defined in Section 4 provides a good testbed to explore some of the issues associated with process reuse, a topic of some relevance to this report. In designing a maintenance (or other) system to meet a specific set of requirements, it is unlikely that a design such as described in this report would be appropriate in its entirety. This suggests the possibility of using elements of the existing process as building blocks of a modified process. After discussing reuse in general, several concrete examples are described, providing some context. Note that the discussion deals with reuse of elements of an existing process which was not built with reuse in mind; it does not deal with the design of processes for reuse.

7.1 Reuse Building Blocks

The project software process model has significant hierarchical structure (see Figure 4.1). At the start of the modeling effort, this structure was not clear. As a particular process diagram grew more complex, it became apparent that some of the modeling detail was better captured at a level below the current one. This situation occurred several times and resulted in a final model having six levels as can be seen in Figure 4.1. One challenge of this decomposing process was to identify the “best” architecture, one which minimized information flow between process diagrams, while it maximized information flow within each diagram.This not only helped with clarity, but also resulted in well defined functional units which mapped directly to the actual process. In summary, the major building blocks are:

- Level 1: define maintenance life cycle.
- Level 2: identify problem, review & approve CR, install modifications, perform CM activities, do external testing.
- Level 3: revise CR, do accounting checks, move files to CM repository, build by developers, build by CM group.
- Level 4: build process.
- Level 5: perform build.
- Level 6: cycle through sub-directories, execute make file.

Communication of information between the process diagrams occurs in two ways: through explicit links between diagrams, and through access of information from a “globally” define resource. In the former case, change request information and status (or condition) changes are the main examples, while in the latter case, information in the databases (data stores) is the most obvious example.

With respect to the levels, it is interesting to examine some qualitative differences. The top level (Level 1) represents a process diagram of the project’s life cycle. For the intermediate levels, humans are the main active agents, although machine agents also operate. At the lowest level (Level 6), only machine agents operate, indicating that the associated sub-processes are entirely automated.

The change request is seen to be an important artifact at the top level, being the “glue” which holds to sub-processes together. It therefore seems that significant changes to the characteristics of the change request form could not be made without significant impact on the global process. Similarly, the architecture of the repository and the manner in which the data is stored using SCCS constraints the higher level CM process. However, so long as the new process maintains the conventions: 1) the inputs to and output from a reused building block are consistent with the project’s conventions, and 2) the changes to the underlying repository are consistent with the project database conventions, then the new process can use the candidate building block (see Figure 7.1).

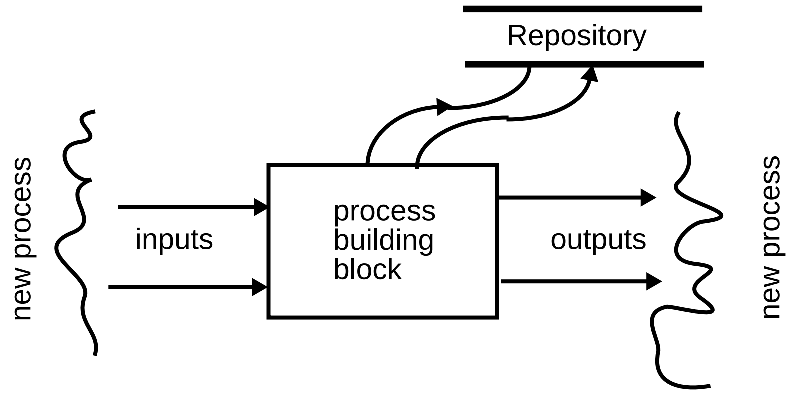


Figure 7.1 Maintaining Data Consistency Around a Building Block

7.2 Some Simple Reuse Examples

To help illustrate issues on reuse, this subsection describes some simple reuse cases involving the project’s sub-processes. The first example looks at modifying the project’s software process in order to account for software development rather than software maintenance. This modification primarily involves modifying the “front end” of the project’s process. To make this change, the three high-level activities *identify problem* and *review/approve CR* and *install modifications* (see Figure 4.2) must be removed and replaced with the activities shown in Figure 7.2. However, this is not a complete replacement, since the old *review/approve CR* process can be reused in the requirements and design areas. In addition, the *develop S/W modules* activity is virtually identical to the old *install modifications* activity, and hence can be reused. Only the *develop requirements* and *develop design* processes needs to be constructed. In this example, the chances of adapting the project process elements for reuse look good.

