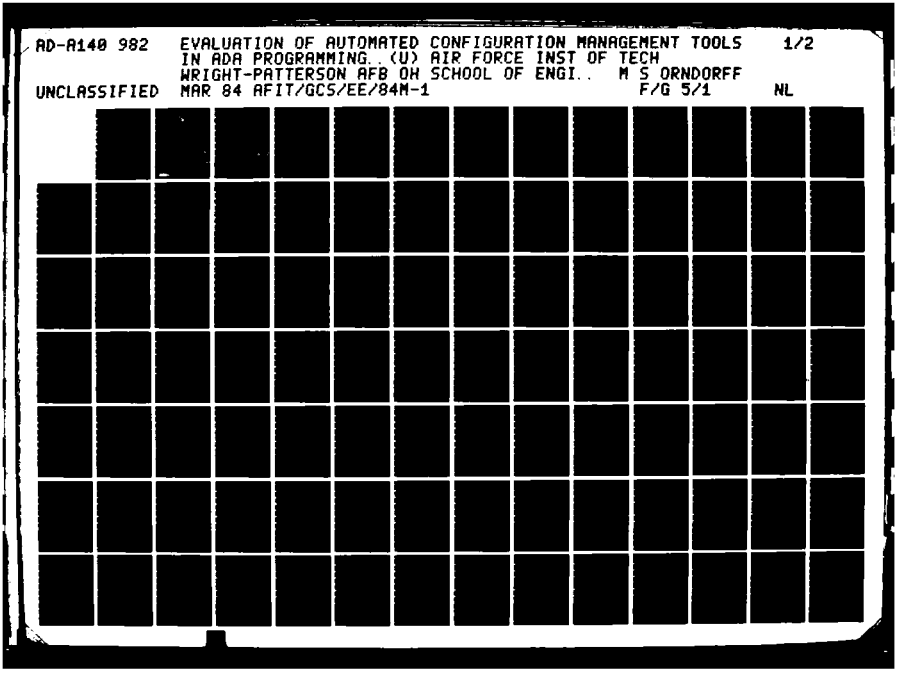
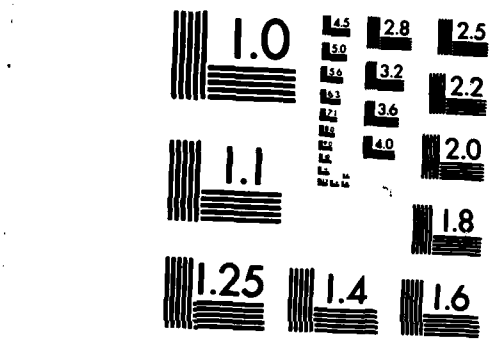
AD-A140 982 EVALUATION OF AUTOMATED CONFIGURATION MANAGEMENT TOOLS IN ADA PROGRAMMING..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. M S ORNDORFF

1/2

UNCLASSIFIED MAR 84 AFIT/GCS/EE/84M-1 F/G 5/1 NL





EVALUATION OF AUTOMATED CONFIGURATION MANAGEMENT TOOLS IN ADA PROGRAMMING SUPPORT ENVIRONMENTS

THESIS

Mark S. Orndorff
Captain, U.S. Army

AFIT/GCS/EE/84M-1

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY (ATC)
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/EE/84M-1

EVALUATION OF AUTOMATED CONFIGURATION MANAGEMENT TOOLS IN ADA PROGRAMMING SUPPORT ENVIRONMENTS

THESIS

Mark S. Orndorff
Captain, U.S. Army

AFIT/GCS/EE/84M-1

Approved for public release; distribution unlimited.

AFIT/GCS/EE/84M-1

EVALUATION OF
AUTOMATED CONFIGURATION MANAGEMENT TOOLS
IN ADA PROGRAMMING SUPPORT ENVIRONMENTS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

Mark S. Orndorff, B.A.
Captain, US Army

March 1984

Preface

The Army has contracted with SofTech Incorporated to develop the Ada Language System (ALS), which is the Army's initial Ada Programming Support Environment. The Army contracted with the Air Force Avionics Laboratory to provide independent evaluation of the ALS. During the Air Force's evaluation, the need arose for developing evaluation criteria for the complex task of configuration management. This need coupled with my desire to learn more about configuration management and the Ada language program led to the selection of this thesis topic.

I would like to thank my advisor, Major Michael R. Varrieur, for all the time and guidance he has given me. His ideas and suggestions during the course of the project were most helpful. I would also like to thank my thesis-committee members, Captain Patricia Lawlis and Doctor Henry Potoczny. Their suggestions and comments were very valuable.

Deep gratitude is also expressed to Mrs. Virginia Castor of the System Avionics Division, Support Systems Branch, Air Force Wright Aeronautical Laboratories, who originally proposed this thesis topic and provided the necessary resources and continuous guidance throughout the project.

Finally, I would like to thank my wife, Beth, for all the support and encouragement she has given me.

Mark S. Orndorff

Contents

List of Figures

List of Tables

Abstract

This investigation studied the task of configuration management of computer software systems. First, a detailed definition of configuration management from the perspectives of project management and project engineers was developed. This definition was used to conduct a requirements analysis of the support required in automated programming environments for the configuration management task. Based on these requirements, evaluation criteria were developed that were appropriate for the evaluation of configuration management tools designed to satisfy the 1980 Stoneman requirements document. These evaluation criteria were used to evaluate the November 1983 release of the Army's Ada Language System.

The requirements and evaluation criteria developed in this thesis are designed to provide designers and purchasers of Ada Programming Support Environments (APSE) with the tools necessary to determine the effectiveness of an APSE implementation in supporting the task of configuration management of large software projects developed for embedded computer systems.

I INTRODUCTION

This thesis will study the concept of automated configuration management systems as part of Ada Programming Support Environments. Although all Ada programming environments should include an automated configuration manager, the Department of Defense requirements document, (Stoneman), does not specify the capabilities and characteristics of the configuration management tool. This thesis will present a working definition of configuration management and a set of metrics for evaluation of configuration management tools produced for the Department of Defense.

Background

The complexity and cost of software developed for the DoD has increased dramatically over the last few years, with more than half of the software costs associated with embedded computer systems (Wegner, 1980:408, Stuebing, 1980:10). At the same time, attempts to create new systems using existing software components have increased the complexity of the maintenance task. The Department of Defense has made several attempts at improving the quality and reducing the development and maintenance costs of software systems.

In 1975, the DoD Common High Order Language program was initiated with the expressed goal of developing a high order language for all DoD embedded computer systems. From the

beginning, the use of this new language as a means of introducing effective software development and support environments was considered a major benefit of the program (Stuebing, 1980:3).

The Ada Language Initiative. The DoD has developed a standard programming language, Ada, which is currently in the process of being adopted as the preferred language for all DoD software projects. The DoD-wide use of Ada will eventually reduce the cost of programmer training and allow portability of programmers and programs (Buxton, 1980:67). The major benefits from Ada result from Ada's appropriateness to military applications, from the portability of a machine independent language, from the availability of software resulting from the acceptance of Ada for non-military applications, as well as from the use of Ada as a mechanism for introducing effective software development and support environments for developing military systems (Stuebing, 1980:3).

In support of this last benefit of the Ada program, the DoD began a requirements analysis for a complete programming support environment. After an analysis of current programming support environments, the DoD published an initial requirements document, called Pebbleman, in 1978. A second version, called preliminary Stoneman, was published in 1979, and a final requirements document was published in 1980.

The 1980 Stoneman document presents a model for Ada programming environments that will be used by designers of initial environments. The Stoneman design addresses the problems associated with software development while realizing the limitations of the current state of the art in programming environments. The Stoneman proposal calls for developing an open-ended environment that is initially consistent with the state-of-the-art (and therefore immediately realizable), and supports easy expansion as new capabilities become available. The software problems addressed by Stoneman will be presented in the following section, and then the Stoneman proposal will be presented in more detail.

Current Software Development Practices. The computers used for embedded systems often do not support software development, thus the common practice has been to develop software on a host machine and perform the testing on a combination of simulators and the actual target machine. This testing procedure results from the target system's development occurring concurrently with the software development. Testing of software developed for weapon systems often relies heavily on simulations due to the high cost of tests using the actual target system. While testing a software system, various changes occur in response to errors detected during the testing process. These changes will result in the introduction of new versions of certain

components of the software system. The changed components and other components related to the changed components must go through a re-compilation process. The programmer currently performs this cycle manually, with the possibility of introducing errors by failing to re-compile all dependent modules.

During the life cycle of a software system, the software requirements often evolve to include development of several different versions for various modifications of the original system. A project manager must maintain each of these versions throughout the life cycle of the system. The maintenance of each of these versions involves the possible introduction of new versions resulting from additional corrections or revisions. The project manager often must resort to managing this large amount of inter-related software manually with his success depending on his own management ability. The cost of this maintenance often reaches as much as 80 percent of the life cycle cost of the system (Stuebing, 1980:10).

The tools used to support the software life cycle generally consist of a compiler, linker, and editor. The development of special purpose tools for a specific project has not been coordinated to allow for the re-use of these often expensive tools. These tools do not provide adequate support for the needs of long-term system maintenance.

The Ada Program Support Environment (APSE). Stoneman specifies the requirements for an Ada Programming Support Environment (APSE), with the approach of developing software on a host system for use on one or more target systems. The APSE requirements were designed to provide support for the specific problems relating to development and long term maintenance of software for embedded systems.

The APSE design goal was to reduce the redundant development of the tools used for the development of embedded systems by providing a complete set of tools that will support the entire software life cycle, including long term maintenance and modification. Rather than produce specific standards for all features of all programming environments, the approach taken in Stoneman was to present a standard structure based on four layers, as shown in figure 1, with a specified minimum set of tools required for all Ada programming environments.

The first layer, level 0, consists only of the hardware and host software. The second layer, level 1, is called the Kernel Ada Program Support Environment (KAPSE). This layer consists of the database, and communications and runtime support functions. This layer provides a machine-independent portability interface that will be standardized for all Ada environments.

The third layer, level 2, is called the Minimal Ada Program Support Environment (MAPSE). This layer consists of

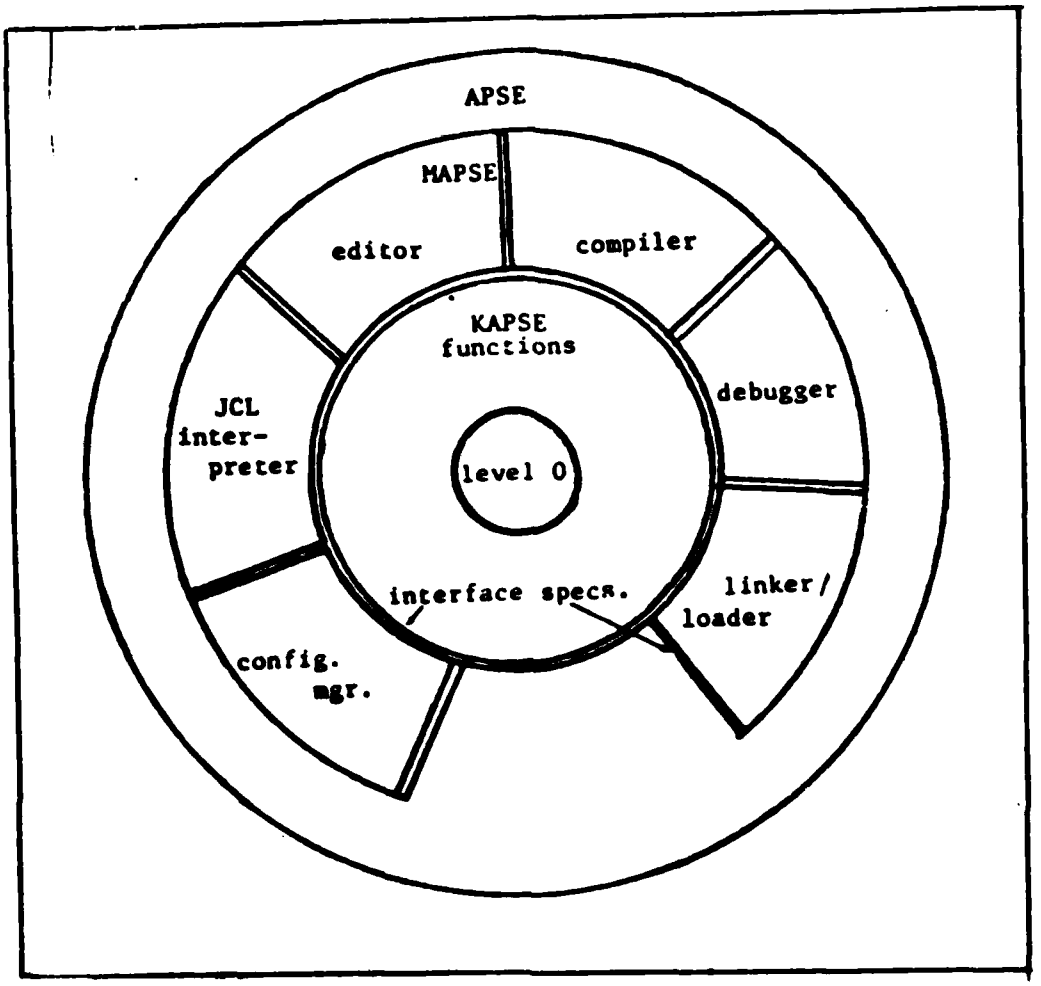


Figure 1. The APSE Structure (Stoneman, 1980:Figure 1.F)

a compiler, debugger, linker-loader, JCL interpreter and a configuration management system. The MAPSE provides a minimal set of tools, both necessary and sufficient for the development and maintenance of Ada programs. The fourth layer, level 3, is called the Ada Program Support Environment (APSE). This layer contains the extensions to the MAPSE that provide support for particular applications

or methodologies. This layered approach to an APSE should provide the basis for the development of tools portable to any Ada environment. The tools in the APSE must support the initial software development and allow for future enhancements and modifications of the developed software for use on unpredicted future systems.

Initial Program Support Environments are currently being developed separately for the Army and the Air Force. The Army has released its Ada Language System (ALS) for evaluation by the Air Force Avionics Laboratory. The Air Force's Ada Integrated Environment (AIE) is still in the development stage. These initial environments should represent the state-of-the-art in programming environments and will become the basis for evaluation of future environments developed for the DoD.

Configuration Management. Software Configuration Management is the discipline of identifying the functional and physical characteristics of a computer software item at discrete points in the software life cycle to control changes and maintain integrity. Configuration management provides the means for program managers to predict the impact of changes to computer software and to incorporate changes in a timely manner.

From a project manager's viewpoint, configuration management is a well defined discipline with specific contractual requirements. For purposes of defining a program sup-

port environment, configuration management takes a broader, and less well specified, definition. Stoneman defines configurations as different collections of objects in a project brought together to form different groupings. These configurations consist of two types. First, some configurations exist as consecutive releases with one being the result of revisions of the other. Second, some groups of configurations coexist, such as separate models, resulting from various target systems or user requirements (Stuebing, 1980:24).

The configuration manager is the software development tool that is involved in all activities related to the creation, modification, retrieval, archiving, and generation of all software items. The configuration manager may also maintain various dependencies between modules that permit automatic recompilation, regression testing, and other features available in advanced programming environments. In an integrated program support environment, the task of configuration management may require the interaction of several tools.

The Stoneman requirements document states that configuration control is a "crucial problem" and requires that all APSE's include a configuration manager, but never defines the features or capabilities of a configuration manager. Although Stoneman establishes the need for some level of automated support for configuration management, it does not

specify the actual features or the method of implementation for a configuration management tool.

Problem Statement

This thesis will study the literature on configuration management systems and determine the characteristics of contemporary configuration management systems. Detailed evaluation criteria will be proposed. This information will be used to evaluate the Army's Ada Language System. Recommendations for future modifications and enhancements of the ALS will be presented. These recommendations will be useful in the design and development of future APSE's developed for the DoD. This study is an initial attempt at evaluating modern programming environments. This effort will become a part of the Avionics Laboratory's overall evaluation of the ALS. The methods developed in this thesis will be useful for the development and evaluation of future APSE's by the DoD.

Scope

This thesis will study the discipline of configuration management of software systems. All features of the Army's Ada Language System related to the task of configuration management will be studied and evaluated. The Configuration Management System will be studied in the following areas: (a) partitioning the project workspace, (b) user defined access control authorization to configuration elements, (c)

configuration management of versions and revisions, (d) configuration management of multiple-target systems, (e) identification and maintenance of baseline products, (g) archive and restoration procedures for configuration elements.

Summary of Current Knowledge

Several systems have been developed for automating support of software development. The systems currently developed do not represent a complete set of tools for weapons systems, but will provide a basis for comparison for the features common to the ALS.

Standards

There are currently no standards for evaluating the performance of a software development support activity. The development of evaluation standards is a major objective of this thesis and will be based on methods proposed in the literature.

The final Stoneman document specifies general requirements, but does not present evaluation criteria. As the development of programming environments continues, the DoD must have a set of metrics to determine if these environments actually meet the Stoneman requirements and the needs of a programming team. There has not been any comprehensive study to evaluate the state-of-the-art for programming environments and propose specific requirements to be evaluated and the criteria for their evaluation.

Approach

The first step of this study will be to define configuration management. Next, the areas that need to be evaluated will be identified and described. Once appropriate criteria are developed, they will be documented and presented to the sponsoring agency for approval. Once an approved set of evaluation criteria is completed, the Army's Ada Language System will be evaluated and the results presented. The evaluation criteria and the actual evaluation will be given to the sponsoring agency for use in their evaluation of the ALS.

Materials and Equipment

Access to the Ada Language System hosted on the AVSAIL VAX-11/780 is required and will be provided. Appropriate reference material is also available.

II Configuration Management

The definition of configuration management given in the introduction to this thesis gives a broad overview of the discipline of configuration management without specifying those actions that must be taken by various members of a software development team to accomplish the task of configuration management. In this chapter, a life cycle model proposed by the Navy (Dept of the Navy, 1982) will be used to study the specific actions that must be taken to achieve effective configuration management. After reviewing the life cycle model, the management requirements will be reviewed as specified in current regulations and military standards. This section will summarize the responsibilities of the software development discipline formally labelled configuration management. Next, the activities of the other members of the software development team will be studied to determine their requirements for a software development environment that supports incrementally developed computer software.

The Navy's Software Life Cycle Model

The Navy sponsored a program to develop a software engineering environment that would provide a DoD-wide standard for a software engineering environment. The fundamental requirements for this environment were to (1)

support the entire life cycle, (2) be methodology driven, (3) provide some type of support for existing projects.

