Documentation Production Under Next Generation Technologies

Richard E. Nance, Benjamin J. Keller, and Dave Boldery

TR 89-26

Technical Report SRC-89-001†

DOCUMENTATION PRODUCTION UNDER NEXT GENERATION TECHNOLOGIES

Final Report

Richard E. Nance
Benjamin J. Keller
Dave Boldery

Systems Research Center
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

15 February 1989

† Cross referenced as Technical Report TR-89-26, Department of Computer Science, Virginia Tech.

Table of Contents

Section	Title	Page
1.	Introduction	1
1.1	Statement of Work	1
1.2	Project Approach and Report Organization	2
2.	Characterizing the Documentation Roles	2
2.1	Explicating the Life-Cycle Relationships	4
2.2	Background Contributions	4
2.2.1	The Draco Approach	4
2.2.2	The Laws of Programming	5
2.3	The Abstraction Refinement Model (ARM)	6
3.	Applying the Abstraction Refinement Model	7
3.1	Software Development in the Abstraction Refinement Model	7
3.2	Software Maintenance in ARM	8
3.2.1	Corrective Maintenance	11
3.2.2	Adaptive Maintenance	11
3.2.3	Perfective Maintenance	11
3.2.4	Preventive Maintenance	13
3.3	Critique of the Abstraction Refinement Model	13
4.	Documentation Sets	13
4.1	The Standard Set	14
4.1.1	The Software Development Phases	14
4.1.2	Category/Phase Relationships	15
4.1.3	Standard Set Specification Tracing	18
4.2	The Maintenance Set	24
4.2.1	The Ideal Maintenance Kernel	24
4.2.2	The Realistic Maintenance Kernel	24
5.	Documentation in the Maintenance Phase	27
5.1	Persistent Problem: Incomplete Development Information	28
5.2	Unrecognized Problem: Maintenance Documentation	28
5.3	Pragmatic Solution: Reverse Engineering	29
5.3.1	The Ideal Process	29
5.3.2	The Achieveable Process	29
6.	The Abstraction Refinement Model in the ADDS Context	30
6.1	Documentation Guidelines and Specification Languages	31
6.2	The Roles of ADDS/MicroADDS in Software Maintenance	33
6.2.1	The Current and Planned Roles	33
6.2.2	Alternative Roles	34
6.3	Improving the AEGIS Maintenance Function	35
7.	Concluding Recommendations	35
	References	37
	Appendix A	1
	Appendix B	6
	Appendix C	7

Figure		Page
1	Lehman Two-Leg Model	3
2	Giddings' Domain Dependent Software Life Cycle	3
3	Example of a Domain Structure Graph	5
4	Illustration of a Semi-lattice Formed by the Abstraction Refinement Model	7
5	Origination of the Embedded Software Development Process	7
6	Illustration of the Resolution of Abstraction Through Development Activities Resulting in the Program P	8
7	Illustrations of the Approaches to the Maintenance Task	10
8	Distribution Functions Illustrating the Location of the Least Common Abstraction for Each Form of Maintenance	12
9	The Seven Specification Levels for the Standard Set	19
10	Illustration of Document Objects Traced Through Three Specification Levels	20
11	Illustration of Delayed and Umbrella Document Objects in Specifications	22
12	Standard Set Document Category Trace Graph	23
13	Development Document Composition of the Realistic Kernel	27
14	The Shadow Path Illustration of the Reverse Engineering Role	31
15	The MicroADDS Development Plan	33

ABSTRACT

This report describes the development of the Abstraction Refinement Model as a basis for linking the development and maintenance tasks in software systems. Documentation is critical in both efforts, and the reliance on development documentation during maintenance is characterized by the model and through a characterization of the development documentation requirement stipulated under DoD-STD-2167A. The Abstraction Refinement Model enables a coherent characterization of the reverse engineering requirements generally caused by a faulty or inadequately documented development process. Within the context of the model, the Automated Documentation Design System (ADDS) is characterized, and the system is evaluated with regard to its current capabilities versus future potential. A set of recommendations regarding ADDS concludes the report.