A second example of process reuse involves: building a maintenance system in which the basic project process is used, but which is supported by a different repository design such as CCC [Softool 91]. The turnkey version of CCC has a life cycle process model embedded in it,

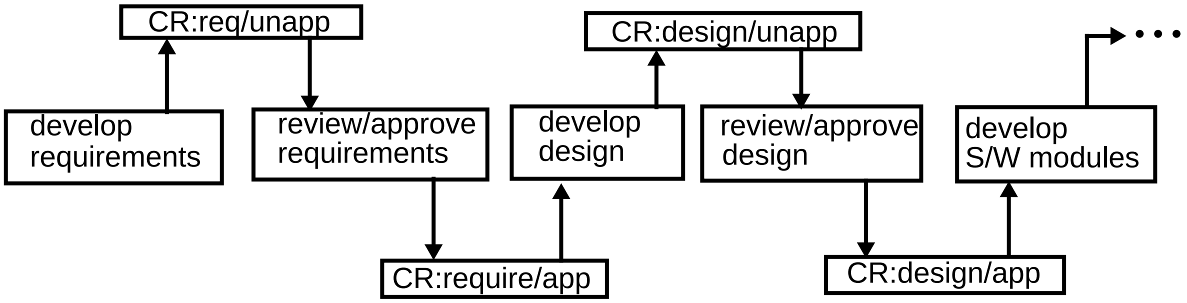


Figure 7.2 Modifying the Front-End of the Project Process

and, as with the project, its change request form is central to its operation. However, CCC has its own notion of project states, some of which are distinctly different from the project process states. At the top level, this difference may be a significant impediment in the use of CCC. In addition, the turnkey CCC may not be able to support CM over multiple platforms in the way the project was designed to do. Finally, in CCC, the test group is responsible for final approval of the software, not the CM group. On the positive side, with appropriate changes in CR status naming conventions, the sub-processes *identify problem* and *review/approve cycle* may be reusable. However, in most of the other cases, the sub-processes are tied too intimately to the structure and the distributed nature of the databases so that their re-use is questionable. Hence the chances of this adaptation being successful are low.

The final example of reuse involves stripping the repository out of the project environment and using it to support an entirely new process. The discussion here will not define the new process, but instead highlight some implications which result from the design of the project's repository. The project repository consists of multiple databases of identical structure, that is respectively, DEVEL, STAGE, BASE, and the CM repository. Each of these is designed to support a specific part of the project's life cycle. If this "life cycle" environment were to support another software process, much of the process "baggage" would have to be accepted also. However, the process constraints are imposed through the use of the multiple databases, not through the design of the database itself. (Each data base is tied to a different state in the project's process.) Thus, if only the basic design of the database were adopted, significant flexibility could be accommodated in the use of both the database structure and the accompanying build and compile script files in the new software process.

8 Summary and Conclusions

The maintenance project described in this document started out with a modest goa l: to support a Unix-based software environment on a small number of platforms. Over time, the demand for the project's products grew as more customers wished to usethem. This resulted in the need to support an increasing number of platforms, operating system variants, and files. In addition, new network technology was introduced and the distinction between terminals and host computers was blurred with the introduction of high performance workstations. Thus the project support group was forced to develop and implement an effective configuration control system for this increasingly complex operation and to work within a well-defined and understood process.

The CM system developed includes a number of significant elements: the change request form, the change control board, the independent CM and test groups, and the functionality provided by the automated build, compile, and install facilities. The change request form is the central document which ties the maintenance process together. All information about the status of modifications is incorporated into the CR; hence, this document provides a “paper trail” for tracking and auditing the progress of changes. The change control board reviews all proposed changes (using the CR), whether those changes are related to new requests, solution approaches, or implementations. The CCB is also responsible for final release of new revisions. The CM and external test functions have been separated from the development activity. This allows the developers to focus on development, while a dedicated CM group provides an objective basis for product quality. As a backup guarantee on product quality, the development group is also able to cross-check the files actually sent to the CM repository. Finally, much work was invested in building the scripts and compile files. These scripts and files allowed for a significant degree of automation of the compile, build, and install activities.

These changes to the process have resulted in many benefits. First, program managers can track the status of all pending changes, and also interact with and influence the decisions made by the developers. Second, the developers can trace individual items which make up a program. Third, customers have benefitted because much of the uncertainty about both versions of the program they are running, and the status of requested changes has been removed. Finally, the quality of the product has improved since the built-in checks minimize the likelihood of wrong file versions being accidentally incorporated into released software.