In the preliminary work of this project, various life cycle models were studied to determine a basis for the development of the environment. The Navy study concluded that life cycle models currently used by the Navy (see figure 2) did not provide an adequate model for the continuing evolution of software from the time of the first release to the time when the last existing version of the software system is retired.

The Navy's research team determined that a new model for the software life cycle must be developed before a software engineering methodology could be developed and an environment standardized. The proposed life cycle model is based on four fundamental ideas: incremental development, early prototyping, extended correctness analysis, and management integration.

Incremental development. The major deviation from more traditional life cycle models in the Navy's proposal is the concept of incremental development. The idea of incremental development is based on the development of a software system in small, manageable increments, with each new increment treated as a new system with additional functions over its predecessor (see figure 3). Each increment is subject to phases similar to the traditional life cycle model

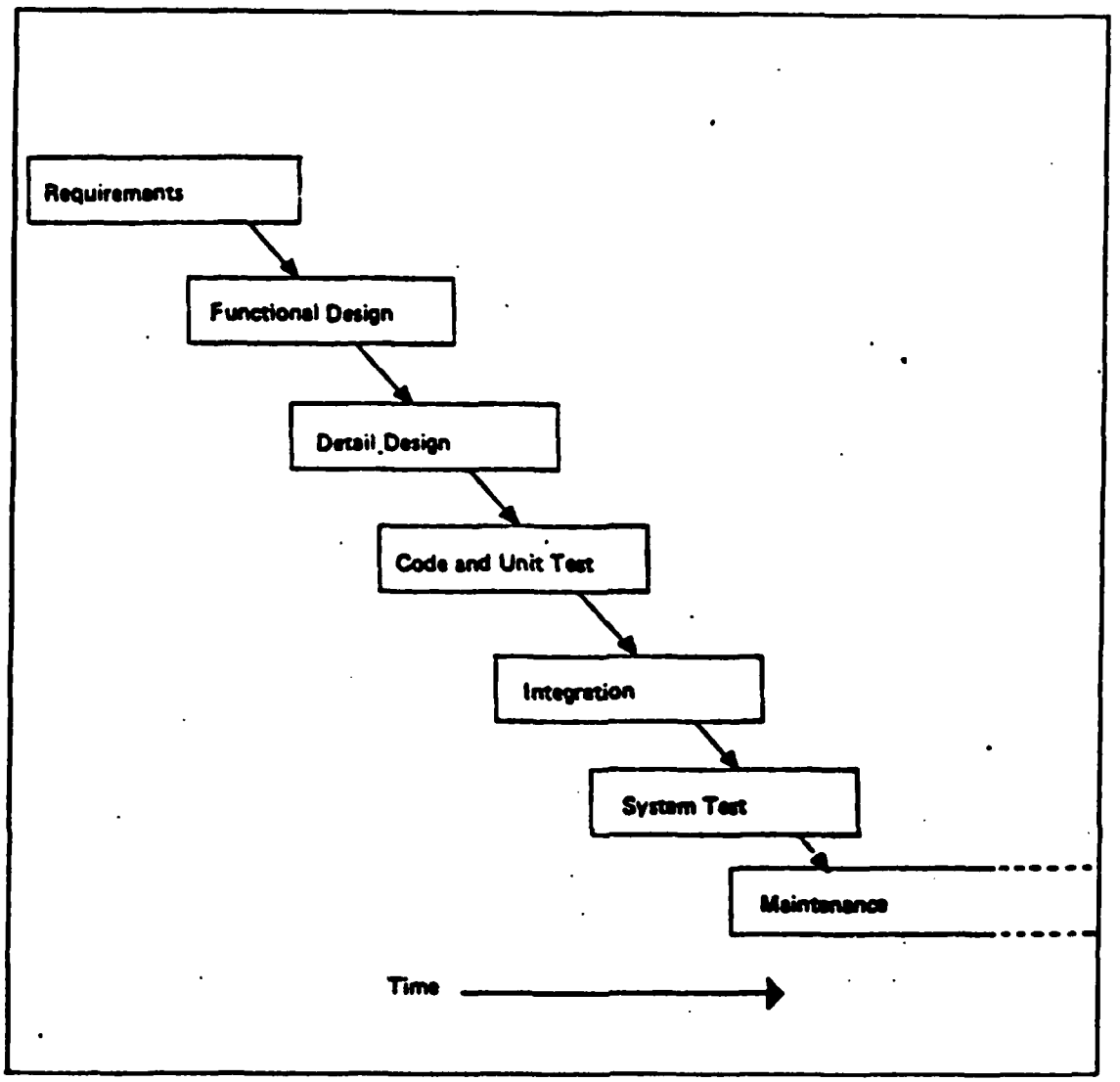


Figure 2. A Traditional Life Cycle Model (Dept of the Navy, 1982:I-5)

(requirements analysis, specification, design, implementation), (see figure 4).

The incremental approach to software development provides efficient management of changes in response to evolving requirements. The incremental model addresses the crucial issue of evolving software without resorting to the

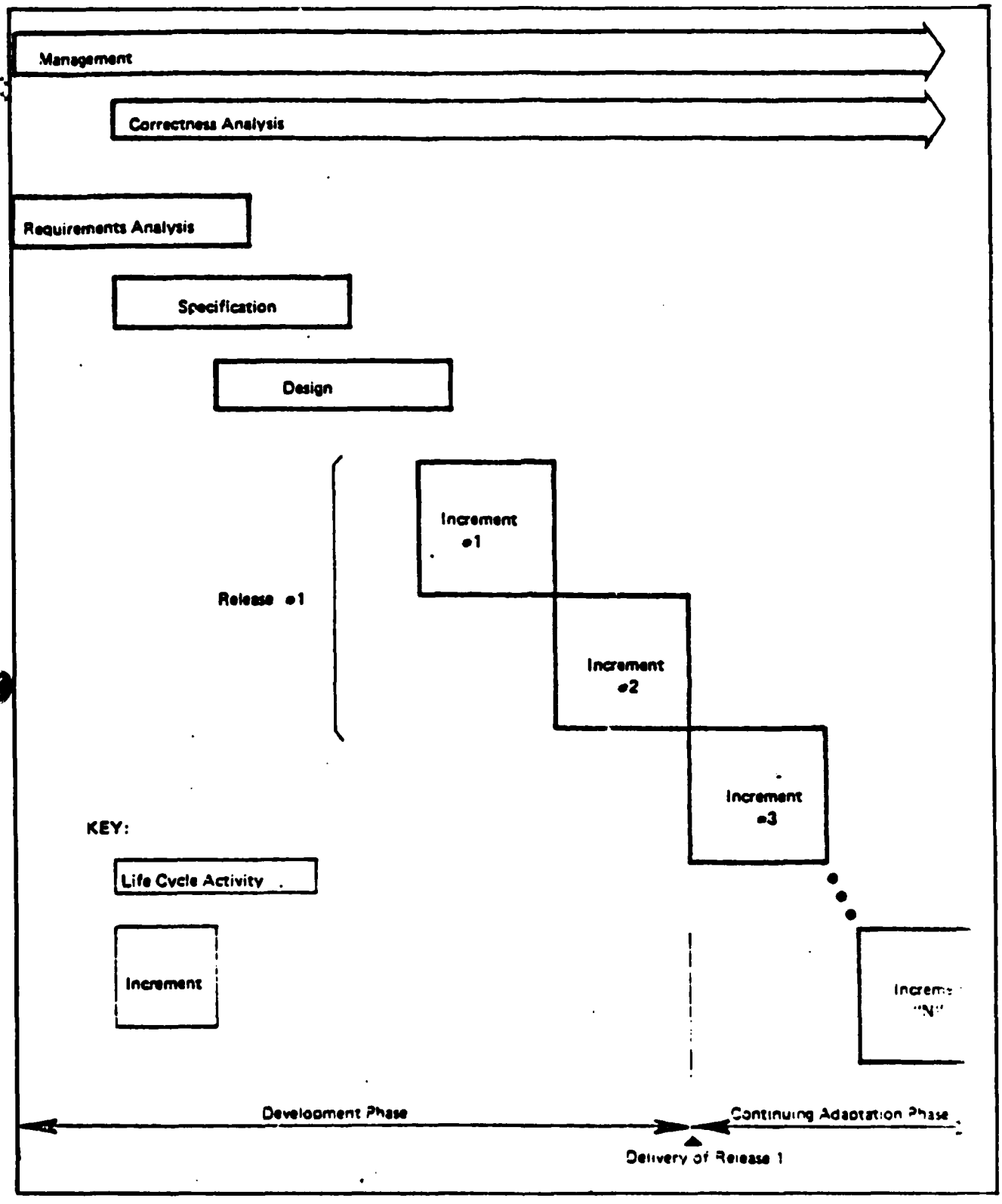


Figure 3. The Incremental Life Cycle Model (Dept of the Navy, 1982:I-10)

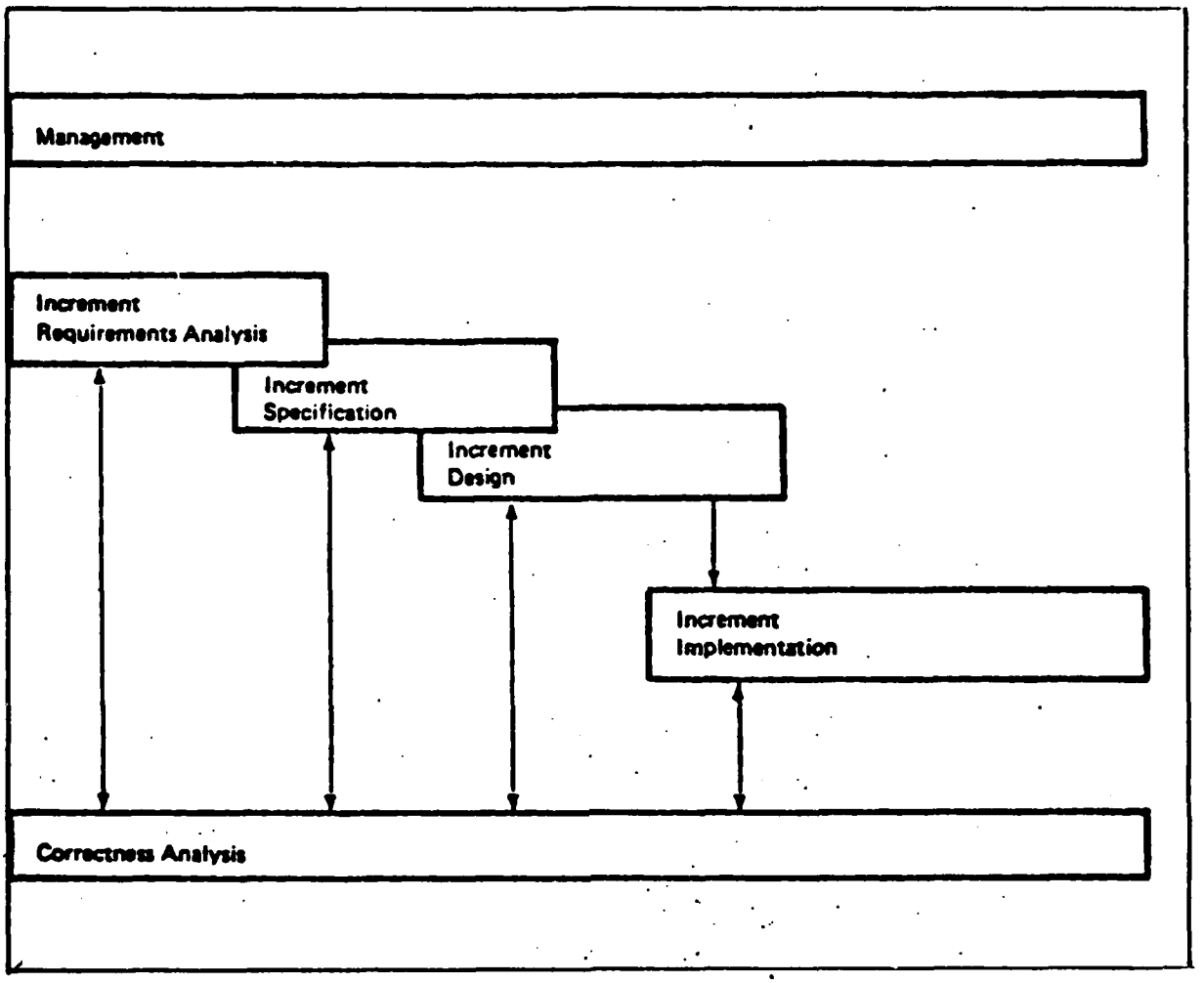


Figure 4. Structure of a Single Increment (Dept of the Navy, 1982:I-11)

umbrella phase labelled "maintenance" in the traditional life cycle model (see figure 2). This single maintenance phase covered the large majority of time, cost, and problems in software projects developed under the traditional life cycle model. The traditional life cycle model did not provide guidance for the activities necessary for insuring efficient maintenance of the software product.

The incremental development model provides excellent support for the concepts of configuration management. The configuration management concept of baselines, referring to a reference point or plateau in the development of a system (Berrsoff, 1979:98), can be mapped directly into the incremental model. This mapping provides a logical relationship between management products and design products.

Early Prototyping. The second fundamental idea of the Navy's model is the need for early prototyping. Early prototyping calls for the development of increments of the software system that satisfy a small portion of the requirements, but are available for testing and evaluation early in the project development. The cost of correcting errors in software system is much less if these errors are detected early in the development. Early prototypes provide a means for identifying errors in requirements and specification early in the development and therefore greatly reduce the cost of corrections.

Early prototyping provides for verification and validation activities early in the development process, to include possible interaction with the intended users. Ill-defined or misunderstood requirements can be identified and further clarified for use in developing the next increment.

This application of the incremental approach places additional responsibility on the configuration manager by

requiring the development of what is currently viewed as a complete life cycle set of configuration management products during the production of these early prototypes. These initial products are placed under configuration management control with later increments managed by repeatedly stepping through the modified traditional life cycle model, as required for incremental development.

Extended Correctness Analysis. The next concept considered in the Navy's model was extended correctness analysis. Traditional life cycle models have considered testing as a single phase in the life cycle, performed after coding is complete (see figure 2). This practice leads to late identification of errors and costly corrections. Even in the incremental model, with this type of testing possible after coding of the first prototype, errors are ignored longer than necessary and costs are increased. For this reason, the Navy's model calls for on-going verification and validation throughout the life cycle (see figure 3).

The practice of extended correctness analysis will require extensive information from the configuration manager to be able to track requirements through to specification, design, and implementation so that verification and validation can be accomplished.

Management Integration. The last main concept considered in the Navy's analysis was management integration. Previous life cycle models have not addressed

the role of management in the software development process. This new model considers management as an ever-present activity requiring data (e.g. frequency of error reports, average time to detection) and directing activity (see figure 3). This view of management will rely on information available from the configuration manager, and management direction will determine the structure of future configurations.

This concept of management control throughout the life cycle requires control over the products of each activity within each increment of the life cycle. This control is achieved by creation of baselines with changes to a baseline controlled by the configuration manager, under the direction of management. A formal change control process must be managed including the tracking of various changes through the stages of approval and implementation. This tracking of changes is the responsibility of the configuration manager.

Management View of Configuration Management

From a project manager's perspective, software configuration management is the discipline of identifying the functional and physical characteristics of a computer software item, at contractually specified points in the software life cycle, to control changes and maintain integrity. Configuration management provides the means for program managers to predict the impact of changes to

computer software and to incorporate changes in a timely manner.

For the purposes of software configuration management, the terms computer software item and computer program configuration item (CPCI) both refer to any collection of computer software that satisfies an end-use function and is designated by the contractor for configuration management. Individual configuration items consist of a group of computer program components (CPC).

Throughout a system life cycle, configuration items are defined, developed and modified based on user requirements and concurrent system developments (e.g. changes in hardware design may cause extensive changes in the software design). The likelihood of a *given function* to change determines the best method of implementation. Since software changes cost considerably less than hardware changes, a function that may change, either during development or after the first release, often requires a software implementation. Because of the highly changeable nature of software products, software development requires special management practices to monitor the development of a software system, and keep track of the historic evolution of current systems. The discipline called configuration management handles the problem of managing software development.

Configuration management must provide the program manager with the information to determine the impact of

proposed changes in system hardware and software on the software development, and to incorporate these changes in an expeditious manner. Within configuration management, configuration identification, configuration change control, and configuration status accounting work together to achieve the stated goals.

Configuration Identification. Configuration identification describes the functional and physical characteristics of configuration items and CPC's. From these characteristics, configuration identification provides information on the internal composition of a configuration item so that managers can quickly determine what affect proposed changes will have on individual system components. Tracking specific requirements with the configuration items that implement them, provides the means for tracing requirement changes through the system. Configuration identification is accomplished by a series of reports representing progressively finer levels of detail of the system design.

The first report maintained by the configuration manager is the System Performance and Design Requirements. This document represents the procurring activity's definition of the product to be produced, giving detailed specifications of the function, reliability requirements, maintenance and support needs, and the environment in which the product will operate (McCarthy, 1980:44).

The next report, prepared by the developing organization, is the Computer Program Development Specification (Part I or Type B5). This document includes the general information flow presented in a block diagram, interface requirements, an expandability plan, a test plan, and a reliability plan (McCarthy, 1980:45). The Part I specification is the procurring activity's key contractual compliance instrument to govern computer program acquisition (Searle, 1977:40).

The third report maintained under configuration management control is the Computer Program Product Specification (Part II or Type C5). This document provides a complete description of each computer program giving the function of each module, the global data characteristics and the impact of each module on the global data, and a description of the input and output of each module. This document is placed under configuration management control after it is approved.

Configuration Change Control. Configuration change control provides the capability for processing changes to the software system. By classifying changes and monitoring resource requirements, change control provides the capability for change tracking and traceability. At the CPC level, change control includes the identification of all CPC's affected by a change to a given CPC. This information determines the testing requirements for changed modules.

For purposes of configuration management, proposed changes are classified as either Class I or Class II. Since the processing requirements differ between these two classes of changes, they will be discussed briefly to point out the products and relations that must be handled by the configuration manager.

A change is designated as Class I if it affects a technical requirement from the Part I specification, the contract schedule, or costs (Searle, 1977:57). Other changes that affect CPCI performance or external interfaces are also classified as Class I. Class I changes are more closely managed than Class II and must be formally proposed by the contractor and approved by the procuring activity. The use of Class I changes for refining the requirements specification has always been encouraged, but the Navy's life cycle model relies on Class I changes to requirements as the principle means of evolving the system's requirements through the incremental development process. These changes may result in the specification of the next successive increment (and eventually baseline) or may call for the introduction of an increment developed in parallel with other increments (e.g. the development of a version enhanced for a particular application). In either case, the proposed change must pass through an approval cycle and either be filed, if disapproved, or implemented as a new increment, if

approved (see figure 5). This process is the responsibility of the configuration manager.