1. Introduction

This report concludes the third in a series of studies of the Automated Design Description System (ADDS), extending over the period 15 July 1987 to 31 December 1988. The focus of each project is reflected in the titles:

The Automated Design Description System (ADDS)

Prospects for Automated Documentation in Support of Software Quality Assurance: A Study of ADDS

Documentation Production Under Next Generation Technologies

Both the systems engineering and the software engineering functions are crucially dependent on documentation for the conclusion of a successful development process. Experience has taught the Navy software development community, contractors and government employees alike, that documentation cannot be considered an after-the-fact requirement, given little attention and produced by the most junior employees. One of the key statements explicit in DoD standard 2167A is that documentation must be accorded the highest priority in contracted deliverables. Standards, guidelines, directives, and educational offerings clearly show a trend toward more formal guidelines and more automated support of documentation production in the development process.

1.1 Statement of Work

The project activities divide into two tasks:

Task 1. Assuming a predefined set of documentation guidelines and Ada-like specification languages, e.g., BYRON, identify related document quality indicators that can provide a basis for the automatic analysis of documentation.

Task 2. Investigate an ADDS-like system (or environment) that (a) provides for document synthesis based on a PDL (program design language) system definition, and (b) supports automatic document analysis through the detection of DQIs identified above (Task 1).

Preceding project reports indicate the difficulty in characterizing the documentation production process followed in current Navy practice. Further, a lack of understanding of the reverse engineering strategy for improvement of development documentation is noted in the earlier interviews with ADDS users. Consequently, the approach to both tasks above initiates

with an explanation of the documentation roles in both the development and maintenance phases of the software life cycle. The understanding of the necessary interdependence of the maintenance process contributes to a clear explication of the reverse engineering objectives.

The definition of document sets, necessary for both the development and maintenance processes, provides an essential link between the development and maintenance phases. Additionally, the document sets prescribe responsibilities of the development process that if neglected, impose increased cost in the subsequent deployed software support phase. Limits to the automated analysis of documentation can be recognized, even in an ideal process, but the utilization of emerging technologies offers some exciting potentials.

1.2 Project Approach and Report Organization

This report describes the development of an Abstraction Refinement Model (ARM) that portrays the development process as a series of specifications, the product of design decisions, concluding with an implementation. This model is described in Section 2, and the utilization of this model in the software maintenance phase is described in the third section. The definition of related document sets and their importance in the support of deployed software is treated in Section 4. Section 5 utilizes the ARM to explain persistent problems in the maintenance process and to provide an insightful description of reverse engineering objectives. The model is considered in the more specific ADDS context in Section 6. Section 7 contains concluding recommendations.

2. Characterizing the Documentation Roles

The earliest models of the software life cycle often portray the development process as stagewise (Benington, 1956) or as "waterfalling" (Boehm, 1976). While giving visibility to the importance of the development process, and to the documentation function within it, the early models tend to minimize the feedback effect. Later models such as Lehman's two-leg model (Lehman, et al., 1984), emphasize the feedback processes, see Figure 1. The domain dependent software life cycle model, offered by Giddings (1984) and shown in Figure 2, explicitly notes the prototyping function, the crux of the rapid prototyping methodology which has become more popular in recent years (Jenkins, 1983).