This report has defined the process in considerable detail, both through textual description and through a graphical process model. The graphical model turned out to be many layers deep, the top layer representing the overall “life cycle” of the maintenance process, and is dominated by the evolution of the change request. The focus of the lowest layer represented the completely automated build process where build scripts were the active agents. Between these extremes, both human and machine agents drive the process. This model added a degree of precision to the definition of the process which is difficult to capture in the textual description.

Through the experience gained in operating this maintenance project, several improvements have been identified and are currently being incorporated. These include increasing the degree of automation of the build process, reducing the delays in the external test phase, and automating the CM audit. With the current system, the build for each platform must be performed individually, while with the anticipated system, builds will be performed automatically and simultaneously on all platforms. The current external test procedure involves a group who have other responsibilities besides testing project software. Consequently slow turn around occurs. A plan is being implemented to use other human resources to perform this test function. Finally, to improve the manual CM audit function, software is being developed to extract information from the CR form and through this, to automate the audit and installation procedures.

A comparison of the project's process against the CM practices of the Capability Maturity Model [Weber 91] was made. This comparison resulted in the identification of strengths and weaknesses in the existing process. These strengths and weaknesses are listed below.

Strengths:

- The process of assuring correctness and completeness of baselines is effective.
- Change control is well defined and managed through an effective CR process.
- The CM function is separate and independent of the development function

Weaknesses: :

- Lack of documentation (procedures, plans etc.).
- Lack of formal training.
- Lack of tracking (e.g. on cost and schedules).
- Lack of independent audits.

This profile of strengths and weaknesses indicates that the required operational aspects of the project's software process are very adequately addressed, while some issues related to, for example, higher management oversight, could be strengthened.

The final issue addressed was that of reuse. The model defined in Section 4 provides a breakdown of the project's process into relatively self-contained sub-processes which communicate with each other. three hypothetical examples were looked at in order to see if these sub-processes could be modified or rearranged to support other processes. These examples were: 1) modifying the process to account for front-end requirements and design phases, 2) replacing the software support environment with a commercial system such as CCC, and 3) porting the project's directory structure together with the build scripts and compile files to another software development process.The conclusions reached were that 1) and 3) might be achievable while 2) is unlikely to be successful.

Appendix A How to Read the Process Diagrams

To understand the process diagrams in Section 4, this appendix provides a brief guide to the formalism. The appendix does not describe how to use the program, ProNet [Christie 92], which generated the diagrams.

The diagrams are based on a modified Entity-Relation model, in which the entities and relationships have defined types. Central to the approach is the entity type *activity*. Activities form "anchor points" to which other entities are attached. The other entity types are:

- **Products**. These can either be required to support an activity or be produced by an activity. Products must be the result of some activity being modeled; otherwise, it must be categorized as resource (see below).
- **Conditions**. These can either be required to initiate an activity or result from an activity.
- **Agents**. These are active entities (either human or otherwise) needed to support an activity. As defined here, agents are roles rather than role instances. (i.e., it is sufficient to specify *project chief* without specifying the project chief's name.)
- **Resources**. These are passive entities (produced externally to the system being modeled) required to support activities and they maybe transformed into products by the activity (e.g., a blank form).
- **Constraints**. These are policy restrictions imposed on the performance of an activity (e.g. a quality assurance constraint).
- **Junctions**. These are Boolean combinations of conditions, products, agents etc.(i.e. the existence or non-existence of a product, etc. is equivalent to a condition).
- **Aggregates**. These are combinations of entities differing in type. For example, a *UNIX tree* is an aggregate which is composed of files (type: *product*) and a directory structure (type: *resource*).

There is no finer-grained definition to the "agent" type. Agents can be humans or machines and the distinction between agent roles and physical agents is not made. However, given the nature of the model this lack of distinction does not lead to any obvious ambiguities.

Most relationships link the entities to the activities. Hence we have such relationships as:

{product} *is entrance product for* {activity}

{product} *is exit product from* {activity}

{agent} *is required agent for* {activity}

In general, products, conditions, resources, junctions and aggregates form inputs to and outputs from activities, while agents and constraints are required by activities.These relationships are all written in italics, and the information they contain allows the reader to identify the type of entity linked to the activity. The activities can be identified since they always have a shadowed box around the entity name. In addition, a small black dot is placed next to the entity at the end of the relationship. For example in the relationship “ABC *is entrance product for* XYZ”, the dot would appear in the graphical relationship close to the box surrounding the activity XYZ. The above information is illustrated in Figure A.1