Class I changes are proposed using two forms: the engineering change proposal, and the specification change notice. The use of these forms is explained in MIL-STD-480 and MIL-STD-490 respectively. A system to automate change processing, based on the use of system stored standard forms, has been developed by General Electric (Zucker, 1983). A configuration management tool similar to the one developed at General Electric would provide a means for integrating the change processing with the system development without resorting to volumes of externally stored documents. This also would allow the use of required information from the change proposal documents during the implementation of approved changes (e.g. automatically tagging modules that require changes and monitoring progress towards completing identified changes).

All changes that do not meet the criteria for Class I changes are considered Class II. Class II changes can be implemented by the contractor without approval from the procurring activity, although the procurring activity must be notified to insure agreement with the change classification. By definition, Class II changes do not propagate through more than one phase of the system life cycle. This characteristic permits simpler management, with a one to one mapping between change proposal and affected

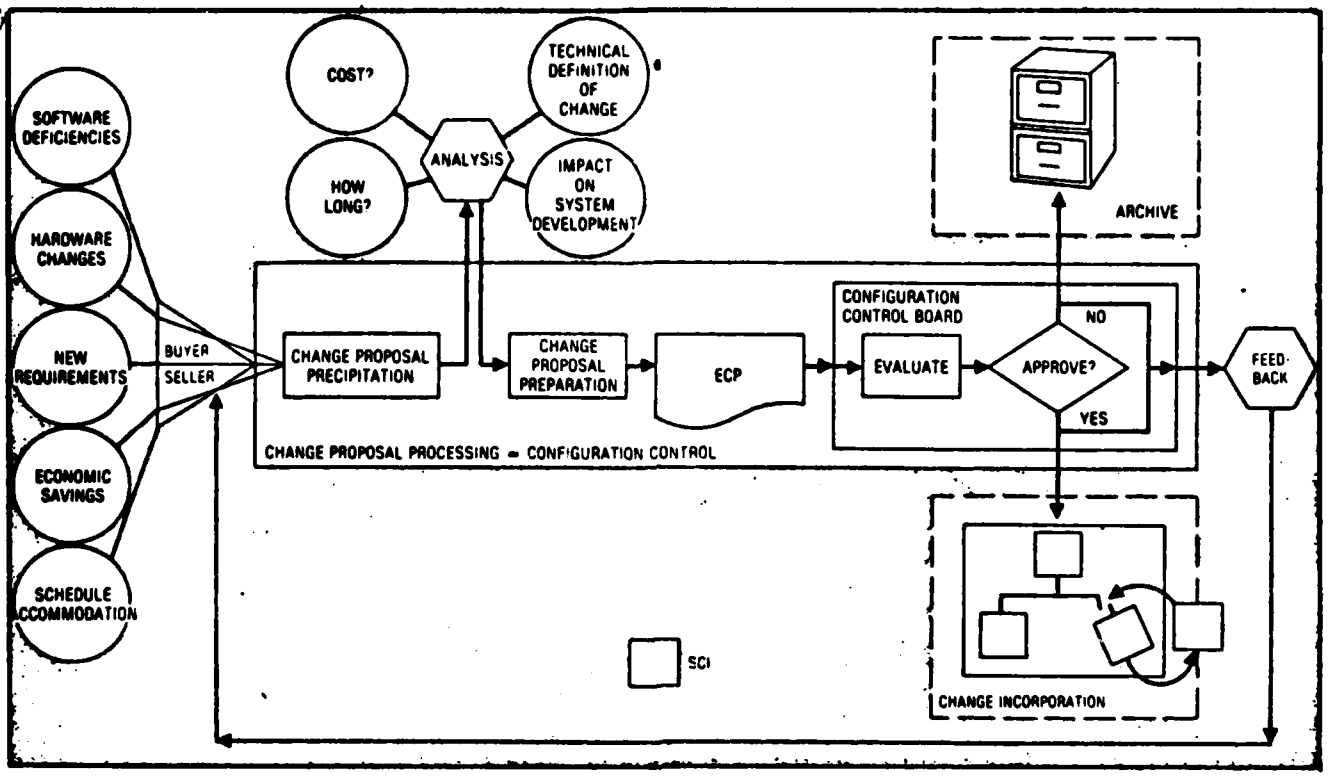


Figure 5. Software Configuration Control Cycle (Bersoff, 1979:8)

document (e.g. a single class II change implemented in a single module of code).

An important category of Class II changes consists of corrections to code between versions. A record of these changes must be maintained for each configuration item. A software engineering environment developed by SofTech (Eanes, 1979) supports the concept of tracked revisions to software components. In this system, each software configuration tree (corresponding roughly to a CPCI) tracks

changes to each component (corresponding to a CPC) with a log file maintained automatically every time a change is made. These log files for each configuration item satisfy the requirement for identifying all Class II changes installed since the last version (Searle, 1977:72).

Configuration Status Accounting. Configuration status accounting is the area of configuration management concerned with insuring that the current state of the software system accurately reflects the system specified in the baseline and requirements documentation (Bersoff, 1979:12). Configuration audits are the means for formally approving design products and establishing baselines. During the life cycle of a software system, five different reviews and audits are conducted to monitor the software system.

First, a System Requirements Review is conducted after completion of the draft system specification. Completion of the System Requirements Review results in establishment of the functional baseline. In this review, software/hardware studies will be reviewed and functions allocated to software. Verifiable performance measures will be specified.

The next review is the System Design Review. Here the draft development specification is reviewed. The allocation of functions to configuration items is checked for completeness and the test plan is reviewed. The allocated baseline is approved during this review.

Once the contractor has completed the preliminary design, the Preliminary Design Review is held. In this review, the procurring activity evaluates the contractor's preliminary design prior to beginning detailed design. All interfaces between CPCI's are checked for consistency and compatability.

After completion of the detailed design, a Critical Design Review is held for each configuration item. The detailed design is checked to insure that the requirements specified in the Part I specification are met. This review is the last check of the design prior to coding and testing.

After coding and testing is completed, a Functional Configuration Audit is held to check the performance of each configuration item against the Part I specification. All test results are reviewed and each approved change is checked to insure proper implementation in the system. The draft Part II specification is reviewed for use in the Physical Configuration Audit. All documentation and manuals are reviewed for completeness.

Either after completion of the functional configuration audit or in conjunction with it, a Physical Configuration Audit is held to examine the "as built" configuration of each configuration item. A product baseline is produced at the completion of this final audit.

This series of reviews and audits represents a well-tested method of monitoring a software development project.

Although based on the traditional life cycle model, these reviews will still be required using the incremental model. With incremental development, a means of maintaining many revisions of the required documents will be necessary. Also, an automated method for monitoring the changes between increments would greatly assist in the processing of reviews after the initial increment is completed. A system similar to the SofTech system described above, that insures that all changes will be listed in the log file would permit a procurring activity to check this log file against the file of approved changes to see that all changes have been incorporated in all appropriate places in the system design.

The methods presented here are designed for initial development and specifically address the communication requirements between the contractor and the procurring activity. Once the system is accepted by the procurring activity, responsibility for incorporating further changes often shifts from the contractor to the government agency responsible for system maintenance. To achieve effective project management, the government agency should continue to use the incremental approach with the same series of internal reviews and audits during the development of each new version of the system. Only with careful control of all changes throughout a system's life cycle will a software maintenance activity be able to insure that changes are properly implemented, tested and documented.

Designer's View of Configuration Management

Ada programming environments are intended to support large software projects. Large software projects associated with embedded real-time applications are generally considered to have many of the following characteristics (Howden, 1982:319):

(1) Three to five year development time

(2) $20 million development budget

(3) 10 year system lifetime

(4) 70 programmers with 5 - 7 managers

(5) developed by external staff and contractors

(6) unsophisticated users

(7) 1 million lines of code

(8) critical reliability

(9) formal reviews conducted to evaluate whole design.

Working on such a project will be a variety of personnel including analysts, programmers, user representatives, industrial engineering personnel, testing personnel, and clerical and operations personnel (Howden 1982:319). Each of these groups of people will be using and creating various products during the system life cycle. From the time development begins to the time the first baseline is established, and between later baselines, these products must be produced, controlled, and coordinated.

The term configuration management is often used to refer to the activities necessary to control these intermediate products of the software development. Although some confusion is likely to result from this extended use of the term, this situation encourages the creation of an integrated configuration management tool used for formal configuration management as well as for control of the intermediate products.

The concept of an Ada programming environment that will eventually be capable of supporting the complete process of program design and evolution, as described in Stoneman (Stoneman, 1980), implies the eventual development of a set of tools supporting every member of the software development team. For the configuration manager, this means that a wide variety of intermediate files will be produced during the software development. These files will be produced by tools in the Ada environment and will often use other files, already stored in the environment, as input. The dependencies resulting from this flow of data provide the necessary information for insuring that all elements used in a configuration are current and consistent.

The task of configuration management of intermediate products can be broken down into the following two steps: First, identify the products that will be produced during the software life cycle, and the methods and tools that will produce these products; and then determine the relationships

and data necessary for insuring the consistency and correctness of these products. These two steps must be considered for each phase of each increment of the incremental life cycle model (requirements analysis, specification, design, and implementation).

Requirements Analysis Products. The requirements analysis phase leads to the construction of the System Performance and Design Requirements. During the requirements analysis process, the computer system specification, produced during system specification is analyzed. This analysis must insure that all software requirements specified in the system design are identified and defined in terms suitable for software specification.

Requirements analysis is normally performed using a semantic model such as Structured Analysis and Design Technique (SADT), or Problem Statement Language (PSL) (Howden, 1982:318). These models portray the requirements, either graphically or through structured text, in a way that produces the structure to be used in system specification. The structure produced by the semantic model must be captured by identifying the key terms, defining the terms, and specifying the relations between these terms (Navy, 1983:I-10). These relations must be maintained by the configuration management tool to allow traceability of requirements.

The configuration management tool must also support the iterative development of the semantic model itself. This includes providing a structured workspace for systems analysis personnel that stores multiple versions of semantic models. An individual systems analyst would be assigned a protected workspace under the manager responsible for his portion of the project. Within this workspace, the analyst would be allowed to edit his design product and periodically update the design version visible to other members of the project development team.

Upon completion of the initial requirements analysis, the configuration management tool will have control over the contractually specified requirements documents, the products of semantic modeling tools (graphic and/or textual) and a database storing the relations created by the structure of the software system. These products will be used in tracing the requirements through the life cycle and in the processing of approved changes.

Specification Products. The specification phase of the software system life cycle leads to production of the Computer Program Development Specification (Part I or Type B5). This document must transform the requirements into a precise description of the system's behavior in sufficient detail to provide the only binding criteria for determining the correctness of the developed system.

The specification process places few demands on the

configuration manager. The products produced (interface requirements, expandability plan, test plan, reliability plan) are generally unstructured text that will be stored as simple text files. These files will be organized based on the structure developed during the requirements analysis phase. A check for completeness will be possible by mapping the requirements already stored in the environment to the specifications. This mapping will produce the necessary relations for tracing requirements to specifications. Specification products will be produced in structured workspaces as described for requirements products.

Design Products. The design process consists of decomposition of the system into a hierarchy of pieces that represent the structure of the software system. The hierarchic decomposition of the system produces pieces that can each be further decomposed, independently, with clearly defined interfaces.

A wide variety of design products and intermediate products are possible depending on the design methodology chosen and the tools available to assist the design process. In all cases, a hierarchy must be produced in a machine understandable form. This hierarchy determines the file structure for the implementation phase. The software requirements must be mapped to the hierarchy to permit traceability.

As in the previous phases of the software life cycle, a structured workspace must be provided with access to the appropriate tools and relations.

Implementation Products. The implementation phase of the software life cycle has the greatest, and most clearly defined, demands on the configuration management tool. The heirarchy produced during system design will be used to structure the workspace of programming teams assigned to the project. Each team will have its own workspace with restricted access to other areas of the system.

As implementation proceeds, a programming team will produce, compile, and test code. The Ada programming language provides the capability for separately compiled program units with restricted visibility of implementation details. The configuration management tool must provide for visible interface specifications and executable implementations. The Ada package facility is designed to support this concept of program development. The package specification provides the interface requirements for the components of the package. Any programming team that will use this package must have read access to the package specification.

The package body provides the executable portion of the package. Only the programming team responsible for development of this package needs read or write access to the package body. Teams with read access to a package

specification will need execute access to the package body.

The interfacing of modules of the software system is critical to the development of a large software project. As modules are developed concurrently by various programming teams, careful control must be maintained to insure that out-of-date modules are not used by members of other teams. A configuration management tool must maintain relationships showing the dependencies of modules used in any desired generation of the system or subsystem. This relation can be automatically constructed from information provided by the compiler and linker/loader, based on WITH clauses in the Ada code. Whenever a modified version of a module is released by a programming team, the modules dependent on the newly modified module must be marked as inconsistent. An automatic re-compile capability can be provided, but must be implemented so as to prevent a new release of a module from interfering with ongoing testing of other modules.

Several versions of a particular module may be produced for various target applications. A single parent module may call these modules in a single call statement. The actual module desired in a particular compilation depends on the target configuration (which is specified to the compiler at compile time). The configuration management tool must provide the information necessary for the compiler to determine which module should be used for this particular compilation.

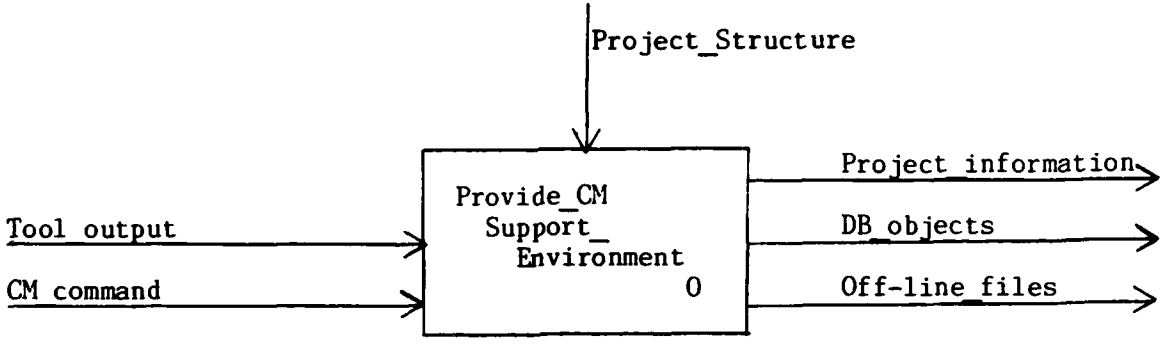
III REQUIREMENTS ANALYSIS

As shown in Chapter II, the task of configuration management involves all members of a software development team during all phases of the project life cycle. In this chapter, the programming environment requirements for support of the configuration management activity will be described. The functions necessary for a MAPSE level configuration management tool will be described. Evaluation criteria for each of these functions will be listed and explained.

At this point, the task of configuration management has been defined, and each subtask has been explained in detail. During this discussion, references were made to the support that these tasks would require from a programming environment. In this section, the requirements developed in the previous chapter will be compiled and described in sufficient detail to allow evaluation of a configuration management tool (CMT) implementation.

The analysis of the requirements associated with the task of configuration management will be functionally decomposed using the Structured Analysis and Design Technique (SADT). SADT activity diagrams will be referenced throughout this discussion to show the functional decomposition of the configuration management task to a level where a detailed requirements analysis is appropriate. The top level SADT diagram (figure 6) shows the

AUTHOR:	CPT Mark Orndorff	DATE:	MAR 84	READER				
PROJECT:	Configuration Management	REV:		DATE				



NODE: A-0	TITLE: Provide_APSE	NUMBER: Figure 6

configuration management task with the input data items and the output products. The processes listed in the SADT diagrams will be described in the text. The sub-headings of this chapter will contain a cross-reference to the appropriate SADT diagram using the numbering convention proposed in (SofTech 1976).

Since the CMT that is to be evaluated was developed to meet the requirements specified in Stoneman, the first step of the requirements analysis will be to establish the level of CM support specified in Stoneman, and apply these quidelines to the CM tasks in Chapter II. These two steps will give a set of requirements for a MAPSE CMT. In Chapter V, some additional requirements that should be addressed in future APSE's will be described.

Every MAPSE implementation is required by Stoneman to have a configuration management tool that supports the task of configuration control (CC). This tool must maintain historic data "sufficient to determine the origin and purpose of each component of the configuration and to control the process of further development and maintenance" (Stoneman, 1980:6.A.12, 2.B.5(8)).

A MAPSE is not required to automate the other configuration management tasks or provide integrated support of configuration management. However, to satisify the requirement in Stoneman that initial environments be "upwards compatible", Stoneman requires the configuration

management tool to use a central database that provides all APSE tools with a uniform and accessible interface to all of the project's information (Stoneman, 1980:2.B.4). The careful use of a central database will permit the addition of APSE level tools that provide the integrated support necessary for effective configuration management of large projects.

To satisfy these guidelines from Stoneman, the CMT must perform two tasks. First, it must provide configuration control for the entire project development. Configuration control is described from a management perspective in Chapter II (pages 21 - 24) and also includes most of the tasks performed by project engineers, also described in Chaper II (pages 27 - 34). The CC task uses the project structure, provided by the project manager, to process the configuration management commands of users and other APSE tools. These commands and the required processing will be described in the next section.