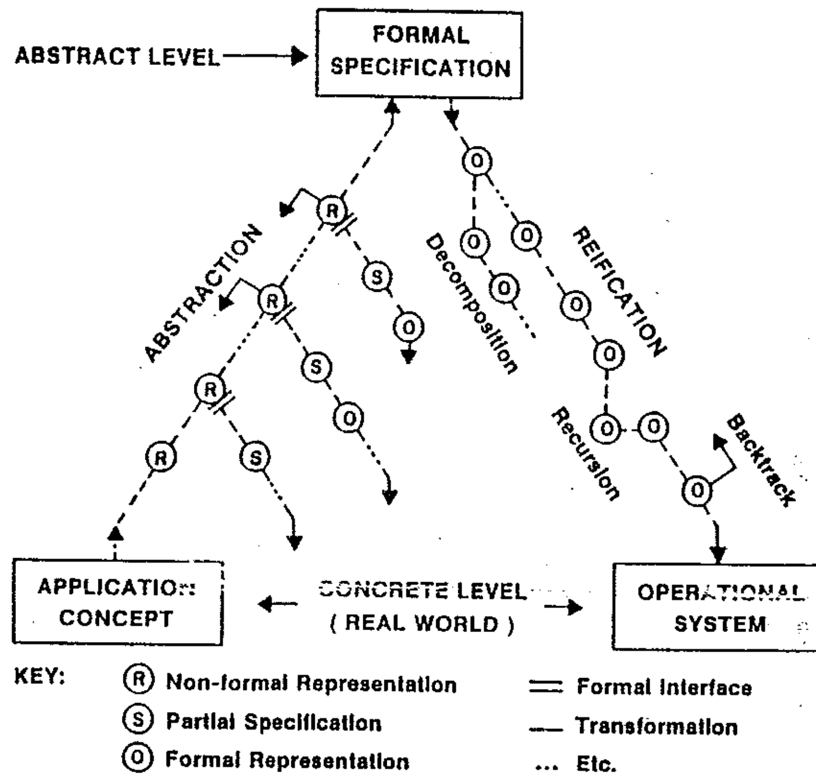

Figure 1. Lehman Two-Leg Model

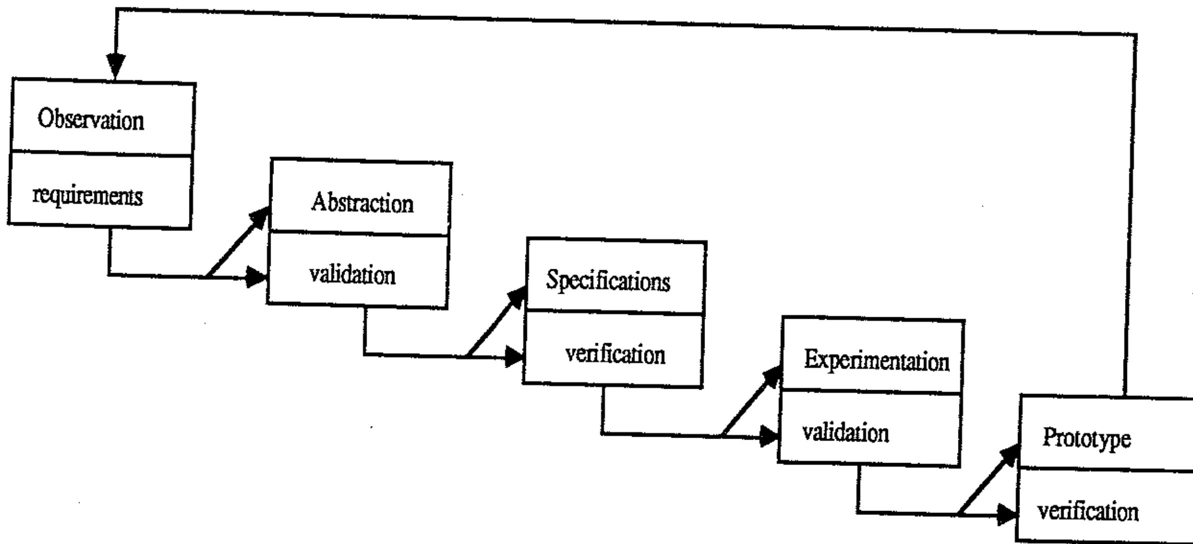

Figure 2: Giddings' Domain Dependent Software Life Cycle

These models and others, such as Boehm's spiral model (Boehm, 1986), focus on the development process, offering no instructive guidance as to the relationship with the maintenance

(deployment support) process. The result is a gap, which has complicated efforts to relate the two processes in a mutually supportive characterization.