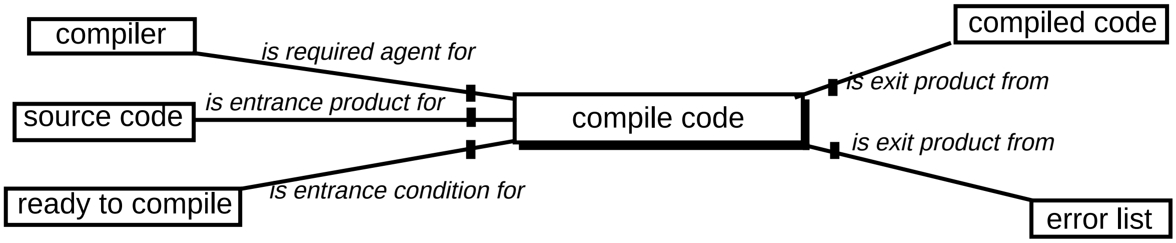


Figure A.1 A Simple Process Diagram

There is a second class of relationship having to do with generalization (inheritance) and aggregation relations. The generalization relations are *is generalization of* and *is instance of*, while the aggregation relations are *includes* and *is part of*. These relationships follow the same syntactic rules as the relations described above. However, there are some additional points to be noted. These relationships can connect activities with non-activity entities, as well as non-activitiy entities with each other. With the generalization relations, only like entity types can be connected together (e.g., product X *is part of* product Y), while with the aggregation relation, arbitrary entity types can be connected.

Aggregation relations can be specified in two different ways. The first way is simply of the form: {activity} *is part of* {activity}. The second way is required in order to add hierarchical structure. For a process model of any size, there is a need to structure the model into higher and lower level diagrams, where the lower level diagrams act like components (or subroutines) to the higher level. In order to expand an entity into its constituent parts, the *is part of* relation is used. Most commonly (but not necessarily), an activity box in one diagram is expanded in another diagram in order to show detail. If the depth of a box framing an entity is doubled, then this indicates that a lower level diagram exists. As an example of this, see figure 4.2, which contains five activities, each of which is expanded (Figures 4.3 through 4.7).

The final class of relationships contains only one relationship type. This class is a “catch all” for entities which are related in ways that cannot be expressed by any of the pre-defined types. This custom relationship allows the modeler is define the relationship expression as arbitrary text. For example, a custom relationship used in Figure 4.6 is the word *replaces* . Custom relationships are used very sparingly since, as they do not have a pre-defined meaning, they could detract from the clarity of the model.

One entity type, the *junction*, requires additional explanation. Before an activity can be initiated, a set of Boolean conditions (here called a composite) may have to be true. For example, if the Boolean composite *(product A or (agent B and condition C))* is true, then activity X can start. In a similar vein, the activity X may generate as output (condition D or condition E).These relationships are represented diagrammatically as shown in Figure A.2.

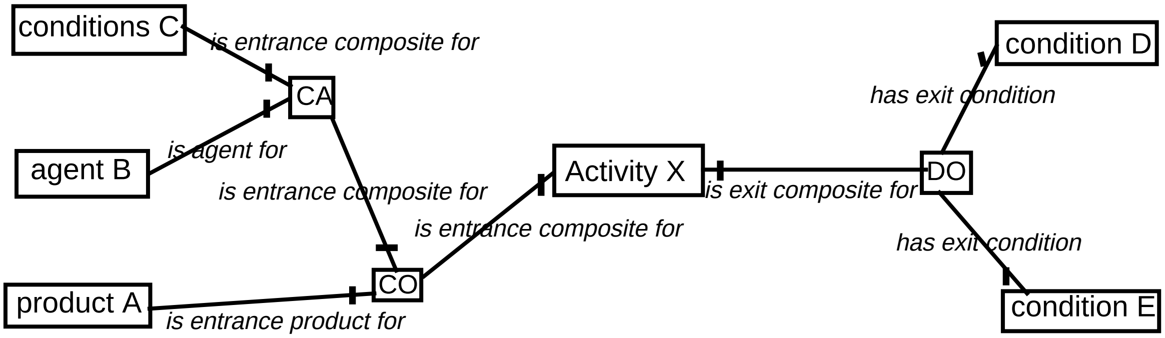


Figure A.2 An Example of Boolean Composites

There are four types of junctions: convergent and divergent, conjunctive ("AND") and disjunctive ("OR"). In the picture, CO implies a convergent "OR", CA implies a convergent "AND" while DO implies a divergent "OR". The final type (not shown in the diagram) is a divergent "AND" (DA). Convergent nodes always precede activities, while divergent nodes always come after activities. It should be noted that, for multiple entities acting directly on an activity, it is assumed that these entities are all ANDed together. This is also true for multiple outputs from an activity.