The second task required of the CMT is to maintain all of the project information in a central database, providing project members with access to appropriate APSE tools, and supporting the addition of future tools that will improve the capabilities of the APSE. The project data must be maintained in the database based on the structure established by the configuration manager. All access requests for objects in the project database are first

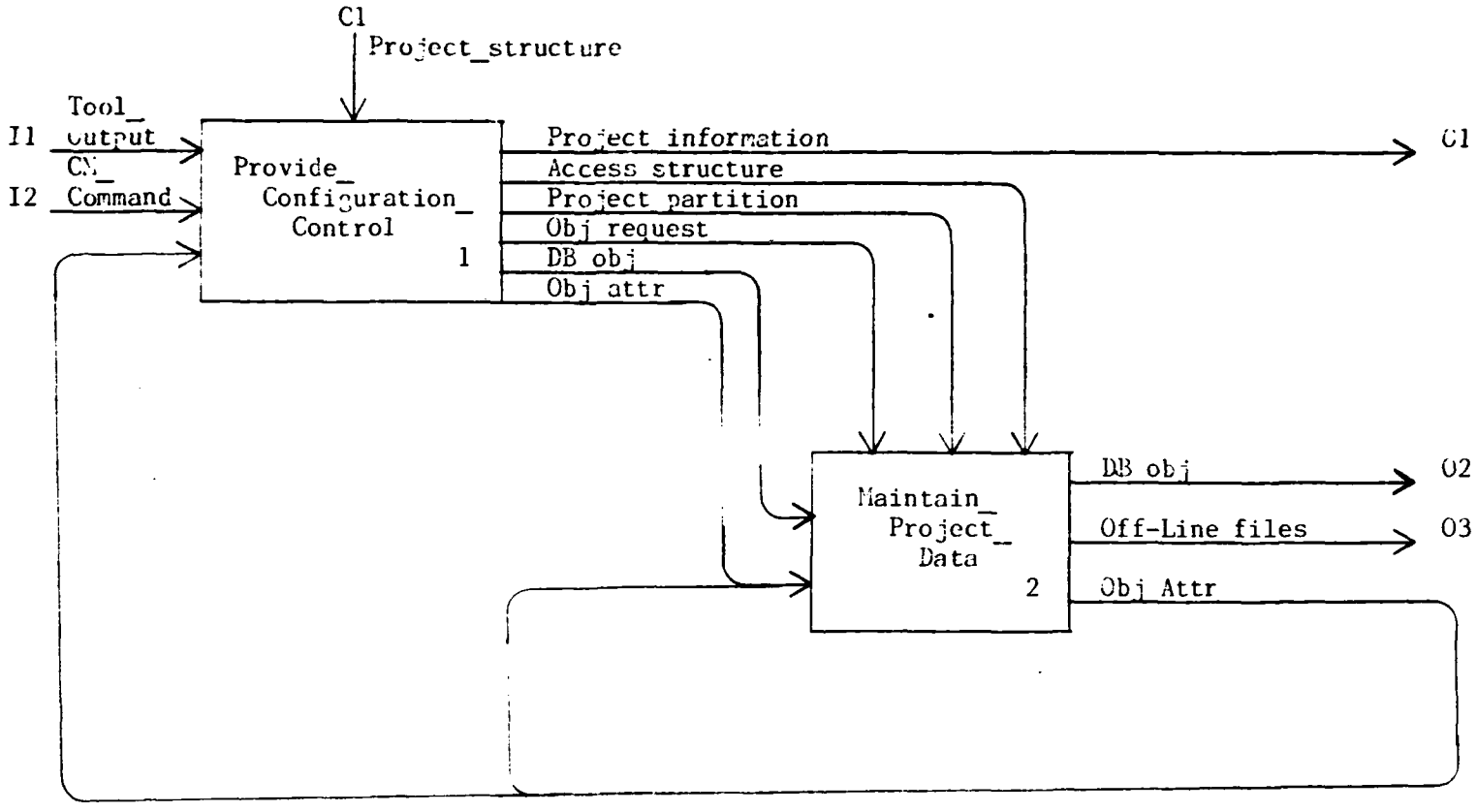
processed by the configuration control tool to insure that the goals of configuration control are met. The relationship between these two tasks is shown in figure 7 SADT A0. The processing required for this task will be described in the section titled "Maintain Project Data."

Configuration Control Requirements (A1)

The configuration control tool (CCT) provides a single interface that is responsible for maintaining the consistency of the project database. As shown in figure 8, the configuration control tool accomplishes this by processing all commands affecting objects contained in the project database. When these commands are received, the configuration control tool performs the processing necessary for maintaining configuration control before requesting objects from the database or creating new objects in the database. With this arrangement, the configuration control tool is constantly in control of the state of the project database and is able to provide current information to all members of the project that have the appropriate access rights.

The configuration control tool must provide a minimum level of processing to support project development. The configuration control tool must support partitioning of the database, control access rights, support multiple versions support multiple targets, provide traceability, and maintain

AUTHOR: CPT Mark Orndorff	DATE: MAR 84	READER				
PROJECT: Configuration Management	REV:	DATE				



NODE: A0	TITLE: Provide_CM_Support_Environment	NUMBER: Figure 7

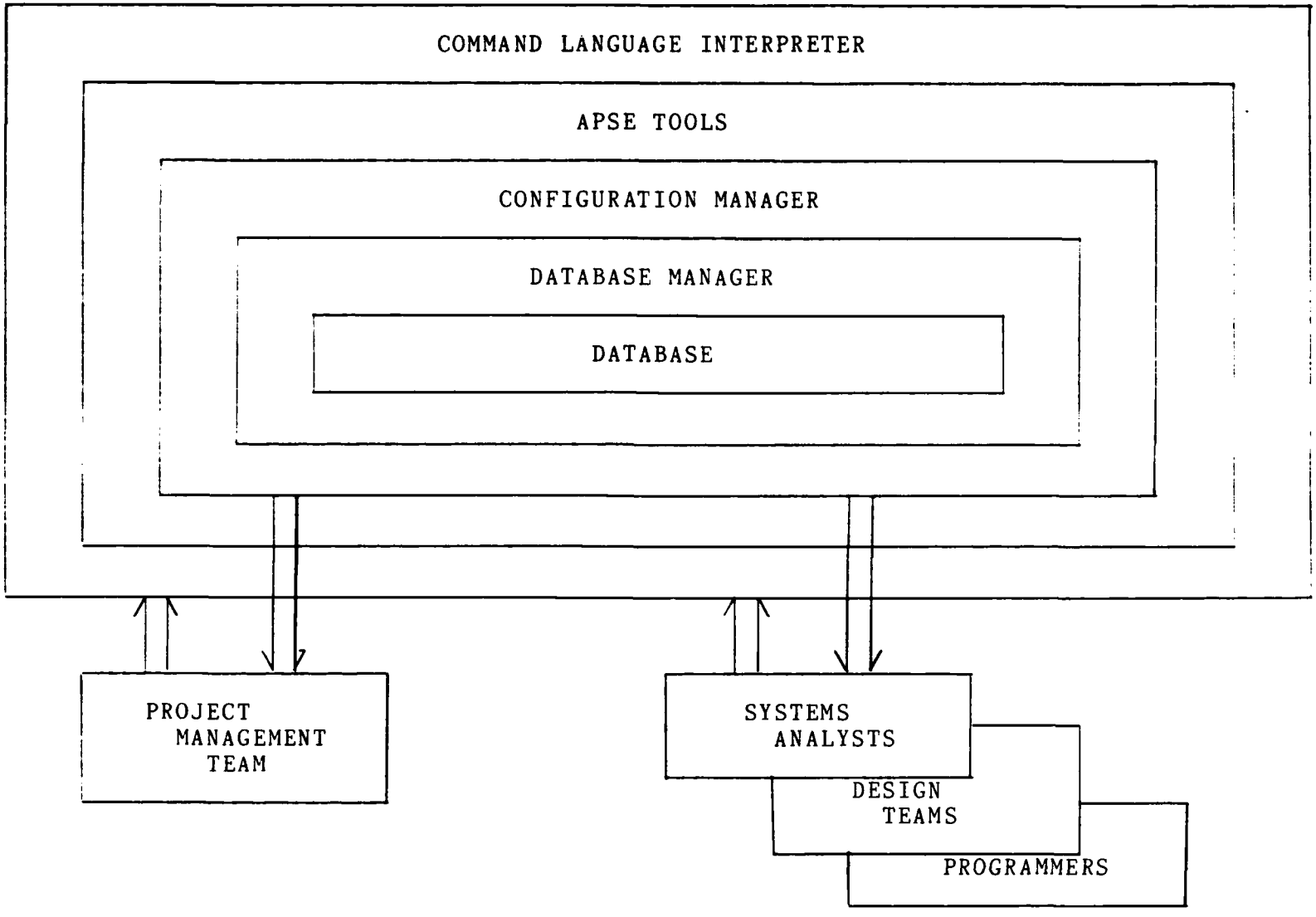


Figure 8. MAPSE Structure

baselines. The relationship between each of these processes is shown in figure 9 SADT A1. This diagram, representing a first-level SADT breakdown of the task Provide_Configuration_Control, shows the exchange of information between these processes and the information required by the configuration control tool from users and management.

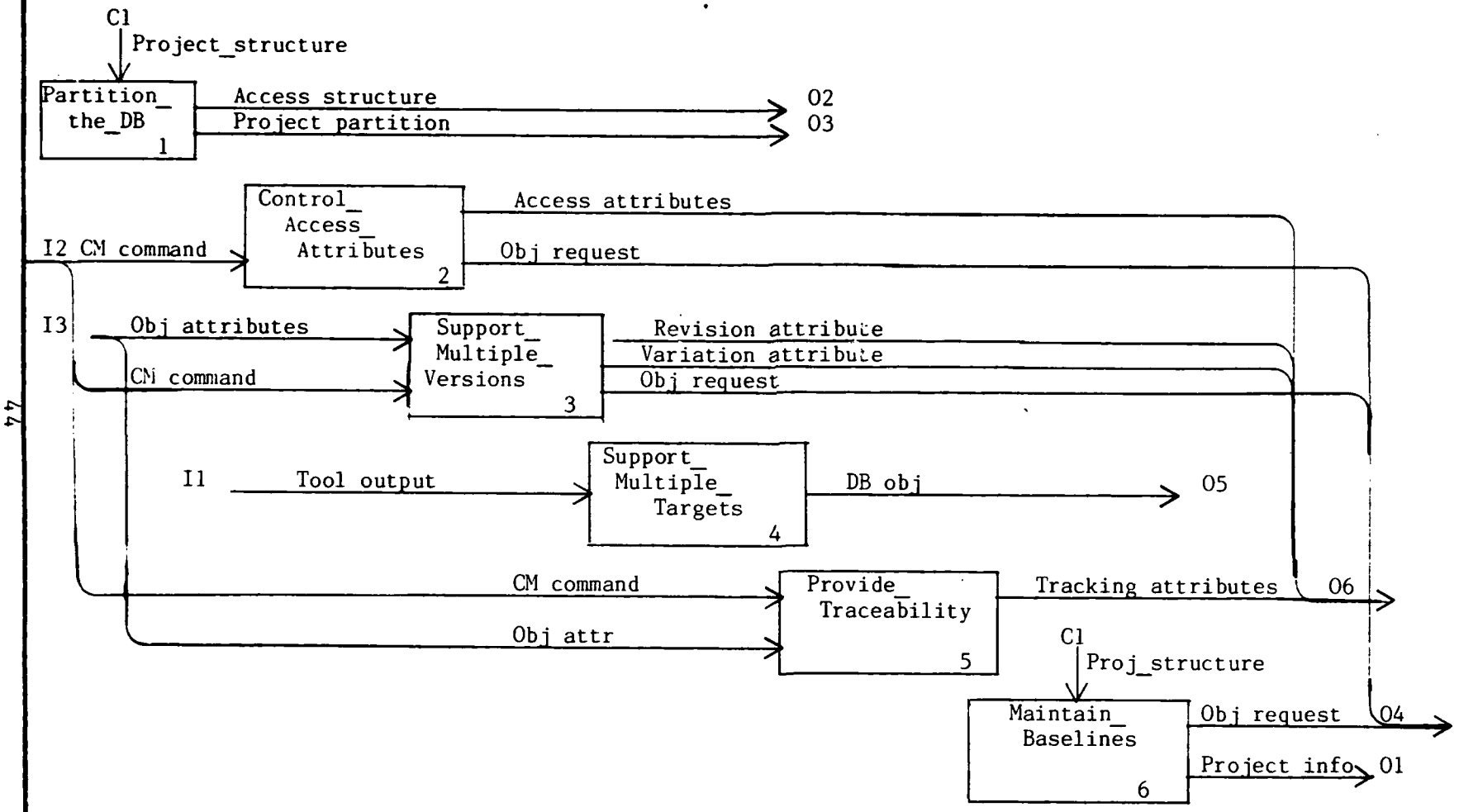
Each of the processes shown on the diagram is described in more detail below.

Partition Project Database (A11). The task of partitioning the project database provides the basis for the rest of the CM tasks. This task can be decomposed into five component parts (see figure 10 SADT A11). The CCT must (a) simultaneously support multiple projects, (b) support multiple teams assigned to each project, (c) provide protected workspaces for each engineer assigned to the project, (d) develop the project partition based on a hierarchical project structure, and (e) provide common libraries for all project members. Each of these tasks will be further described below.

The component requirements of the task Partition Project Database and the criteria for evaluating this task are listed in Table I Partition Requirements.

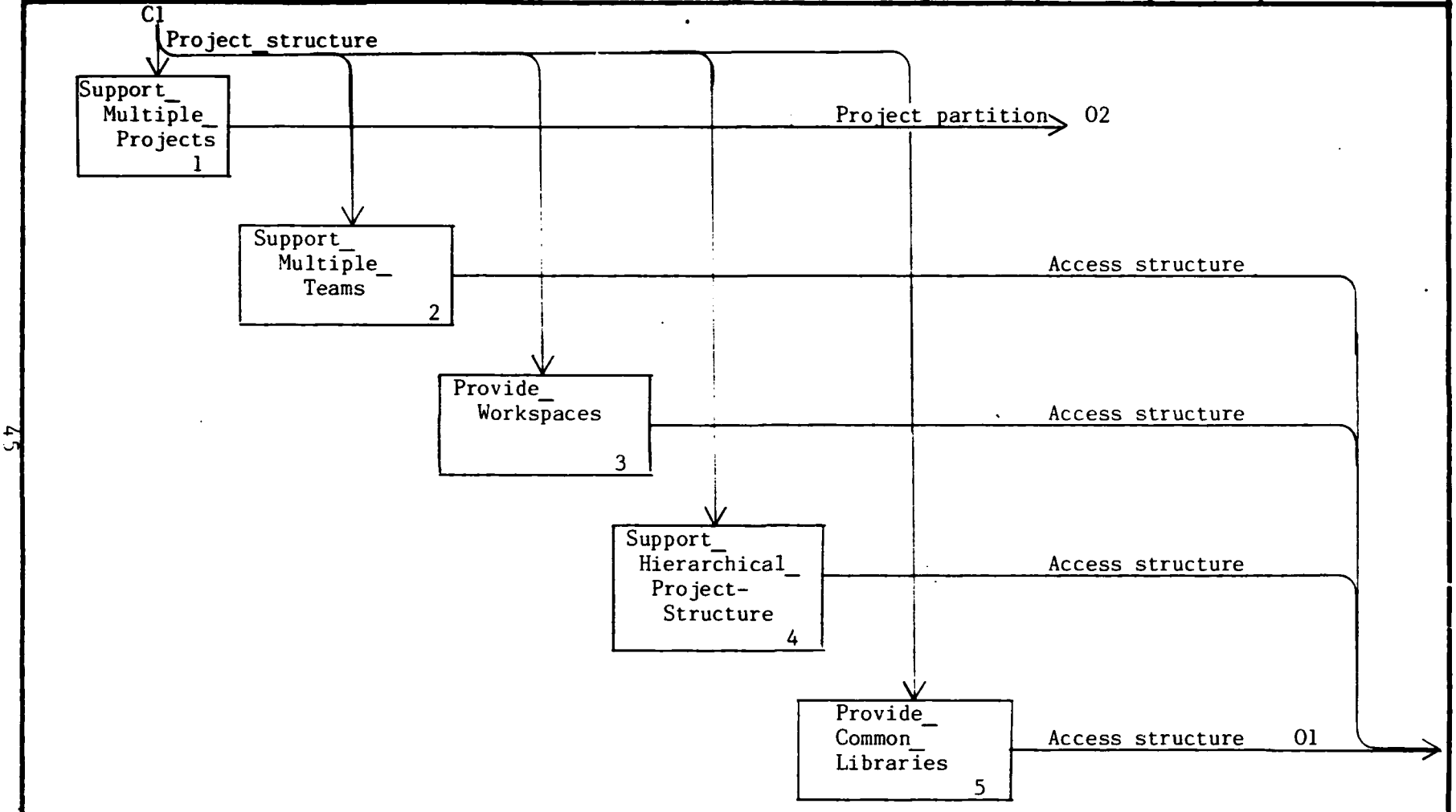
Support Multiple Projects (A111). The APSE database must be capable of supporting several projects

AUTHOR: CPT Mark Orndorff	DATE: MAR 84	READER				
PROJECT: Configuration Management	REV:	DATE				



NODE: A1	TITLE: Provide_Configuration_Control	NUMBER: Figure 9

AUTHOR: CPT Mark Orndorff	DATE: MAR 84	READER				
PROJECT: Configuration Management	REV:	DATE				



NODE: All	TITLE: Partition_the_DB	NUMBER: Figure 10

TABLE I Partition Requirements

Requirements	Evaluation Criteria
1. Support Multiple Projects.	Restrict access to other projects.
2. Support Multiple Teams.	Control access between teams.
	Support multiple levels of teams.
3. Provide Engineer Workspace.	Permit multiple revisions.
	Provide consistent interface to rest of project.
	Inform users of new revisions of shared objects.
[illegible] hierarchical structure.	Allow default revision.
	Allow default variation.
	Provide automatic variation selection.
	Support user views of project structure.
5. Provide Common Libraries.	Maintain consistency of compiled objects in common library.
	Record changes to object in common library.
	Permit single copy of object under revision at a time.
	Provide Functional index.
	Allow sharing of library objects.

simultaneously. Each of these projects will have engineers and managers assigned to it. The CCT must insure that these projects can develop concurrently without interference. The CCT protects projects from interference by isolating the object names of a project from the object names of all other projects, and by isolating users assigned to one project from the actions of other users. The isolated name space is achieved by using the project name as the root directory for each project. Under this root each project can create objects using whatever names desired without confusion with other projects using the same names.

Isolation from other users is achieved by creating access rights based on project affiliation. A user can use these access rights to protect objects developed for one project from being altered (intentionally or otherwise) by personnel working on other projects.

Support Multiple Teams (A112). The project's team structure, provided by project management, is used to partition this project area within the APSE database. Each individual working on the project will be assigned to a certain portion of the project with specific duties. Based on the individual's assignment, he will require access to a certain subset of the project partition.

The CCT supports this structured access requirement by associating project members with teams, with each team having access to a subset of the project. Within these

teams, the types of objects each individual will access will depend on his duties. The CCT must provide a mechanism for restricting access within a team so that individuals will only have access to the types of objects they need. The team structure should permit multiple levels of teams in a single project hierarchy.

Provide Workspaces (A113). The configuration control tool must provide a protected workspace for each individual assigned to the project. Within this workspace, an engineer will develop a software product for release to the other project members requiring access to it. During the development of this product, the engineer will, in general, produce several revisions before a version ready for release to the other project members is developed. An engineer's workspace must maintain these intermediate products, allowing the engineer to recall any previous design.

The engineer must also be provided with a consistent interface to other objects in the project database. The configuration control tool must provide access to the current revision of any object referenced by the engineer, but the release of a new revision must not be forced upon an engineer without his knowledge. This can be accomplished by making the current revision of referenced objects the default version at the beginning of the iterative design process, and informing the engineer whenever an updated

version has been released to the project members. The engineer then must decide when to begin using the new object.