2.1 Explicating the Life-Cycle Relationships

Clearly, documentation must be inherent in the specification process in order to fulfill the expectations of current software development methodologies. Documentation of requirements, high level and detailed design, program design, and the implementation product must be supported in the development methodology. Equally important is the documentation of unit testing, integration and subsystem testing, and the subsequent experiences with the deployed product. While software configuration management and software quality assurance are perceived as critically dependent on the execution of the documentation function, no existing model relating the two attempts to describe the cause and effect relationships.

The documentation of design decisions, emphasizing both the alternatives and the reasons for discounting or preferring certain alternatives, represents information produced in the development process that is crucial in the execution of the maintenance process. A model that demonstrates this importance is necessary to establish this crucial linkage.

2.2 Background Contributions

Realization of the model demonstrating the linkage between development and maintenance phases draws upon two prior research developments: the Draco approach (Neighbors, 1984) and the enunciation of the laws of programming (Hoare, et al., 1987).

2.2.1 The Draco Approach

Developed at the University of California-Irvine, the Draco approach has the primary objective of developing reusable software. An environment based on the Draco approach is described in (Neighbors, 1980).

The Draco approach is contrasted with other approaches following the distinctions of language spectra (Neighbors, 1984, p.564):

> *Wide-spectrum-languages* represent the developing program from specification to final executable code; while *narrow-spectrum-languages* offer separate language forms for each phase of program development.

Characterized as a domain language approach, Draco represents application domain knowledge in terms of sets of objects with defined operators and relations to comprise the specifications. Specifications are made using a special language defined to express this domain knowledge. A domain structure graph, shown in Figure 3, characterizes the progression from an abstract

specification to a concrete program.

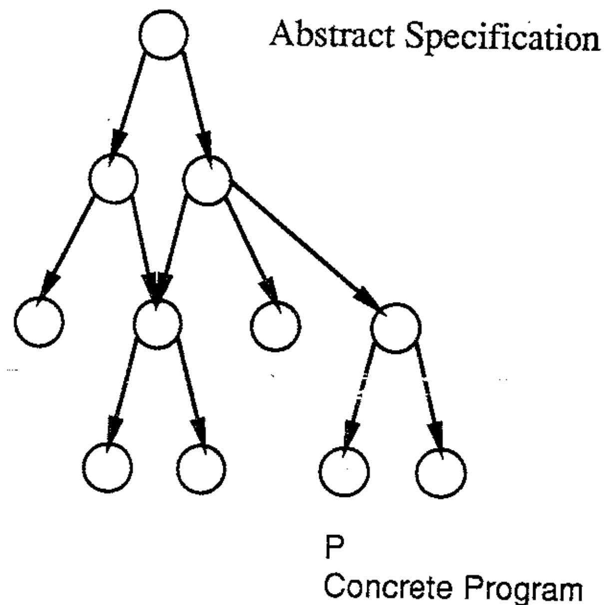

Figure 3. Example of a Domain Structure Graph (Adapted from [Arango 1985])

The term "specification" is used in a very general sense to mean a complete description of the system at any level of representation (or abstraction). Thus a "specification" is a representation of the system, be it in the form of requirement specifications or a program listing.

The domain structure graph depicts the successive refinements of the high level specification into corresponding lower level specifications based on design decisions at each level. The specifications are represented by the nodes and the refinements (design decisions), by the arcs (Arango, et al., 1985). The lowest level represents the concrete program P produced in the final implementation stage. Other branches represent either implementations or intermediate design stages that can be expanded into an implementation as further design decisions are made.

2.2.2 The Laws of Programming

A formal view of program development based upon an algebra of programs is described in (Hoare, 1985) and (Hoare, et al., 1987). A special concern is the partial ordering on programs emanating from the concept of an "abstract command" or program.