Some final rules must be described in order to understand the process aspects of this modeling approach. Except through the generalization and aggregation relationships, activities never connect to other activities. For example, from a process point of view, an activity may produce a product as output. The existence of this product now allows a second activity to start. Thus, activities are activated only indirectly through the consequences of other activities (products available, conditions set to true, etc.), not through direct connections between activities. It is thus not appropriate to read the diagrams as flow charts.

Appendix B Unix Scripts for System Build

This appendix contains a shortened version of the file “compile.env.” This file is stored at the top of the tree and is used by the build scripts at each node of the tree to set platform dependent build parameters. The scripts which build and install the project software must be aware of the type of hardware and the software version of the operating system running on the platform. This is the only file that must be altered in order to build and install thesoftware on a particular platform. The platform is selected by using an editor to move the arrow (->) next to platform type. In this example the AT&T 3B15 running Unix version 2.2 has been selected.

```
#
# This file is read by the compile shell script. The version
# pointed to by the arrow is the version we will compile.
#

#
# this is the version for the AT&T 3B15
# running Unix 2.2
#
-)3B15_UNIX_2.2

#
# this is the version for the AT&T 3B15
# running Unix 3.1
#
3B15_UNIX_3.1

#
# this is the version for the scope 1000 80386 “b” and “c”
machines
# running SCO XENIX 2.3
#
SCO_386_2.3

#
# this is the version for the scope 1000 80386 “b” and “c”
# machines running SCO XENIX 3.2
#
SCO_386_3.2

.
.
.
.
.


#
# this is the version for the SUN 3
# running SunOS 4.0.3
#
SUN3_4.0.3
```

```
#
# this is the version for the SUN 4
# running SunOS 4.0.3
#
SUN4_4.0.3
```

Appendix C   Sample Makefile

Make files are description files used by the Unix *make* program. Makefiles are stored at each node of the project’s source tree. Most of the actual work of building and installing the software is done by shell scripts stored with the makefiles at each node in the tree. Shell scripts are used to do most of the compile, build and install functions to avoid the maintenance problems associated with platform variations of *make*.

```
# %Z%USG %M% %I% %G% dist=%Q%

help:

     @echo “\tuse \”make all\” to ./compile link install”
     @echo “\tuse \”make file\” to ./compile link”
     @echo “\tuse \”make clean\” to remove *.o and file”
     @echo “\tuse \”make install\” to install file”

all: install clean

file : compile decl.h hlpdeclare.h errdeclare.h\
      file.o consts.o exist.o args.o subs1.o subs2.o
subs3.o
      @chmod +x compile
      @./compile link

.c.o :
      @chmod +x compile
     ./compile $*
clean:
      @chmod +x compile
      @./compile clean

install: file
      @chmod +x compile
      @./compile install

file.c:
      get SCCS/s.file.c

consts.c:
      get SCCS/s.consts.c

exist.c:
      get SCCS/s.exist.c

args.c:  get SCCS/s.args.c

subs1.c:
      get SCCS/s.subs1.c
```

```
subs2.c:
      get SCCS/s.subs2.c

subs3.c:
      get SCCS/s.subs3.c

decl.h:
      get SCCS/s.decl.h

errdeclare.h:
      get SCCS/s.errdeclare.h

hlpdeclare.h:
      get SCCS/s.hlpdeclare.h

compile:
      get SCCS/s.compile
      chmod +x compile
```