During the development of an object in this protected workspace, the engineer must have the capability for revising the object, storing the revision, restoring previous versions and releasing the object to other project members. The configuration control tool is responsible for maintaining the information necessary for performing these tasks.

Support Hierarchical Project Structure (A114). The structure of a project will be determined by the functional decomposition of the project into portions that will be either further decomposed (independently) or else developed by a single team of engineers. The structure will also be influenced by the requirement for parallel development of a particular functional unit for different applications (i.e. target systems).

The result of these two influences on the project file structure is a hierarchy of objects determined by the functional decomposition of the project. On a single branch of this hierarchy, there may actually be several different variations of an object, each designed for a particular application. Within each of these branches, there may be a series of sequential revisions with one being the result of corrections or modifications of its predecessor. Each of

these revisions is designed to perform the same function for the same application. This arrangement is illustrated in figure 11, Project Hierarchy. This figure illustrates the actual project structure which contains the products of all of the individual engineers.

The configuration control tool must isolate the user from this multi-layered structure by automatically selecting the correct objects for a specified application. Figure 12 User View shows three different views of this hierarchy, all from the same level of the project structure, but for developers of different target systems. In this diagram, the same weapon system is being developed for the Army, Navy and the Air Force. The same central transform is applicable to all three target systems, but each application will have a different afferent and efferent section specially suited to the target system.

The configuration control tool supports the parallel development of specialized branches in this project structure by isolating the user from this structure and automatically selecting the correct module when a configuration is built. In the example above, when a configuration is built for the Army system, the user would only specify, when invoking an APSE tool, that code for the Army system was desired, and the configuration control tool would be responsible for determining what files were used, based on the project structure (choosing the default

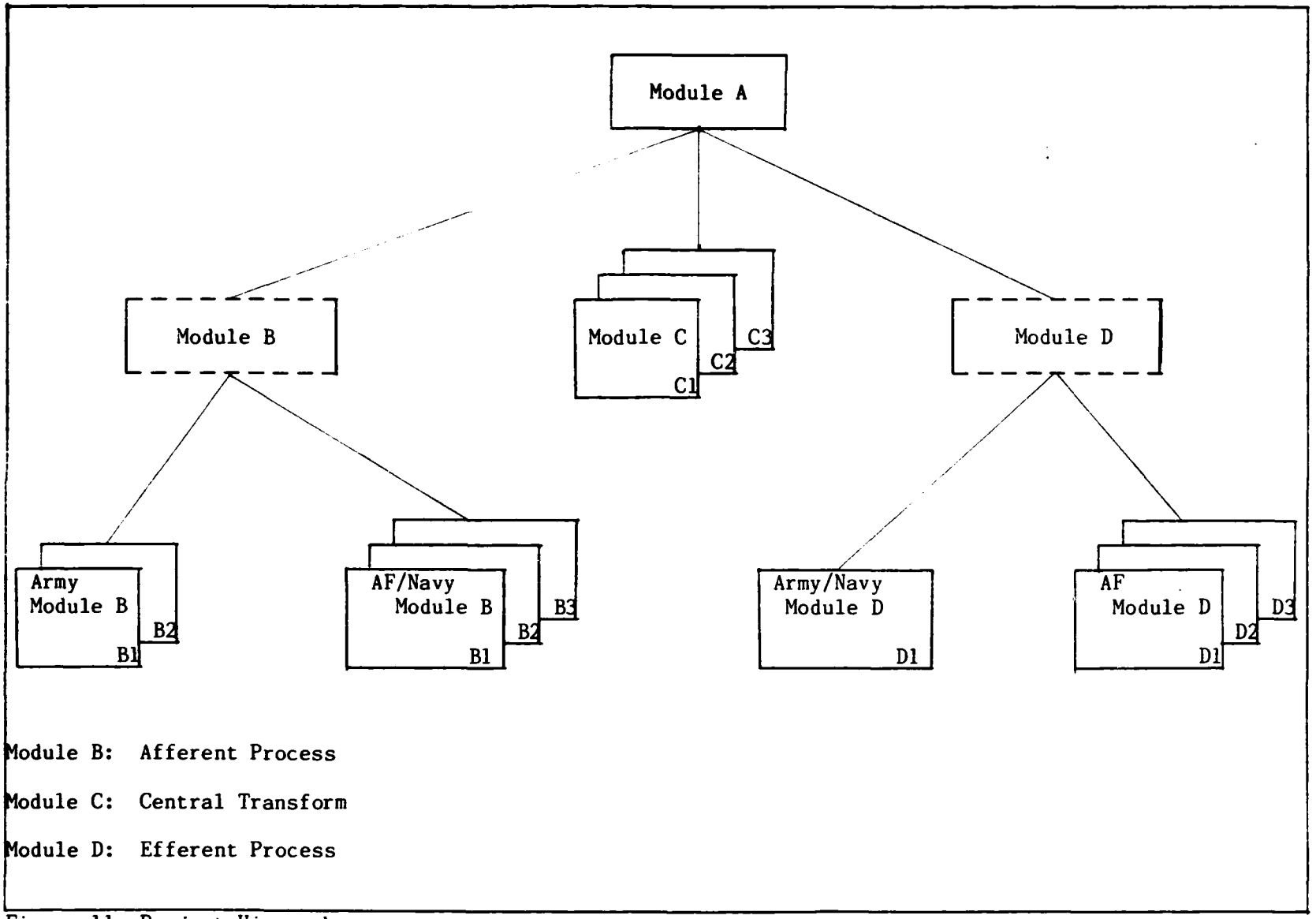


Figure 11 Project Hierarchy

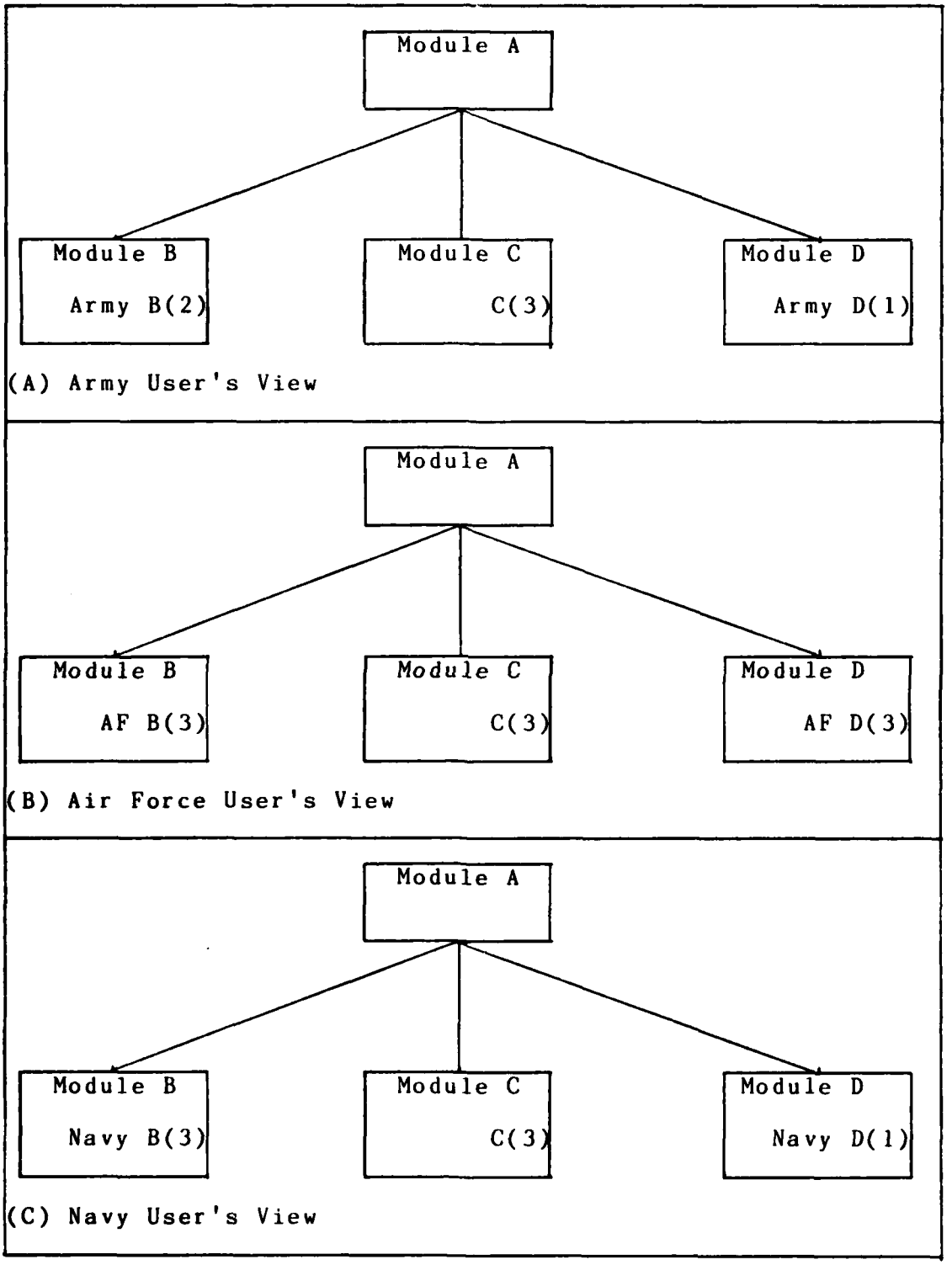


Figure 12 User's Views

revision of the Army variation wherever it existed and using the default revision of other modules as necessary).

Provide Common Libraries (A115). As mentioned earlier, the engineers assigned to a project must have access to common libraries of objects developed by other project members. The CCT must support this requirement by providing a controlled project library that can be shared by all project members. Sharing of database objects must be allowed in three forms. First, an individual data object can be shared. Next, an entire revision set can be shared, with new revisions made available to all sharing users. Finally, an entire subtree can be shared, including all revision sets in the subtree.

Control Access Rights (A12). The engineer who develops an object and the configuration manager responsible for maintaining the objects in the project library must both be able to restrict access to the objects under their control. There are four requirements that the CCT must satisfy to achieve the desired level of access control.

First, the CCT must allow each user to specify a set of default access rights that will apply to every object he creates. These access rights can be specified based on the team structure by listing the teams that will have a given access to an object, or access can be specified by listing individual users that will be allowed access to the objects.

The next requirement that the CCT must satisfy is to permit project members to change the access rights of objects in the project database. The ability to change the access rights of an object should be one of the types of access rights the user controls.

The next requirement which the CCT must satisfy to achieve satisfactory access control is to support several types of access restrictions for both objects in the database, and directories. The type of restrictions implemented on any particular implementation is a design decision, but every implementation must support the simple and efficient addition of new access restrictions.

For example, an APSE implementation may decide to support the specification of read, write, execute, and attribute changing. For many projects this will be adequate, but if a configuration manager requires more control over objects in the project database, he must be able to establish additional restrictions. In this example, he may decide to add controls on deletion, sharing, and revising.

If a single access code in the original system controls several types of access rights, the system must permit the individual specification of these rights as a user option. For example, if in the original system, the write access attribute also controlled deletion, with simple modifications, the system must support separate access

controls for these two operations. This capability must be provided to the extent that operations are available with APSE tools, so that, if desired, a user could specify by name or team who could use each APSE tool on each object in the database. Although this extreme case will not normally be implemented, this capability provides the configuration manager with the flexibility to control a project to whatever level he desires.

Finally, access control should be provided for both objects in the database and directories. This permits a user to protect entire subtrees in his workspace by simply changing the access control to a single directory. Directory access control should not replace any of the access controls provided for individual objects, but rather act as a screen to gain access to the access controls of the objects in the subtree under the directory.

Table II Access Requirements lists the requirements for controlling access rights and the criteria for evaluating this portion of the CCT.

Support Multiple Versions (A13). As shown in the previous section on the hierarchical project structure, multiple versions of objects in a project database will be related to one another in two ways. First, there are objects that are revisions of their predecessor. These objects exist as a result of corrections or modifications with both objects performing the same function for the same

TABLE II Access Requirements

Requirements	Evaluation Criteria
1. Support user default set of access rights.	Allow user specification of default access rights.
	Specify default access rights by team or by individual user.
2. Allow user to modify access rights.	Allow user to change each access attribute.
	Allow creator of object to designate who can change access controls.
3. Allow configuration manager to create new access rights.	Allow configuration manager to add access controls.
	Provide access control for each APSE tool.
4. Provide access control on every object and directory in database.	Provide standard set of access controls.
	Base access control on project, team and user name.

application. The second relationship exists when more than one object is created for a given function, with each object designated for a particular application. These objects are considered variations of each other.

As a result of both of these influences, there will exist in the project database a large number of objects which the user would like to refer to by the same name and

have the system sort out which object he wants. This service is provided by the CCT by supporting both revisions and variations and allowing the user to specify, either by using the user defined default version or by completely identifying an individual object, exactly what objects will be used for a particular operation.

Therefore the task of supporting multiple versions is satisfied by meeting three requirements. The CCT must (1) support revisions, (2) support variations, and (3) allow user-defined default values for both revisions and variations.

The requirements and criteria for evaluating the task of supporting multiple versions are listed in Table III Version Requirements.

Support Multiple Targets (A14). The overall objective of the APSE program is to "offer cost effective support to all functions in a project team ... particularly in the embedded computer system field" (Stoneman 80:2.B.1). The APSE design calls for meeting this objective by adopting the approach of developing software on a host computer for execution on target machines. In many cases, a single host machine will support the development of software for several target machines as well as for the host itself. In this situation, the CCT must organize the executable modules so that a configuration is composed of only modules for a single target system.

TABLE III Version Requirements

Requirements	Evaluation Criteria
1. Support revisions.	Provide automatic incremental revisioning when objects modified. Provide over-write option on user request. Provide listing of current revision of objects in subtree to record state of project.
2. Support variations.	Support multiple levels of variations. Apply single variation specification to all objects in sub-tree.
3. Allow user-defined defaults for both revisions and variations.	Allow default of specific revision by revision number. Allow default of the most recent revision. Allow user defined default of variation.

To satisfactorilly support multiple targets, the CCT must meet three requirements. First, it must provide a mechanism for grouping executable modules by target system. Second, it must insure that when a configuration is linked, or a module dependant on other modules is compiled, all input modules are intended for the same target. Finally, the CCT must provide a means for a single object file to be used for several target systems when appropriate.

These three requirements and their evaluation criteria are shown in Table IV Multiple Target Requirements.

TABLE IV Multiple Target Requirements

Requirements	Evaluation Criteria
1. Group modules by target system.	Record identification of target systems for object files and executables. Group objects in project structure by target system.
2. Insure consistency of compiled and linked objects.	Check target system ID when compiling and linking. Reject commands that mix target systems.
3. Allow single object to be used on multiple targets.	Support multiple target ID's on a single object. Allow multiple target object to appear in multiple groupings of object files and executables.

Provide Traceability (A15). One of the key functions under configuration control is to be able to track requirements, specifications, design, code and tests through the system life cycle. In a MAPSE, there will be little support for automatically tracking this information. An advanced APSE will use some of the techniques from artificial intelligence to better support this task. For early implementations, the support which the CCT must provide consists of providing a means to record the relationship between objects in the database and to retrieve objects based on these relations. These relationships should be immune to changes in the name or directory of

either object. This support would allow the configuration manager to develop the structure for the products of the next phase of the project life cycle by establishing objects that have a recorded relation to the objects from the previous phase.

These relations would provide the necessary information for tracking a change in an object at one phase of the project life cycle by identifying the objects in the next phase that would potentially be impacted by this change.

This area of configuration control is probably the least supported of any of the configuration control tasks, but is also one of the most important. Considerably more research needs to be done before effective tracking of project development becomes a reality.

The requirements and evaluation criteria for this low level of support for the traceability task are listed in Table V Traceability Requirements.

Maintain Baselines (A16). The last task performed by the CCT supports the manager's configuration control requirements. This task involves the support of project baselines. Associated with this task are many of the same requirements listed in the previous four categories. Within a project baseline, there will be revisions, variations, multiple targets, and the access to the baseline must be controlled just as access is controlled to objects in the engineer's workspace. However, there are additional

TABLE V Traceability Requirements

Requirements	Evaluation Criteria
1. Record relationships between objects.	Maintain attribute that contains path name to related objects. Relationship between objects not affected by change in name or directory.
2. Retrieve objects based on relation to other object.	Provide tool to retrieve referenced objects.

requirements that must be satisfied and some of the same requirements must be evaluated in different ways.

In this section, it will be assumed that the configuration manager has available the same support in the above listed areas as do the engineers assigned to the project. The requirements and the evaluation criteria listed will only include those areas that are unique to the task of supporting project baselines.

For project management, the task of configuration control consists of establishing baselines at specific points in the project life cycle and controlling changes to these baselines. To accomplish configuration control, project management needs a structured workspace designed to carefully regulate interaction with the work areas used by the project teams.

The workspace used by project management will contain baseline elements with many of the same attributes as required for project engineers. The primary difference will be the addition of special change control information. Each baseline object will be developed as a result of a requirements document with a set of approved class I changes and a set of class II changes. The control of the class I changes and tracking of the requirements is the responsibility of the CCT.

Each proposed change must be entered into the project database (management workspace). When the approval cycle is complete, the change is either stored in a log of disapproved changes or recorded for use in building a new baseline. This record will contain the components that must be changed to implement the approved change. This record is the basis for the introduction of a new baseline (although, in general, many approved changes will be included for each new baseline).

This record of approved changes will include information on the specifications, designs and modules of code affected by the change. Project management will use this information to assign work to various teams of engineers. When a portion of this change is completed by a project team, the team will release a copy to the project management workspace.

The change record will record which components have been changed and released to the project workspace to monitor progress on implementing the approved changes. When all changed components have been released to the project management workspace, the configuration manager builds a new configuration by selecting the correct combination of changed and unchanged modules. This new configuration is a proposed new baseline.

When a new baseline is established, it becomes a permanent part of the project database. Along with the baseline itself are stored all of the changes implemented in this baseline and a log of class II changes derived from the logs of the development teams.

The task of supporting project baselines consists of three requirements for the CCT. The CCT must maintain multiple fixed reference points while development continues towards the next baseline, the CCT must control changes to these fixed reference points, and the CCT must assist in the processing of proposed changes and the implementation of approved changes.

The requirements and evaluation criteria for evaluating the task Maintain_Baselines are listed in Table VI Baseline Requirements.