An abstract command is defined as a specification of the general behavior *desired* of a program (Hoare, 1985). Using this idea, the partial ordering on two abstract programs X and Y

can be defined. If X is less (or as equally) abstract as Y, and X "satisfies" Y then the relationship can be written $Y \supseteq X$. Note that "X satisfies Y" means that the requirements (computational or otherwise) given by Y are completely met by X†. The partial ordering relationship among specifications, or abstract programs, allows the definition of a semi-lattice (Gratzer, 1978). Coupled with the Draco approach, the semi-lattice structure provides the basis for the Abstraction Refinement Model developed in the following section.

2.3 The Abstraction Refinement Model (ARM)

The Abstraction Refinement Model combines the domain structure graph of the Draco approach and the partial ordering on abstractions described in the work of Hoare. In actuality, the kernel of the same idea is embodied in both. While the partial ordering defines the semi-lattice, the domain structure graph represents a portion of the semi-lattice rooted with the original specification.

Let Y be a specification and X be a concrete realization of Y. The partial ordering relating the two can then be written as:

$$Y \supseteq X$$

which indicates that X satisfies a specification Y. This partial ordering then defines an upper semi-lattice (or *join semi-lattice*), and a most general specification (MGS) can be defined that is satisfied by all conceiveable specifications. This claim is equivalent to asserting that each of the two specifications has a least upper bound (or "least common abstraction)," see (Arango, et al., 1985) but not necessarily a greatest lower bound ("greatest common realization").

The lattice-theoretic basis for the abstraction refinement model can be viewed with a conical illustration shown in Figure 4. The MGS assumes the apex, and all programs which satisfy the MGS fall on the planar section at the base of the cone.

---

† As an example of the *satisfies* relationship, let Y be a specification of the form "compute the square root of variable Z." X could then be of the form "compute the square root of variable Z with algorithm α " or it could be the algorithm itself.

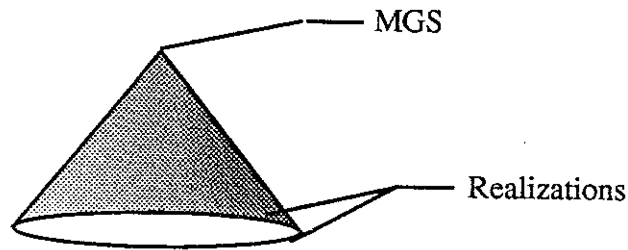

Figure 4. Illustration of a Semi-lattice Formed by the Abstraction Refinement Model

3. Applying the Abstraction Refinement Model

The Abstraction Refinement Model characterizes the inherent dependency of the maintenance function on the software development function through the required documentation produced in the development process. This dependency is described in subsequent paragraphs.

3.1 Software Development in the Abstraction Refinement Model

Requirement specifications are constructed during development according to the systems engineering methodology of the application domain. Typically described as systems analysis, this activity in an embedded software development process results in a node representing a specification from which software development initiates. In Figure 5 this node is shown as somewhere in the middle of the semi-lattice, illustrated by the cone. Beginning from this node is defined a domain structure graph (DSG), following the Draco approach.