Appendix D Sample Compile/Build/Install Script

This is an example of a compile script which is stored at nodes within the tree. Compile scripts are executed by the makefiles at each node. The compile script determines the platform by consulting a file at the top of the tree; it then adjusts the compile, build, and install parameters and does the installation. This compile script builds and installs the program FILE.

```
#!/bin/sh
# @(#)USG compile 2.4 3/19/91 dist=
# compile

# This is a simplified example of a shell script which is
# responsible for compiling, linking and installing a
# program at one node in the tree. The script is executed
# by the makefile at the mode. The script accepts a set of
# parameters which can be keywords and/or module names. A
# data file (compile.env) at root of the tree is examined
# to determine the hardware/software platform on which the
# build is taking place.
#
# example: compile all link install clean
#
# Set var EXECUTABLE to the program file name
#

EXECUTABLE=FILE

#
# Set ALLMODULES to the list of modules included in the
# program
#

ALLMODULES=”file consts exist args subs1 subs2 subs3”

# Check the compile.env file to determine the platform

case ‘grep “^-)” ../compile.env‘ in

#
# Each section of the “case” statement sets the shell
# variables which will be used to compile, link and install
# the program. The platform type is defined in the
# “compile.env” file and must match a line in one of the
# sections of the case statement.
#

#
# AT&T 3B15 2.2
#
```

```
 *3B15_UNIX_2.2 )
      CCFLAGS=”-O -K sd”
      LDFLAGS=-lproject
      INCLUDE=
      TARGET=/usr/ptssexec
      MODES=700
      UID=project
      GID=emer
      ARCHIVE=
       LINKNAMES=
      ;;
#
# AT&T 3B15 and 3B2 3.x
#
 *3B15_UNIX_3.1 | \
 *3B15_UNIX_3.2_MLS | \
 *3B2_UNIX_3.1 | \
 *3B2_UNIX_3.2 | \
 *3B2_UNIX_3.2MLS )
      CCFLAGS=”-O -K”
      LDFLAGS=-lproject
      INCLUDE=
      TARGET=/usr/ptssexec
      MODES=70
      UID=project
      GID=emer
      ARCHIVE=
      LINKNAMES=
   ;;
#
# SUN
#
 *SUN3_4.0.3)
      CC=/usr/5bin/cc
      CCFLAGS=”-O -f8881”
      LDFLAGS=”-lproject /usr/5lib/libc.a”
      INCLUDE=
      TARGET=/usr/ptssexec
      MODES=700
      UID=project
      GID=emer
      ARCHIVE=
      LINKNAMES=
      ;;
#
# SUN
#
 *SUN4_4.0.3 | \
 *SUN386I_4.0.3 )
      CC=/usr/5bin/cc
      CCFLAGS=”-O”
      LDFLAGS=”-lproject /usr/5lib/libc.a”
      INCLUDE=
      TARGET=/usr/users/hms
      MODES=700
      UID=project
      GID=emer
      ARCHIVE=
      LINKNAMES=
      ;;
#
# EASY DATA
#
 *SCO_386_2.3 | \
 *SCO_286_2.2 | \
 *PS2 )
      CCFLAGS=”-Oat -DPCAT”
      LDFLAGS=-lproject
      INCLUDE=
      TARGET=/usr/ptssexec
      MODES=700
      UID=project
      GID=emer
      ARCHIVE=
      LINKNAMES=
      ;;
#
# EASY DATA
#
 *SCO_386_3.2 )
      CCFLAGS=”-Oat -DSCO_UNIX”
      LDFLAGS=-lproject
      INCLUDE=
      TARGET=/usr/ptssexec
      MODES=700
      UID=project
      GID=emer
      ARCHIVE=
      LINKNAMES=
      ;;
#
# Many of the platform descriptions were removed from here
# to simplify the example.
#

#
# ERROR - specified platform (in compil.env) does not have
# a matching entry in the CASE statement!
#
   *)
      echo “Can’t identify the machine. Exiting.”
      exit 1
      ;;

esac

#
# Check the input parameters to determine what must be
# done. Possible values in the parameter list are: all,
# link, install, clean or the names of specific modules to
# compile.
#

if [ -z “${1}” ]
then
   CPCUNITS=${ALLMODULES}
else
   CPCUNITS=${*}
fi


if [ “${1}” = “all” ]
then
   CPCUNITS=${ALLMODULES}
   shift
   CPCUNITS=”${CPCUNITS} ${*}”
fi

OBJ=

cd ‘dirname ${0}‘

#
# If the compiler was not specified, then make it “cc”
#
if [ -z “${CC}” ]
then
   CC=cc
fi

#
# Process the parameter list, set flags for LINK, INSTALL
# or CLEAN and try to copile everthing else.
#

for FILE in ${CPCUNITS}
 do
   case ${FILE} in
      link)
         LINK=1
         continue
         ;;
      install)
         INSTALL=1
         continue
         ;;
      clean)
         CLEAN=1
         continue
         ;;
      * )
         echo “${CC} -c ${CCFLAGS} ${INCLUDE}${FILE}.c”
```

```
        ${CC} -c ${CCFLAGS} ${INCLUDE} ${FILE}.c

        if [ ! ${?} -eq 0 ]
           then
              exit 1
        fi
        OBJ=”${OBJ} ${FILE}.o”
        ;;
    esac

 done

#
# If the OBJ list is empty (not constructed by the compile
# step) then make the OBJ list for ALLMODULES.
#

if [ ! “${OBJ}” ]
then

 for FILE in ${ALLMODULES}
 do
   OBJ=”${OBJ} ${FILE}.o”
 done
fi

#
# Do either a link or an archive based on whether or not
# the shell variable EXECUTABLE contains an executable
# name.
#

if [ “${LINK}” = “1” ]
then


 if [ -z “${EXECUTABLE}” ]
 then
   echo “Archiving ${TARGET}/${ARCHIVE} ${OBJ}”
   /usr/localbin/archive rv ${TARGET}/${ARCHIVE} ${OBJ}
   if [ ! ${?} -eq 0 ]
   then
      exit 1
   fi

 else
   echo “Linking cc ${OBJ} ${LDFLAGS} -o ${EXECUTABLE}”
   cc ${OBJ} ${LDFLAGS} -o ${EXECUTABLE}
   if [ ! ${?} -eq 0 ]
   then
      exit 1
   fi
 fi
fi
```

```
#
# Install the program if specified and if EXECUTABLE
# contains an executable name. Otherwise, install the
# program in an archive (usually libproject.a).
#

if [ “${INSTALL}” = “1” ]
then

 echo “Installing”
 if [ -z “${EXECUTABLE}” ]
 then
   echo “Archiving ${TARGET}/${ARCHIVE} ${OBJ}”
   /usr/localbin/archive rv ${TARGET}/${ARCHIVE} ${OBJ}
   if [ ! ${?} -eq 0 ]
   then
      exit 1
   fi

 else
   rm -f ${TARGET}/${EXECUTABLE}
   for FILE in ${LINKNAMES}
   do
      rm -f ${FILE}
   done
   cp ${EXECUTABLE} ${TARGET}/${EXECUTABLE}
   if [ ! ${?} -eq 0 ]
   then
      exit 1
   fi
# on some platforms “chown” must be executed as a
# priviledged (su) program?!
   chown ${UID} ${TARGET}/${EXECUTABLE}
   chgrp ${GID} ${TARGET}/${EXECUTABLE}
   chmod ${MODES} ${TARGET}/${EXECUTABLE}
   ls -l ${TARGET}/${EXECUTABLE}
   for FILE in ${LINKNAMES}
   do
      echo “linking ${TARGET}/${EXECUTABLE} ${FILE}”
      ln ${TARGET}/${EXECUTABLE} ${FILE}
   done
 fi
fi
#
# Clean out the .o files and the executable
#
if [ “${CLEAN}” = “1” ]
then
 echo “Cleaning”
 rm -f *.o
 rm -f ${EXECUTABLE}
fi
```