TABLE VI Baseline Requirements

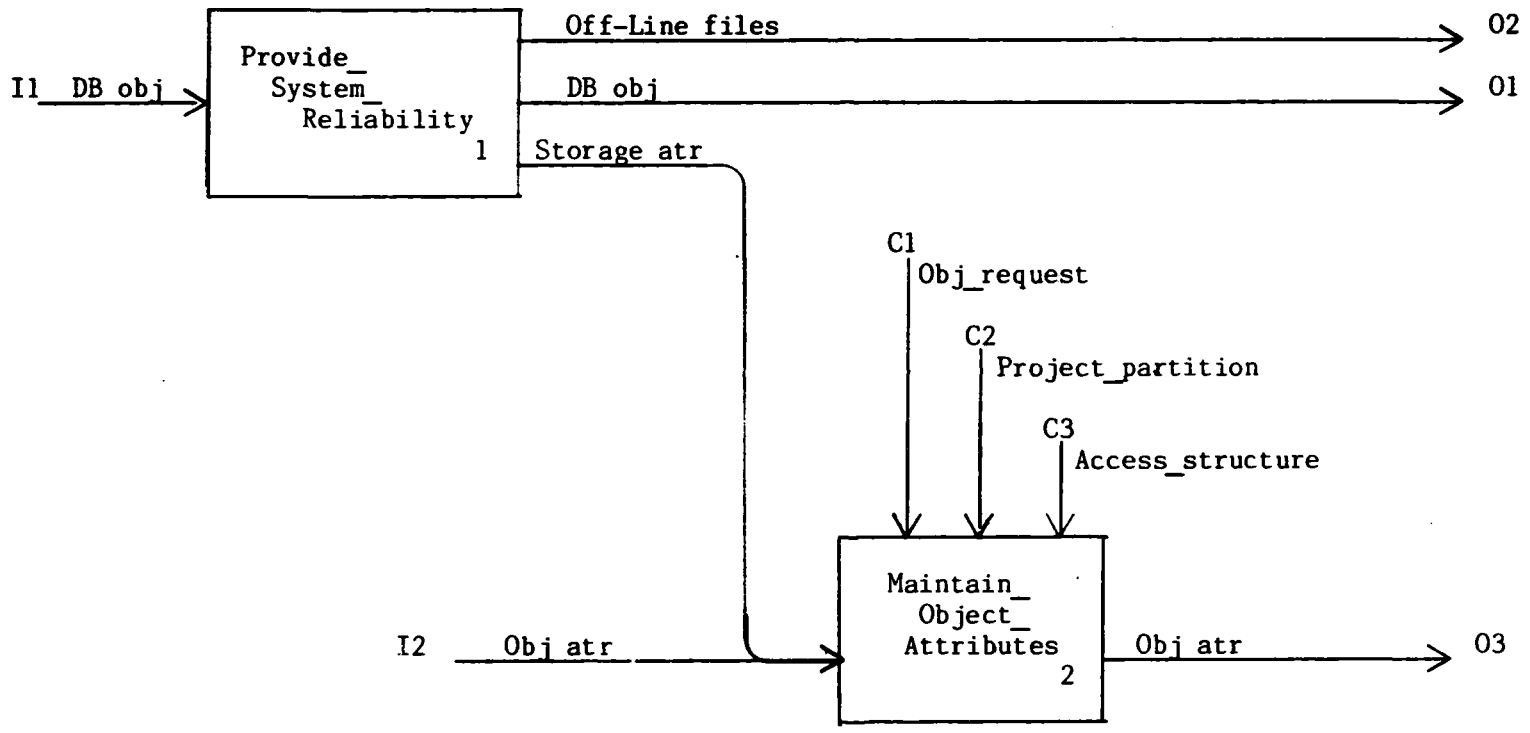
Requirements	Evaluation Criteria
1. Maintain fixed reference point.	Control release from engineer's workspaces. Identify entire baseline by single reference.
2. Control changes to project baseline.	Restrict modification of project baseline. Monitor modification of project baseline. Maintain consistency of compiled units in baseline.
3. Process changes to baselines.	Maintain log of approved changes. Identify objects changing in new baseline. Record approval status of proposed baseline.

Maintain Project Data (A2)

As shown in figure 7 SADT A0, the MAPSE level configuration management task consists of two components. The requirements for the larger and more complex tasks have just been developed. Now, the requirements for the task of maintaining the project data will be developed.

The maintenance of project data will often be performed by several different components of the MAPSE with some functions provided by the DBMS and some by the file manager of the KAPSE. This task will logically be decomposed, as shown in figure 13, SADT A2, into two components: provide system reliability and maintain object attributes. These

AUTHOR: CPT Mark Orndorff	DATE: MAR 84	READER				
PROJECT: Configuration Management	REV:	DATE				



NODE: A2	TITLE: Maintain_Project_Data	NUMBER: Figure 13

two tasks will each be described below and evaluation criteria proposed.

Provide System Reliability (A21). As described in the previous section, the configuration control tool is responsible for selecting the correct version of an object from the project database based on information provided by the user and the requesting tool. This capability is achieved by making the configuration manager responsible for maintaining sufficient information on every object in the database to choose the correct object and locate the object for retrieval by the operating system.

Associated with this task of choosing between several possible versions of an object is the task of maintaining duplicate copies of objects to protect the project from system failures. Since the information necessary for keeping track of redundant copies of objects is similar to that required for maintaining various versions of an object, the task of controlling the backup of project objects falls on the configuration management tool.

The MAPSE backup system uses off-line tape storage to store a copy of objects in the project database. These archived copies are tracked by the configuration management tool so that they can be retrieved when required to reconstruct a lost object. Object backup should be supported in several ways. A total system backup, backup of objects changed since the last system backup, backup of

subtrees or individual objects, and backup of a single baseline.

Users of the system must also be allowed to use off-line storage to store seldomly used objects from the project database. The directory entry for these objects and specific attributes should be maintained on-line with the data portion and the remaining attributes stored off-line, releasing the disc space previously held by the object. One of the attributes maintained on-line should completely identify the tape volume and index required for recovering the object from tape.

This method of object recovery is supplemented by recording a detailed history of designated objects in the system database. This history provides the information necessary to determine the tool used to create the object, the other objects used as input to the tool, the parameters applied to the tool and a script of the commands executed by the tool. This information is all that is required to reconstruct a lost object from the most recently archived predecessor. This capability is protected by maintaining a reference count for every object and preventing the deletion of an object as long as the reference count is greater than zero.

The requirements and evaluation criteria for the task of maintaining reliability are listed in Table VII Reliability Requirements.

TABLE VII Reliability Requirements

Requirements	Evaluation Criteria
1. Maintain off-line backup.	Permit backup of entire database. Permit backup of changes since last backup. Permit backup of subtree of database. Permit backup of baseline. Reconstruct database or baseline from backup. Maintain index of backup tapes.
2. Maintain off-line supplementary storage.	Allow users to store and retrieve infrequently used objects off-line. Maintain, in database, specified attributes of objects stored off-line.
3. Maintain derivation history.	Record input objects and tool used to create object. Support forward and backward tracing of derivations. Prevent deletion of input objects as long as derived object exists.

Maintain Object Attributes (A22). Many of the capabilities of the configuration management tool will depend on the use of attributes of objects stored in the project database. This capability must be made available to users of the system and designers of APSE tools. The capability for maintaining additional attributes provides the means of developing an integrated APSE.

Attributes maintained by the configuration management tool can be divided into four categories (Texas Instruments, 1981:3-3). First, the configuration management tool must be able to automatically maintain information in areas predefined by the environment. For example, the derivation attributes of an object must be known by the environment to permit reconstruction of derived objects. This data should be maintained without operator intervention.

The second category of attribute data includes information required by MAPSE tools, but provided by the user. An example of this type of information is the identification of the target system for a module of code. This information is required by the environment when invoking the compiler, but must be entered by the user. Once entered into the environment, this information should be available for use by all tools in the environment.

The third category of attribute data is defined by the user and maintained automatically by the configuration management tool. This category of data includes information available from various tools in the environment that is not normally required by MAPSE level tools. This information must be stored and updated automatically and be available to the user and all tools in the environment.

An example of this type of attribute would result from the addition of a tool that manages the audit process for a proposed baseline. This tool would require the addition of

attributes that indicate the individuals that must evaluate the objects in the baseline. These attributes would be added automatically by this new tool and later used by the tool to monitor the progress of the evaluation. A possible use of this tool would be to monitor the progress of this thesis. A single list of the readers and sponsors would be stored in a file. Whenever a chapter is released to the readers, an approval attribute for each reader would be associated with the chapter. Also, attributes that point to a comment file for each reader would be maintained. As readers review each chapter, their approval attribute would be changed to reflect their approval or disapproval of the chapter. To check on the current status of the thesis, the tool would check these attributes on each chapter. A report would be generated showing who has approved, disapproved or not evaluated each chapter. The same tool could access the comment files to produce a more detailed report.

It is this category of tools that provides the flexibility required for adding the tools necessary to create an integrated APSE. This capability must be included in all MAPSE implementations, however, no MAPSE level tools will utilize this capability.

The last category of attribute data is defined and maintained by users of the environment. An example of this use of object attributes would be the addition of a new attribute identifying the reason the object was created.

For example, the addition of a 'bug_id' attribute that contains an identifier of a bug that the engineer is attempting to correct in his project workspace. If several modules were modified, the engineer could create a list of the versions that should be used to test the fix by retrieving the file names of the objects whose bug_id attribute matched the bug_id identifier under study. This list would provide the names of the modules to be used to compile a new test version. If the test failed, these objects could all be deleted by a single command deleting all objects with the correct identifier.

Like the previous category, this category must be supported in a MAPSE, but will not be used by the MAPSE tools. This data must also be available for all tools in the environment.

The first two categories of attributes provide the information necessary for MAPSE level configuration control. This information is maintained in the form of attributes that describe and identify objects in the database. In Table VIII Attribute Types, the attributes that must be maintained for effective configuration control are listed and identified as being either category I or category II. Categories III and IV of attribute support must be provided in a MAPSE to support further expansion of the environment, but will not be used by MAPSE tools.

TABLE VIII Attribute Types

Attribute Description	System Provided	User Provided
Tool creating object	X	
User ID		X
Input database objects		X
Date/Time stamp	X	
Variation/Revision code	X	X (1)
Script of revisions	X	
Parameters to tool		X
Purpose of operation		X
Reference count	X	
Object type (e.g. text, code)	X	X (2)
Target system		X

NOTES:
(1) Provided automatically when sequential revision created or manually when new variation created.

(2) Provided automatically by some tools (e.g. compiler, linker) and manually with others (e.g. editor).

To satisfy the requirements for the task of maintaining attributes, the CMT must maintain object attributes that describe the objects and directories, maintain object associations that provide the necessary links to other objects in the project database, support APSE expansion by allowing the addition of new attributes and associations, and allow users to use these attributes and associations as

identifiers for objects stored in the database.

The requirements and evaluation criteria for the task maintain attributes are listed in Table IX Attribute Requirements.

TABLE IX Attribute Requirements

Requirements	Evaluation Criteria
1. Maintain object attributes.	Provide attributes for every object and directory. Support addition of user defined attributes. Maintain attributes automatically when appropriate. Permit attributes to be modified by system users.
2. Maintain object associations.	Provide associations for every object and directory. Support addition of user defined associations. Maintain associations automatically when appropriate. Permit associations to be modified by system users. Insure that associations are not affected by changes in path names of referenced objects.
3. Support APSE expansion.	Provide automatic attribute support for user-added tools. Provide automatic association support for user-added tools.
4. Support retrieval by attribute value.	Permit retrieval of all objects in subtree by attribute value reference. Support use of associations as input to APSE tools.

IV Evaluation of the ALS

The Ada Language System (ALS) is an initial Ada Programming Support Environment designed and developed by SofTech, Inc. under contract with the U.S. Army (contract number DAAB07-82-C-J151). The ALS is designed to be a complete MAPSE satisfying the objectives of Stoneman (SofTech 1983:1-7). The configuration management features of the ALS will be evaluated in this chapter using the evaluation criteria from Chapter III.

Before presenting the evaluation, a brief introduction to the structure of the ALS and the configuration management features will be given to help the reader understand the terms and tools discussed in the evaluation.

Introduction to the ALS

The ALS is a programming environment designed using the four layered model of Stoneman. From a user's perspective, the ALS consists of an ALS command language, a set of software tools and an environment database (SofTech, 1983a:1-1). The user uses the command language to invoke tools that create and manipulate objects in the database.

The ALS command language is a simple programming language that is designed to support interactive use using the syntax of Ada. The command language can be invoked directly by a single command from an interactive terminal, or a series of commands can be placed in a command file that

will invoke each of the commands in succession whenever the command file is called. The command language supports structures of high level programming languages including assignments, loops and conditionals. These features are designed to permit users to create command files that perform tasks not provided by any single tool in the ALS. This allows users to customize their environment to their own particular requirements.

The next component of the ALS is the tool set. The tool set consists of an expandable set of tools that are available to the user through the command language. The ALS tools are designed to support development of Ada programs throughout the entire life cycle. Tools can be added to the toolset either by addition of command files to the tool directory or by writing and compiling new user-created tools. All of the KAPSE functions that were used in the initial tool set are available to users in system libraries, for use when developing new tools. These libraries provide the capability for expanding the APSE as required in Stoneman.

The last component of the ALS is the environment database. SofTech's own description of the database varies considerably from one document to another. In the Users Reference Manual, the database is described as a "comprehensive database under full configuration control" (SofTech, 1983a:1-3), while The ALS Textbook describes this

component of the ALS as "a file structure called the environment database" (SofTech, 1983b:1-7). The ALS database does not support the operations traditionally associated with a relational database system (e.g. cross products, selects, and joins), but does rely heavily on tree walking algorithms traditionally associated with hierarchical database systems. The environment database could be better characterized as a hierarchical file structure similar to UNIX with some added capabilities. However, the term 'database' will be used throughout this thesis to be consistent with the ALS documentation.

The ALS database consists of a hierarchy of directories and files just as in UNIX. A single directory can contain a combination of other directories, files, and any number of the other components of the database. The only other components of the ALS database are variation headers and program libraries.

Variation headers are similar to directories, but instead of indicating a logical decomposition in the project hierarchy, they indicate a grouping of equal alternatives at the same level of decomposition. In the example from chapter III, a variation header would be used to structure the afferent and efferent subtrees of the hierarchy (chapter III, page 43).

The last component of the ALS database is the program library (PL). The ALS documentation treats the PL as a type

of directory, however the operations and components are totally different, so they will be treated here as a separate database component. The PL is a directory used solely for compiled or linked programs. All compiled compilation units and linked object files must be placed in a PL. The PL consists of directories and containers. The directories within a PL are created automatically by the compiler, based on the internal program structure, with a directory for each package and separate procedure. In these directories are 'containers' for each of the objects in that package or procedure. For example, compiling a package called Math_pax and a separate procedure called Newton_Romberg into a PL called MyLib, would produce the structure shown below:

```
d         MyLib
d              Math_Pax
f                   Math_Pax.SPEC(1)
f                   Math_Pax.BODY(1)
d              Newton_Romberg
f                   Newton_Romberg.BODY(1)


d -- directory
f -- file
(n) -- revision number
```

These program library entries are based on the actual names used in the Ada program and have no relation to the file names of the source files. The same PL entries would be made with all compilation units in the same file as with each compilation unit in a separate file. All packages

referenced in a compilation or link must be in the same PL. Each PL supports a single target system.

A program library is designed to support a single program with separate PL's created for each program under development. For large programs, each team or user would have a PL in his workspace for compiling and testing his portion of the project. A single project PL would contain all compiled and linked components after development and separate testing were complete. This project PL would eventually contain all compilation units for the program.

ALS Support of Configuration Control

The ALS is designed to provide "full configuration control." The tasks required for configuration control are supported by a variety of ALS tools and features.

The ALS tools are functionally grouped into sixteen categories, one of which is configuration control. The tools included in this group perform some tasks that are unrelated to configuration management, and some of the configuration management tasks are performed by tools from other groups (especially the file administrator and database manager). For this reason, the tools evaluated in this thesis will include any tool of the ALS that addresses any of the requirements developed in chapter III. Tool names will be listed throughout this section and in the evaluation tables using all capital letters. Descriptions of the

function and format of each of these tools is inculded in the Users Reference Manual (SofTech, 1983a).

In addition to the tools that support configuration management, there are ALS features that support many of the configuration management requirements. Primarily this support is provided by attributes and associations. Attributes and associations are associated with every object in the database. Attributes consist of a name identifier and a character string value. Attributes are used to contain information about the object and are used by APSE tools to control access and restrict operations performed on the object.

Associations are named relations to other objects in the database. An association consists of a name and a list of pathnames. The pathnames can be either relative to the object itself or absolute (from the database root).

In addition to the file structure established in the database using directories and variation headers, the ALS supports a team structure for users of the ALS. Each user allowed access to the ALS must be listed in a system defined file maintained by the system administrator. In this file, the system administrator lists the user's identification (last name) and the team or teams the user is assigned to. The ALS supports a hierarchy of teams using a dot delimiter to separate the levels of the hierarchy (for example, Army_project.module_A.input_process.quality_assurance), just

as it is used in naming the hierarchy of nodes in the database. The team identification and user identification are used by the ALS in creating and enforcing access rights.

The ALS documentation states that a feature called the 'current project directory' (CPD) and a command called 'CHANGE PROJECT' will be added. The documentation does not state how this will work, but it does show that the problem of controlling multiple projects in a single ALS is being addressed. If the CPD simply adds a top project layer to the team_id attribute, and uses this extended team_id for access control, the requirements listed here will be satisfied.

The last feature of the ALS that must be explained before beginning the evaluation is the protected project database. The Project Database is designed to provide the configuration manager with a protected workspace for objects that have completed development and testing and are ready for integration into the proposed baseline. The Project Database will include a PL that will contain a copy of all modules used in the project. The Project Database has special access restrictions that only allow access using special configuration management tools. These special tools are designed to support the requirements associated with maintaining baselines. The configuration manager protects the Project Database by specifying exactly which users will have access to each configuration management tool as well as

the access restrictions on each object in the project database.

The ALS Evaluation

The version of the ALS used in this evaluation is the November 1983 release. This release is an interim product provided to the Army and the Air Force Avionics Laboratory for test and evaluation. It is not a completely implemented ALS and the portion that is implemented has not been completely tested. For this reason, the evaluation was conducted in two phases. First, the documentation was studied to determine how the ALS plans to address the configuration management requirements. Next, the available tools were evaluated to determine if they performed as specified in the documentation and if they satisfied the evaluation criteria. Tools that did not function at all, did not function as stated in the documentation, or were not yet available were all evaluated based on the description in the documentation.

The evaluation method described above and the guidelines from Stoneman (Stoneman, 1980: Chapter 3) led to the following evaluation technique.

First, the method the ALS uses to satisfy the evaluation criteria is described. Next, a series of metrics are used to indicate the status and usefulness of the ALS implementation. These metrics are the implementation status, the criteria success rating, and the simplicity

rating. Each of these metrics will be explained briefly below.