Appendix E CCB Change Request Form

=======CCB REPORT ID #nnnn========

(nnnn is inserted automatically by CCBNEW program)

CREATED BY:

(This field is automatically filled when CCBNEW is run, the program inserts the user’s name and a time stamp.)

MODIFIED BY:

(This field is automatically filled when CCBOLD nnnn is run and will be repeated as many times as necessary.)

SUBJECT:

(Short one line description of problem or update request.)

PROBLEM DESCRIPTION:

(Description of the symptoms and circumstances surrounding the failure or a request for a new feature.)

ANY PREVIOUS CCB REFERENCES:

(Usually empty but may be filled in by the developer. Provides a cross reference to related CCB items.)

GENERAL DESCRIPTION OF CHANGE:

(Non-technical description of the changes made. This field provides input to the non-technical members of the CCB board.)

LIST OF OS RELEASES AFFECTED:

(Machines and operating system versions that the change must be applied to.)

MANUAL PAGE CREATED/MODIFIED:

(Formal manual pages affected by the changes and the location of the updated manual pages.)

LIST OF SOURCES CHANGED WITH SCCS DELTAS:

(Pathnames to the new SCCS files including the SCCS delta numbers of the changes.)

DETAILS OF CHANGES MADE (IF APPROPIATE):

(Technical description of what was changed. This section might contain small code segments and is primarily aimed at developers who may make follow up changes to the same items.)

TEST HISTORY AND SCENARIO FOR CHANGES:

(When, how and on what platforms were the changes tested. The test team will use these notes to develop test scenarios for the next release.)

LIST OF INSTALLED MODULES CHANGED:

(List of executable affected by the change. The list may be long if the changes were made to one of the library modules.)