The implementation status simply shows if the function is currently implemented and operational (I); the function is implemented, but has unresolved deficiencies (I-); the function is designed, but not implemented (D); or is not, and is not scheduled to be, implemented (NI).

The criteria success rating is a partially objective rating of the level of success with which the evaluated function satisfies the evaluation criteria. The value of this rating shows if the ALS automatically satisfies the criteria (S), satisfies the criteria using a combination of ALS tools or features (S-), or does not satisfy the criteria (U).

Criteria that are rated (S-) will also receive a rating indicating whether or not a command file can be used to satisfy the criteria. A rating of (S-/CF) indicates that a command file will satisfy the criteria. A rating of (S-/M) indicates that the user must manually invoke each tool or feature every time the function is performed.

In areas where the ALS does not satisfy the criteria, a rating will be added indicating the ALS support for the development of a tool that would satisfy the criteria. A rating of (U/S) indicates a simple combination of library functions would satisfy the criteria. A rating of (U/U) indicates that a new tool would be required. Criteria

receiving this rating either cannot be supported using the ALS technology or else major modifications would be required before the necessary tool could be developed.

Finally, a simplicity rating is given to each function that is implemented and operational. This rating ranges from A to F, with C equating to the simplicity rating that would be given to a UNIX like function. Although this rating is very important in determining whether or not the ALS functions will be used or bipassed, the rating listed here is only one user's evaluation. A more valuable rating would be made if a group of users were asked to rate the function and the results were compiled.

The results of the evaluation are tabulated in the Appendix. The tables are grouped by the functional breakdown from Chapter III, with a single requirement evaluated in each table.

General Discussion

A summary of the information contained in the evaluation tables developed during this evaluation would be impossible due to the large number of topics considered and the quantity of information presented. Instead, in this section general conclusions regarding the suitability of the ALS to the configuration management task will be presented. This discussion will include some of the more significant results from the evaluation tables, providing readers with a broad

overview of the evaluation of the ALS, and giving additional information concerning user acceptance of the ALS. The ALS will be viewed on a much broader scale to give the reader a big picture of the ALS's strengths and deficiencies.

ALS Deficiencies. The greatest and most harmful weakness of the current version of the ALS is its slow response time. The system currently hosting the ALS is not dedicated to the ALS, so precise evaluation of the ALS response time was impossible, however the response on a loaded system to relatively simple commands (e.g. list directory) was orders of magnitude greater than the same command on a heavily loaded system using UNIX (e.g. the AFIT VAX system), or on the same system using VMS instead of the ALS. The slowness of the ALS is more than a nuisance, since users will quickly look for ways to avoid using the ALS as much as possible. The end result will be software developed in an ad hoc manner and delivered on the ALS simply to satisfy DoD requirements.

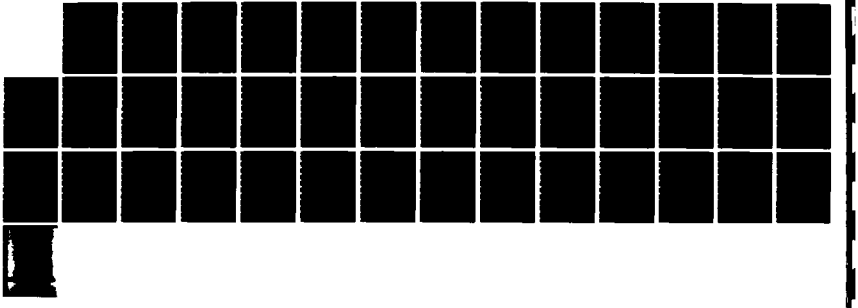
A second, and related, weakness is the command language. This component of the ALS relates to all tools, and was therefore not specifically addressed in the evaluation criteria. However it is worth noting that the command language presents another significant detractor from user acceptance of the ALS. The ALS command language uses an arbitrary combination of complete words and abbreviations, with word pairs sometimes simply concatenated and sometimes

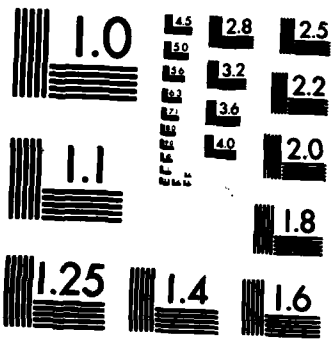
seperated by the underline character. The result is a command language that is difficult to remember and use. The situation would be greatly abated if command substitutors were better supported. Users can create their own list of substitutors that can be used to replace the cumbersome command language, however each use of a substitutor must be prefaced with the symbol '#'. Also, the substitutors are not visible in command files unless they are either initialized within the command file or else declared as global substitutors initially and in every command file where they are used. The Stoneman objective that the environment be suitable for both novice and advanced users is addressed by providing substitutors to permit development of a more comfortable command language, but the current solution should not be considered acceptable.

The next problem area is the support for expansion of the ALS. The ALS does allow the addition of an unlimited number of new tools. These can be added using either command files or by compiling new tools using Ada and the available program libraries. The program library support is excellent and will be listed as a strength of the ALS, but the capabilities provided by command files are greatly restricted. The command language supports the necessary constructs of a programming language (sequence, condition, loop), however there is no way to use the results from one tool as input to another tool in the same command file.

AD-A140 982 EVALUATION OF AUTOMATED CONFIGURATION MANAGEMENT TOOLS IN ADA PROGRAMMING.. (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. M S ORNDORFF 2/2

UNCLASSIFIED MAR 84 AFIT/GCS/EE/84M-1 F/G 5/1 NL





For example, the author attempted to write a simple command file to add a 'share' access attribute to each file whenever the editor was invoked (the ALS requires read access to the parent directory to invoke the share tool, but currently fails to operate correctly unless the sharing user also has write access to the parent directory). The desired initial value was the creator of the file. The first obstacle encountered was the lack of access to the current team_id and user_id from within the ALS. A simple solution to this problem was to list the 'write' access attribute after creating the file and initializing the 'share' attribute with the same value (the 'write' attribute is always initialized to the creating user's id).

The final and unsermountable obstacle to this simple command file was the inability to get the results of the 'LSTATTR' command (the ALS command that retrieves the current value of an attribute such as the write access attribute) into a form that could be used as input to the 'CHATTR' command (the ALS command used to create new attributes). The output of ALS tools is designed for screen display, and no method of separating the labels from the data and using the data in a later command is provided.

The last negative observation concerns the support for user-added attributes and associations. The ALS goes to considerable trouble maintaining attributes and associations for every object in the database. The usefulness of these

features is greatly reduced by the lack of support for the common database operations. The only use of attributes that approximates a 'select' is in the use of variation headers. When selecting objects under a variation header, the user can specify attribute names and values instead of using the variation name. This allows the user to create variations based on multiple characteristics, with a separate attribute created for each of the distinguishing characteristics. This feature provides excellent support for variations and should be expanded to permit similar select operations in conjunction with other ALS tools, so that attributes could be used to organize a project instead of merely providing a labelled comment field for users.

ALS Strengths. The ALS has many features that work well towards the goal of effective configuration management. The Project Database, the support of multiple variations and revisions, the off-line storage system, the library support for additional tools, and the support for derivations all represent significant advances in the 'state-of-the-art' in programming environments. The support for variations and the library support for additional tools were discussed in the previous section. The remaining features will be discussed briefly below.

The ALS design for a Project Database is probably the most significant advance in the area of configuration management. The Project Database brings the project

development under the control of the configuration manager, with a specially controlled set of tools designed for the task of configuration management. This arrangement not only provides good initial support for configuration management (especially baseline management), but also provides an excellent basis for the addition of advanced configuration management tools as they are developed. The separation of the database into two distinct areas with one for project development and one for controlled configuration management allows designers of new tools to address the needs of managers separately from the needs of engineers, and therefore permits the development of specialized configuration management tools.

The ALS's support for off-line storage provides several tools for the various requirements of system users and administrators. The ALS uses three sets of tools to support the three different functions of off-line storage: backup, supplementary storage, and baseline storage or transfer. These three tool sets are specialized for a particular task, and therefore provide automated support for a complete task rather than requiring a combination of several more general purpose tools. The supplementary storage tools are especially convenient, providing users with an excellant facility to reduce on-line storage. A single fault in the current design requires the person performing the actual tape operation (usually the system administrator, using

ROLLOUT) to have attribute change access to the objects being transferred. This requirement needlessly destroys the access control features of the ALS by giving someone without any valid requirement complete access to these objects. A change to check the attribute change access of the user requesting the transfer to tape (using ARCHIVE) would solve this problem without further complication.

The ALS supports derivations by automatically recording the tools used, the input files, the command format, and the derivation count of the input files whenever a derived object is created. Derived objects are created as output from a certain class of tools called 'generators' (e.g. compiler, linker). Users can create new generators by developing a command file and then using the 'MAKEGEN' tool.

One feature associated with derivations that is particularly significant in that it would be useful elsewhere in the ALS is the recording of the VMS filenames of input files. Input files are recorded in derivations using both the ALS pathname and the permanent filename associated with the file under VMS. The use of the permanent filename allows the ALS to always be able to locate the original file and decrement the derivation count when the derived object is deleted. This prevents objects with derivation counts greater than zero from being locked in the database whenever their ALS pathname is changed. This support for maintaining derivation counts is excellent,

and should be expanded to allow users to maintain associations with objects in spite of name changes or changes in the project structure. The ALS has the capability to track these changes, but it is not available to the users.

In conclusion, this evaluation has shown the strengths and weaknesses of the ALS's support for configuration management and tabulated the ALS's effectiveness in addressing each of the configuration management requirements. The results presented here must now be reviewed to determine the trade-offs that must be made so that a satisfactory implementation can be produced given the prevailing time and budget constraints.

V Summary and Recommendations

Summary

In this thesis, the task of evaluating automated configuration management tools was addressed in three phases. First, the discipline of configuration management was defined from the perspectives of project management and project engineers. This definition was based on current DoD policies and software engineering practices.

The next step was to use this definition of configuration management to develop evaluation criteria for configuration management tools. The evaluation criteria were based on a requirements analysis of the configuration management task. Evaluation criteria were developed only for those areas of configuration management designated in Stoneman, the DoD's requirements document, as requiring support in APSE implementations. This limitation was imposed to make the evaluation criteria appropriate for the evaluation of initial APSE implementations designed to meet the 1980 Stoneman requirements.

In order to develop the evaluation criteria, the task of configuration management was functionally decomposed into a hierarchy of component tasks using the Structured Analysis and Design Technique (SADT). Each component task was then stated in the form of a requirement for Ada environments, and evaluation criteria were developed. These evaluation criteria were tabulated and presented in Chapter III.

The final phase of this thesis consisted of applying the proposed evaluation criteria to the Ada Language System. The Ada Language System was evaluated using an interim version of the software and the design documentation. A set of metrics were used to measure the ALS's effectiveness in satisfying each of the evaluation criteria. The implementation status of the evaluated function, the success of the ALS in satisfying the evaluation criteria and the simplicity of the ALS were all addressed using these metrics.

Considerable emphasis was given to evaluating the open-endedness of the ALS. In each area where the ALS received less than the highest success rating, an indication was added to show the ease with which users of the ALS could develop additional tools that would satisfy the evaluation criteria. This rating is considered most important in determining the robustness of the ALS, as it shows the support inherent in the ALS design for improvements and the addition of advanced functions.

The processes just described produced three products. First, a detailed definition of the configuration management task suitable for requirements analysis was developed. This definition can be further analyzed as the role of the configuration management tool is expanded and additional needs are addressed.

Second, a set of evaluation criteria, appropriate to the evaluation of Ada environments designed to meet the 1980 Stoneman requirements was developed. These evaluation criteria are implementation independent and will be useful in comparing various environments as additional APSE's are developed.

The third product was the actual evaluation of the ALS. This evaluation can be used by the Army in determining the acceptability of the proposed design as well as by system designers in determining areas for improvement in future releases of the ALS.

The thesis presented here will assist the designers and developers of software engineering environments by providing a set of evaluation criteria that will give an accurate indication of the potential for success of a proposed programming environment. The best test of a programming environment measures the acceptance and use of the environment by software developers, and the improvements in productivity, reliability, and maintainability of the software produced using the environment. Unfortunately, this test cannot be made until after a complete release of the system is ready and a substantial investment in fielding the environment has been made. Hopefully, the evaluation criteria presented here will give an accurate indication, early in the development process, of the eventual success of a proposed design.

This thesis did not attempt to award an overall acceptability rating for the ALS. Rather, a comprehensive evaluation of individual requirements was developed, accompanied by some general observations concerning the design and performance of the ALS. The intention was to provide the Army with enough information to evaluate the current status of the ALS, make knowledgeable trade-offs, and initiate changes that will improve the ALS prior to its acceptance and general use.

Recommendations

The recommendations that result from this thesis fall into two categories. First, recommendations for the developers and procurrers of the ALS are made. Second, there are recommendations for designers and researchers that will be working on future environments that will provide advanced functions over the current generation of programming environments.

The next step that must be taken by those involved in the development and fielding of the ALS is to use this list of deficiencies for determination of the changes and trade-offs that will be made in future releases of the ALS.

The recommended approach is to first give prospective users the evaluation criteria presented here and have them rank them in order of importance. The next step would be to use this information to determine the changes that must be

made before release of the ALS and changes that can be deferred to later releases of the system. The last step would be to have representatives from prospective using organizations apply the evaluation criteria to a later release that has incorporated the approved changes to determine the potential for acceptance of the ALS when released to software developers.

The areas for future research and development in the current generation of configuration management tools include the configuration management tasks not addressed in Stoneman (configuration identification and configuration status accounting) as well as advances in the areas addressed here and in Stoneman.

The next generation of software engineering environments will include the support of a particular methodology. The first step that must be taken is to determine an acceptable methodology for software development and then develop future environments to support and enforce this methodology.

The current generation of environments have attempted to address the tasks that are performed during the software life cycle, and develop tools that automate these tasks. The result is a collection of tools that work together to simplify the activities of software development, but do not force the use of good software engineering principles.

The role of future environments must expand to include the enforcement of a methodology that will improve the

software development process. The Navy's Software Engineering Environment Work Group has taken the first step in this direction. Their work provides the basis for the development of the next generation of configuration management tools. The development of a software engineering environment that not only simplifies the task of software development, but also promotes the use of good software engineering practices must be considered the immediate goal of the software development community.

The emphasis here has been on the task of configuration management and the tools that would support it. However, satisfying the evaluation criteria developed here does not represent successful automation of the task of configuration management. This study merely measures the affectiveness of achieving a currently obtainable level of configuration management support without the delay that would be necessary for the development of a satisfactory methodology or the development of advanced tools.

This is not a stopping point or even a major advance into a new area, but merely represents the collection of available technology in an effective manner to create a working system that can be expanded once a methodology has been adopted and technological advances become available. The need for more advanced systems that add order to the descipline of software engineering is great, and the ALS represents a first step towards acheiving this goal.

The research presented here provides a means for evaluating currently obtainable support for configuration management, as well as the theoretical background necessary for additional research into more advanced support for configuration management. This information coupled with the development of an acceptable methodology, forms the basis for the next step in the automation of the software engineering discipline.

Appendix: Evaluation Tables

Key to Evaluation Codes

Implementation Status:

I -- Implemented and Operational

I- -- Implemented with unresolved deficiencies

D -- Designed but not yet implemented

NI -- Not implemented or designed

Success Rating:

S -- Satisfies the criteria

S- -- Satisfies the criteria using combination of tools

- S-/CF -- Command file will satisfy the criteria
- S-/M -- Tool combination must be manually invoked

U -- Does not satisfy the criteria

- U/S -- Simple combination of library functions would satisfy the criteria
- U/U -- New tool required to satisfy the criteria

Simplicity Rating:

A -- Simplest

C -- Equivalent to UNIX like function

F -- Most complex

TABLE X (A) Partition Evaluation

Requirement: Support Multiple Projects.

Evaluation Criteria	Implementation	Implement	Success	Simplic
Restrict access to other projects.	Current Project Directory will be used to indicate project partitions.	D	S	-

TABLE X (B) Partition Evaluation

Requirement: Support Multiple Teams.

Evaluation Criteria	Implementation	Implement	Success	Simplic
Control access between teams.	Team_id is used as a component part of access attributes.	I	S	B
Support multiple levels of teams.	Team_id consists of a hierarchy of team levels.	I	S	C

TABLE X (C) Partition Evaluation

Requirement: Provide Engineer Workspace.

Evaluation Criteria	Implementation	Implement	Success	Simplic
Permit multiple revisions.	Objects with same name may exist with sequential revision numbers.	I	S	A
Provide consistent interface to rest of project.	SHARE tool allows sharing of revision sets or subtrees from other project areas.	I-	S-/M	-
	ACQUIRE tool allows sharing of containers from other PL's.	I	S-/M	D
Inform users of new revisions of shared objects.	Not supported. New revision of shared object is automatically made available. New revision of acquired object must be acquired -- user not informed when new revision is available.	NI	U/U	-

TABLE X (D) Partition Evaluation

Requirement: Support Hierarchical Project Structure.

Evaluation Criteria	Implementation	Implement	Success	Simplic
Allow default revision.	Most recent revision or most recent frozen revision are only available defaults.	I	S-/CF	B
Allow default variation.	Default_variation attribute selects variation to be used when specific variation is not chosen.	I	S	B
Provide automatic variation selection.	Not supported -- subtree wide default can be selected.	NI	U/S	-
Support user views of project structure.	Not supported -- explicit variation selection forced on users.	NI	U/U	-

TABLE X (E) Partition Evaluation

Requirement: Provide Common Libraries.