ANY NON-STANDARD INSTALLATION PROCEDURES:

(Anything that involves more than “make install”. This would include network changes, OS changes or anything that might require a reboot to install etc.)

ANY DOCS FOR RELEASE NOTES:

(This information will be included in the release notes with the next major release. They are used to alert system administrators and users to changes that affect the way executables behave or to document new features that have been added since the last release.)

Appendix F Example of an Initialized CR Form

This is an example of a completed CR form. Several of the fields have been changed to support automatic processing of the CM auditing functions.

================ CCBID #1743 ================

```
CREATED: by radavis on Wed Nov 20 19:01:13 GMT 1991

SUBJECT: eth_load and proj_getty for Sun OS, DGUX

PRODUCT: project

PROGRAMMER: radavis

PROBLEM DESCRIPTION:

 For the Sun, eth_load needs changing to know about GUL,
and also about “rotaring”. For the Data General, the
programs needs an initial port.

 On the Data General, proj_getty requires a tricky
adaptation. You can’t open the slave side of a pseudo tty
on the Aviion unless the controller side is already open.
The solution proposed here is to have proj_getty hold up,
in an infinite sleep, until eth_load awakens it after
opening the controller. Proj_getty registers his pid in
/project/gettypids and locks same (so eth_load will know
the proj_getty is still alive).

The network part of the code is hard to read because of
excessive #ifdefs. These have now been encapsulated into
the module “proj_list”.

::::::::::

ANY PREVIOUS CCB REFERENCES:

::::::::::


GENERAL DESCRIPTION OF CHANGE:

   Add GUL to Sun version.
   Add rotaring to Sun version.
   Add proj_getty sleep, eth_load wakeup trick for DGUX
version.
   Hack out most of the network calls and replace with
proj_list.

::::::::::
```

LIST OF MACHINE MODELS, OS RELEASES AFFECTED:

Sun OS
DGUX

::::::::::

MANUAL PAGE CREATED/MODIFIED (Yes/No/Not-Applicable)

:::::::::::

LIST OF SOURCE MODULES CHANGED:
u SYS/eth_load/SCCS/s.eth_load.c 2.10
u SYS/proj_getty/SCCS/s.proj_getty.c 2.3

::::::::::

DETAILS OF CHANGES MADE (if appropriate):

1. “@(#)SHAREDPORTS defined -- eth_load.c” SCCS string added if SHAREDPORTS #ifdef is true.
2. Module changed to key off standard project/define.h GUL define vice GRAND_UNIFIED_LOGIN.
3. Use LIKE43SOCKETS define instead of TLISOCKCOMP.
4. Ported to DGUX.
5. Implemented rotaring on the Sun. (Needed #ifdef for Sun OS 4 locking bug.)
6. Turn off catcher for SIGCLD during network socket close. If the network close is interrupted it doesn’t complete properly.
7. Added logic to sense the projterm “panic” character, to increase the liklihood eth_load will exit when projterm goes away and KEEP_ALIVE doesn’t work.
8. Add logic to awaken proj_getty under DGUX.

::::::::::

TEST HISTORY & SCENARIO FOR THIS CHANGE:
Tested by daily use on Quark, Fillet and Quasar.

::::::::::

LIST OF INSTALLED MODULES CHANGED:
/usr/localbin/eth_load
/usr/localbin/proj_getty

::::::::::

ANY NON_STANDARD INSTALLATION PROCEDURES:

::::::::::

ANY DOCUMENTATION FOR THE RELEASE NOTES:

```
::::::::::

)From: MINUTES/ccb1126
   #1743 eth_load and proj_getty for Sun OS, DGUX : All OK :
   status=INC_project
```

References

[Christie 92] Alan M. Christie, "A Graphical Process Definition Language with Application," Software Engineering Institute, To be published.

[Paulk 91] Mark C. Paulk et al., "The Capability Maturity Model for Software," Software Engineering Institute Technical Report CMU/SEI-91-TR-24, ADA240603, August 1991.

[Sobel 89] Mark G. Sobell, "A Practical Guide to the Unix System," Benjamin Cummings Publishing Company Inc., Second Edition, 1989.

[Softool 91] Softool Corporation, Goleta, CA, "CCC/DM Turnkey User’s Manual," Revision nn3-111, June 1991.

[Weber 91] Charles V. Weber et al., "Key Practices of the Capability Maturity Model," Software Engineering Institute Technical Report CMU/SEI-91-TR-25, ADA240604, August 1991.