Evaluation Criteria	Implementation	Implement	Success	Simplic
Maintain consistency of compiled units in common library.	ACQUIRE tool allows sharing of containers from common PL.	I	S-/M	D
	IMPACT tool tells user which objects will need re-compiling if new object added to PL.	D	S-/M	-
	Need_recompile attribute set when container not consistent with rest of PL.	I	S	B
	Linker will reject attempts to link if containers need to be re-compiled.	I	S	A
Record changes to objects in common libraries.	LST_INSTALLS produces listing of user name and date of all changes made to the library.	D	S-/M	-
Permit single copy of object under revision at a time.	GET tool checks out copy of object for revision and prevents another use of GET until UNGET or INSTALL.	D	S	-
Provide functional index.	Not supported.	NI	U/S	-
Allow sharing of library objects.	Acquire tool allows sharing of containers from common library.	I	S-/CF	D

TABLE XI (A) Access Evaluation

Requirement: Support User Default Set of Access Rights.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Allow user specification of default.	Not supported -- System specifies an unchangeable set of defaults.	NI	U/S	-
Specify default access rights by team or by individual user.	Wild card permits access attribute to include entire teams or specific users.	I	S	C

TABLE XI (B) Access Evaluation

Requirement: Allow User to Modify Access Rights.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Allow user to change each access attribute.	CHATTR command allows user to change any access attribute.	I	S	D
Allow creator of object to designate who can change access controls.	attr_change attribute determines who can change access attributes.	I	S	D

TABLE XI (C) Access Evaluation

Requirement: Allow Configuration Manager to Create New Access Rights.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Allow configuration manager to add access controls.	Requires major modification to all APSE tools affected by the change.	NI	U/U	-
Provide access control for each APSE tool.	Requires re-writing all APSE tools to enforce change.	NI	U/U	-

TABLE XI (D) Access Evaluation

Requirement: Provide Access Control for Every Object and Directory.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Provide standard set of access controls.	System automatically establishes access controls in system defined categories.	I	S	A
Base access control on project, team, and user name.	Access attribute syntax includes combination of team_id, user name and wild-card.	I	S	C

TABLE XII (A) Version Evaluation

Requirement: Support Revisions.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Provide automatic incremental revisioning when objects modified.	FREEZE command forces automatic new revision upon object modification.	I	S	A
Provide over-write option on user request.	Non-frozen objects overwritten - impossible to un-freeze object once frozen.	I	S	B
Provide listing of current revision of objects in subtree to record state of project.	INVENTORY tool will create a file listing current revisions -- not yet implemented.	D	S	-

TABLE XII (B) Version Evaluation

Requirement: Support Variations.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Support multiple levels of variations.	Variation headers permitted at any level in project hierarchy.	I	S	A
Apply single variation specification to all objects in sub-tree.	Not supported -- a single default variation can be specified for each object.	NI	U/U	-

TABLE XII (C) Version Evaluation

Requirement: Allow User-defined Defaults for Revisions and Vaiations.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Allow default of specific revision by revision number.	Only most recent revision allowed as default.	NI	U/U	-
Allow default of the most recent revision.	Automatically defaults to most recent revision.	I	S	A
Allow user defined default of variation.	Default variation allowed in every variation header.	I	S	B

TABLE XIII (A) Multiple Targets Evaluation

Requirement: Group Modules by Target System.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Record identification of target systems for object files and executables.	Compatible_targets attribute records all target systems listed in PRAGMA SYSTEM of source program.	I	S	A
Group objects in project structure by target system.	A program library only contains objects for a single target system.	I	S	A

TABLE XIII (B) Multiple Targets Evaluation

Requirement: Insure Consistency of Compiled and Linked Objects.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Check target system ID when compiling and linking.	Target specified in SYSTEM pragma must match target_system attribute of program library.	I	S	A
Reject commands that mix target systems.	All objects in a PL are for same target system and all objects used for compile or link must be in same PL.	I	S	A

TABLE XIII (C) Multiple Targets Evaluation

Requirement: Allow Single Object to be Used on Multiple Targets.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Support multiple target ID's on a single object.	SYSTEM pragma allows user to specify all targets for a compilation -- source file must be modified to add an additional target.	I	S	D
Allow multiple target object to appear in multiple groups.	ACQUIRE tool allows sharing of objects between PL's if both target systems are listed in compatible_targets attribute.	I	S	C

TABLE XIV (A) Traceability Evaluation

Requirement: Record Relationships Between Objects.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Maintain attribute that contains path name to related objects.	Associations used to relate objects and directories.	I	S	D
Relationship between objects not affected by change in name or directory.	Not supported -- associations only record pathnames as given by user. Change in pathname will not be tracked.	NI	U/U	-

TABLE XIV (B) Traceability Evaluation

Requirement: Retrieve Objects Based on Relation to Other Objects.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Provide tool to retrieve referenced objects.	Not supported -- pathnames found in association must be manipulated manually by user.	NI	U/U	-

TABLE XV (A) Baseline Evaluation

Requirement: Maintain Fixed Reference Point.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Control release from engineer's workspaces.	Configuration manager must allocate space for objects using NEW and engineer must use INSTALL to add object to project database.	D	S	-
Identify entire baseline by single reference.	INVENTORY tool creates file listing current revision of all objects in project DB.	D	S	-
	SNAPSHOT tool creates file listing current revision of all containers in PL.	D	S	-

TABLE XV (B) Baseline Evaluation

Requirement: Control Changes to Project Baseline.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Restrict modification of project baseline.	CMLOCK prevents anyone from changing object in project database.	D	S	-
Monitor modification of project baseline.	INSTALL records name and date of user returning modified object to project DB -- reason for change is not recorded.	D	U/S	-
	LST_INSTALLS produces a listing of all changes that have been made to an object in the project DB.	D	U/S	-
Maintain consistency of compiled units in baseline.	IMPACT cool tells user what objects will have to be re-compiled if object added to PL.	D	S	-
	Need_recompile attribute set when object not consistent with rest of PL.	I	S	F

TABLE XV (C) Baseline Evaluation

Requirement: Process Changes to Project Baselines.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Maintain log of approved changes.	Not supported.	NI	U/S	-
Identify changing objects for new baseline.	Not supported.	NI	U/S	-
Record approval status of proposed baseline.	Not supported.	NI	U/S	-

TABLE XVI (A) Reliability Evaluation

Requirement: Maintain Off-line Backup.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Permit backup of entire database.	BACKUP copies project database onto tape.	I-	S	-
Permit backup of changes since last backup.	BKPCHNG copies nodes changed since last BACKUP or BKPCHNG.	I-	S	-
Permit backup of subtree of database.	BKPTREE copies subtree of project to tape.	I-	S	-
Permit backup of baseline.	Not supported -- TRANSMIT stores entire subtrees or individual files, but does not accept a list of baseline objects.	NI	U/U	-
Reconstruct database or baseline from backup.	RESTORE reads copy of database from tape -- can be used to reconstruct entire database, subtree or individual files.	I-	S	-
Maintain index of backup tapes.	Not supported -- system administrator must maintain manual index of volume numbers and search names.	NI	U/S	-

TABLE XVI (B) Reliability Evaluation

Requirement: Maintain Off-line Supplementary Storage.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Allow users to store and retrieve infrequently used objects off-line.	ARCHIVE and UNARCHIVE tools allow users to send list of objects or subtrees they want swapped off-line or back on-line.	I	S	B
	ROLLOUT and ROLLIN tools allow system operator to store and retrieve objects, automatically retrieving disc space.	I-	S	-
Maintain, in database, specified attributes of objects stored off-line.	System defined list of attributes maintained on line -- includes tape volume and search name for off-line storage.	I-	U/U	-

TABLE XVI (C) Reliability Evaluation

Requirement: Maintain Derivation History.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Record input objects and tool used to create object.	Automatically recorded for tools classified as generators -- user can create new generators.	I	S	B
Support forward and backward tracing of derivations.	Only forward tracing supported, only derivation count affected on source objects.	I	U/U	-
Prevent deletion of input objects as long as derived object exists.	When derivation count is greater than zero, object can not be deleted -- ROLLOUT still possible.	I	S	A

TABLE XVII (A) Attribute Evaluation

Requirement: Maintain Object Attributes.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Provide attributes for every object and directory.	Automatically provided.	I	S	A
Support addition of user defined attributes.	CHATTR allows users to create new attributes.	I	S	D
Maintain attributes automatically when appropriate.	Pre-defined system attributes maintained automatically.	I	S	A
Permit attributes to be modified by system users.	CHATTR allows user to change value of attributes except some system required ones.	I	S	B

TABLE XVII (B) Attribute Evaluation

Requirement: Maintain Object Associations.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Provide associations for every object and directory.	Automatically available.	I	S	A
Support addition of user defined associations.	CHASS allows user to create new associations.	I	S	B
Maintain associations automatically when appropriate.	Associations used to maintain derivations are maintained by system including absolute reference not affected by changes in path name.	I	S	A
Permit associations to be modified by system users.	CHASS allows user to change association values.	I	S	C
Insure that associations not affected by changes in path names of referenced objects.	Only supported for associations created as a result of a derivation.	NI	U/U	-

TABLE XVII (C) Attribute Evaluation

Requirement: Support APSE Expansion.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Provide automatic attribute support for user-added tools.	All attribute management tools are available to builders of new APSE tools.	I	S	D
Provide automatic association support for user-added tools.	All association management tools are available to builders of new APSE tools.	I	S	D
Provide attribute and association support for command files.	Available, but not well supported.	I	S	F

TABLE XVII (D) Attribute Evaluation

Requirement: Support Retrieval by Attribute Value.

Evaluation Criteria	Evaluation	Implement	Success	Simplic
Permit retrieval of all objects in sub-tree by attribute value reference.	Not supported -- LST will list a specific attribute for all objects in subtree, but select operation is not supported.	NI	U/U	-
Support use of associations as input to APSE tools.	Command language provides poor interface -- results of LST_ASS command cannot be fed to next command in command file.	NI	U/U	-

BIBLIOGRAPHY

Bersoff, Edward H., et. al. "Software Configuration Management: A Tutorial," Computer: 6-14 (January 1979).

Buxton, John N. and Larry E. Druffel. Requirements for an Ada Programming Support Environment: Rationale for Stoneman," IEEE Computer Society's 4th International Computer Software and Applications Conference. 66-72. NewYork, NewYork, IEEE, October 1980.

Department of the Navy. A Software Engineering Environment for the Navy. Report of the NAVMAT Software Engineering Environment Working Group. March 31, 1982.

Eanes, R. Sterling, et. al. "An Environment for Producing Well-Engineered Microcomputer Software," Proceedings of the 4th International Conference on Software Engineering: IEEE, 386-398 (1979).

Howden, William E. "Contemporary Software Development Environments," Communications of the ACM, 25 (5): 318-329 (May 1982).

Huff, Karen E. "A Database Model For Effective Configuration Management in the Programming Environment," Proceedings of the 6th International Conference on Software Engineering: IEEE, 54-61 (1981).

Intermetrics, Inc., Ada Integrated Environment I Design Rationale, Prepared for Rome Air Development Center, Intermetrics, 15 March 1981.

McCarthy, Rita. "Applying the Technique of Configuration Management to Software," Tutorial: Software Configuration Management. 42-47/ New York: IEEE Computer Society, October 1980.

Metzger, J.J. and Dniestrowski, A., "PLATINE, A Software Engineering Environment," 1983 Softfair -- A Conference on Software Development Tools, Techniques, and Alternatives. 193-199. Silver Spring: IEEE Computer Society, 1983.

Notkin, David S. and A. Nico Habermann, "Software Development Environment Issues as Related to Ada," Tutorial: Software Developments. 107-137. New York: IEEE Computer Society, 1981.

Searle, Lloyd V. An Air Force Guide to Computer Program Configuration Management. Prepared for Deputy for Command and Management Systems, Electronic Systems Division. Santa Monica, CA, System Development Corporation, August 1977.

SofTech, Incorporated. ALS VAX/VMS Target Users Reference Manual, November, 1983a.

-----. The ALS VAX/VMS Textbook, November, 1983b.

-----. An Introduction to SADT Structured Analysis and Design Technique, Waltham: SofTech, November, 1976.

"Stoneman," Requirements for Ada Programming Environments, Department of Defense, February, 1980.

Stenning, Vic, et al. "The Ada Environment: A Perspective," Tutorial: Software Developments. 36-45. New York: IEEE Computer Society, 1981.

Stuebing, H. G. "A Modern Facility for Software Production and Maintenance," IEEE Computer Society's 4th International Computer Software and Applications Conference. 407-418. NewYork, NewYork, IEEE, October 1980.

Texas Instruments, Inc. Ada Integrated Environment III Computer Program Development Specification. Lewisville, TX: Report prepared for Rome Air Development Center, RADC-TR-81-360, Vol II, (December 1981).

Wasserman, Anthony I., "The Ecology of Software Development Environments," Tutorial: Software Developments. 47-52. New York: IEEE Computer Society, 1981.

Wegner, Peter. "The Ada Language and Environment," ACM SIGSOFT, Software Engineering Notes, 5 (2): 8-14 (April 1980).

Zucker, Sandra, "Automating the Configuration Management Process," 1983 Softfair -- A Conference on Software Development Tools, Techniques, and Alternatives. 164-172. Silver Spring: IEEE Computer Society, 1983.

Vita

Mark S. Orndorff was born on 25 May 1955 in Arlington, Virginia. He attended Washington-Lee High School in Arlington and graduated in 1973. In September of that year, he enrolled in Brown University in Providence, Rhode Island. After one year of study and a year employed by Western Electric, he transferred to the University of Virginia in Charlottesville, Virginia and subsequently graduated with High Distinction, receiving a Bachelors of Arts degree in Environmental Science in May 1978. After graduation, Captain Orndorff was commissioned in the U.S. Army and attended the Signal Officers Basic Course at Fort Gordon, Georgia. He was then assigned as a platoon leader in B Company, 5th Signal Battalion, 5th Infantry Division (Mechanized) at Fort Polk, Louisiana. While at Fort Polk, he also served as a platoon leader in A Company and as the A Company Commander. After leaving Fort Polk, Captain Orndorff attended the Signal Officers Advanced Course at Fort Gordon, Georgia and the Teleprocessing Operations Course at the Air Force Institute of Technology at Wright Patterson AFB, Ohio. After completing the Teleprocessing Operations Course, he entered the Air Force Institute of Technology School of Engineering.

Permanent address: 883 N. Jefferson St.
Arlington, VA 22205

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/EE/84M-1	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION AFIT/ENG Air Force Institute of Technology 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) Evaluation of Automated Configuration Management Tools in Ada Programming Support Environments	
12. PERSONAL AUTHOR(S) Orndorff, Mark Stephen	
13a. TYPE OF REPORT MS THESIS 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) 1984, MAR, 7 15. PAGE COUNT 132
16. SUPPLEMENTARY NOTATION	Approved for public release IAW AFR 190-17 LYNN E. WOLAVER Dean for Research and Professional Development Air Force Institute of Technology (ATC)

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	
09	02		CONFIGURATION MANAGEMENT, ADA, PROGRAMMING ENVIRONMENTS SOFTWARE ENGINEERING ENVIRONMENTS, APSE, SOFTWARE MAINTENANCE

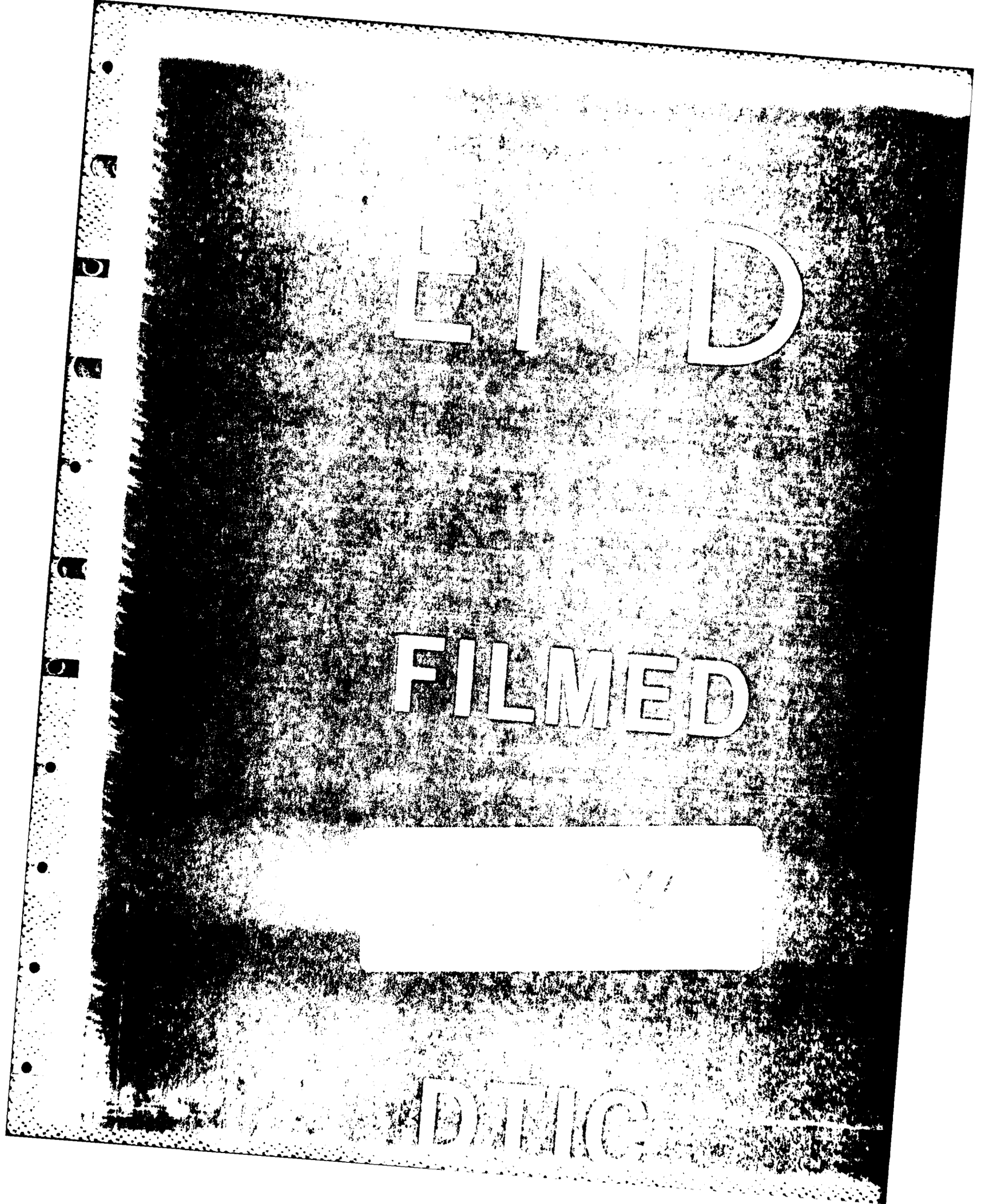
19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This investigation studied the task of configuration management of computer software systems. First, a detailed definition of configuration management from the perspectives of project management and project engineers was developed. This definition was used to conduct a requirements analysis of the support required in automated programming environments for the configuration management task. Based on these requirements, evaluation criteria were developed that were appropriate for the

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL M.R. VARRIEUR	22b TELEPHONE NUMBER (Include Area Code) 513-255-3576 22c. OFFICE SYMBOL

evaluation of configuration management tools designed to satisfy the 1980 Stoneman requirements document. These evaluation criteria were used to evaluate the November 1983 release of the Army's Ada Language System.

The requirements and evaluation criteria developed in this thesis are designed to provide designers and purchasers of Ada Programming Support Environments (APSE) with the tools necessary to determine the effectiveness of an APSE implementation in supporting the task of configuration management of large software projects developed for embedded computer systems.


END
FILMED
DTIC