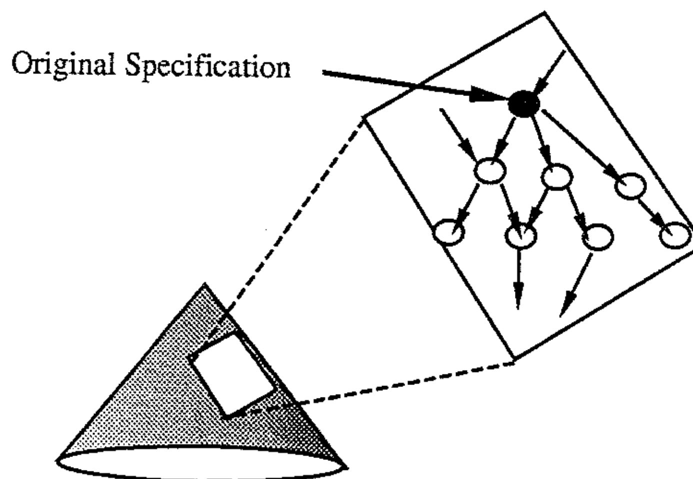

Figure 5. Origination of the Embedded Software Development Process

Development proceeds by defining a path through the DSG from the original specification. The DSG is characterized as a tree in this particular example. The path is constructed through design decisions that effect refinements of higher level specifications to produce lower level ones. By capturing these design decisions in the documentation of the development process, the reusability of software components is accommodated. Such a path is shown in Figure 6 below, although it may actually represent only a segment (the software development branch) of a particular *systems* development process.

The edges in the graph depicted in Figure 6 can be considered development activities: design decisions, refinements, or transformations. No distinction is made among the three; however, the concept of a transformation is quite instructive at this point. The implication of "transformation" is a design decision that maps a specification into a less abstract specification which satisfies its predecessor. In principle, an inverse mapping between the two specifications can be assumed to exist (from the more concrete to the more abstract).

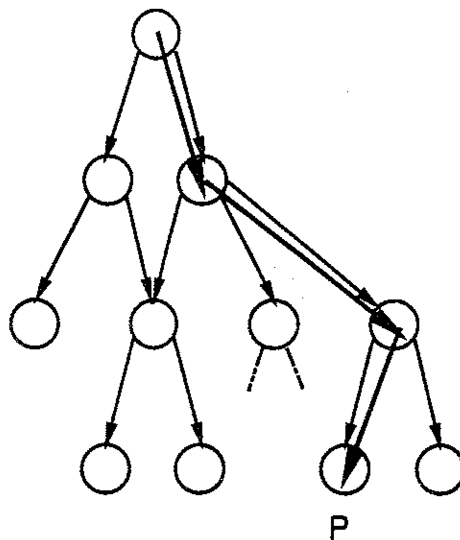

Figure 6. Illustration of the Resolution of Abstraction through Development Activities Resulting in the Program P

3.2 Software Maintenance in ARM

Software maintenance can be characterized as a transformation of a program P into a program P', or more conventionally by the mapping

$$\mathbf{P} \xrightarrow{m} \mathbf{P'}$$

The required transformation shown in Figure 7, depicts an unbalanced DSG, representing the absence of concrete realizations in all paths (having made an alternative design decision, no

further activities are pursued along that path). The transformation shown in Figure 7 might be accomplished in one of three ways:

(1) A *blind transformation*, which represents an attempt to realize P' only from actions on P, neglecting the predecessors of P. Such a transformation is considered undesirable and probably impossible to achieve. An example of such an attempt is the correction of a program malfunction without recourse to other than the source code and internal documentation.

(2) *Redevelopment* from the original specification. This approach requires a complete regeneration of design decisions in the determinations of the path terminating with P'. In practice, this is likely to be done, but it is obviously quite expensive.

(3) Identification of the *least common abstraction(s)*, which requires an inverse mapping from P to S followed by a direct mapping from S to P'. In lattice-theoretic terms the inverse mapping is called the *join* of P and P' and results in the definition of the lowest common ancestor of both nodes.

(Note that arrows are omitted from the edges in Figure 7 to denote that the activities can proceed in either direction and are no longer representing program development only.) Clearly, the third approach represents the preferred alternative; however, the practicality of defining the least common abstraction places a heavy documentation responsibility on the development process.

All too often, the maintenance function is viewed simply as an error correction process. That perception dominates the management perception at NSWC to the extent that "maintenance" is considered to have negative connotations. The term "in-service engineering" is preferred by some; nevertheless, the term "maintenance," as used throughout this report, neither has negative connotations nor is viewed as describing only an error correcting responsibility. Swanson (1976) identifies four application purposes that are grouped under and stimulate different forms of maintenance: (1) corrective, (2) adaptive, (3) perfective, and (4) preventive.

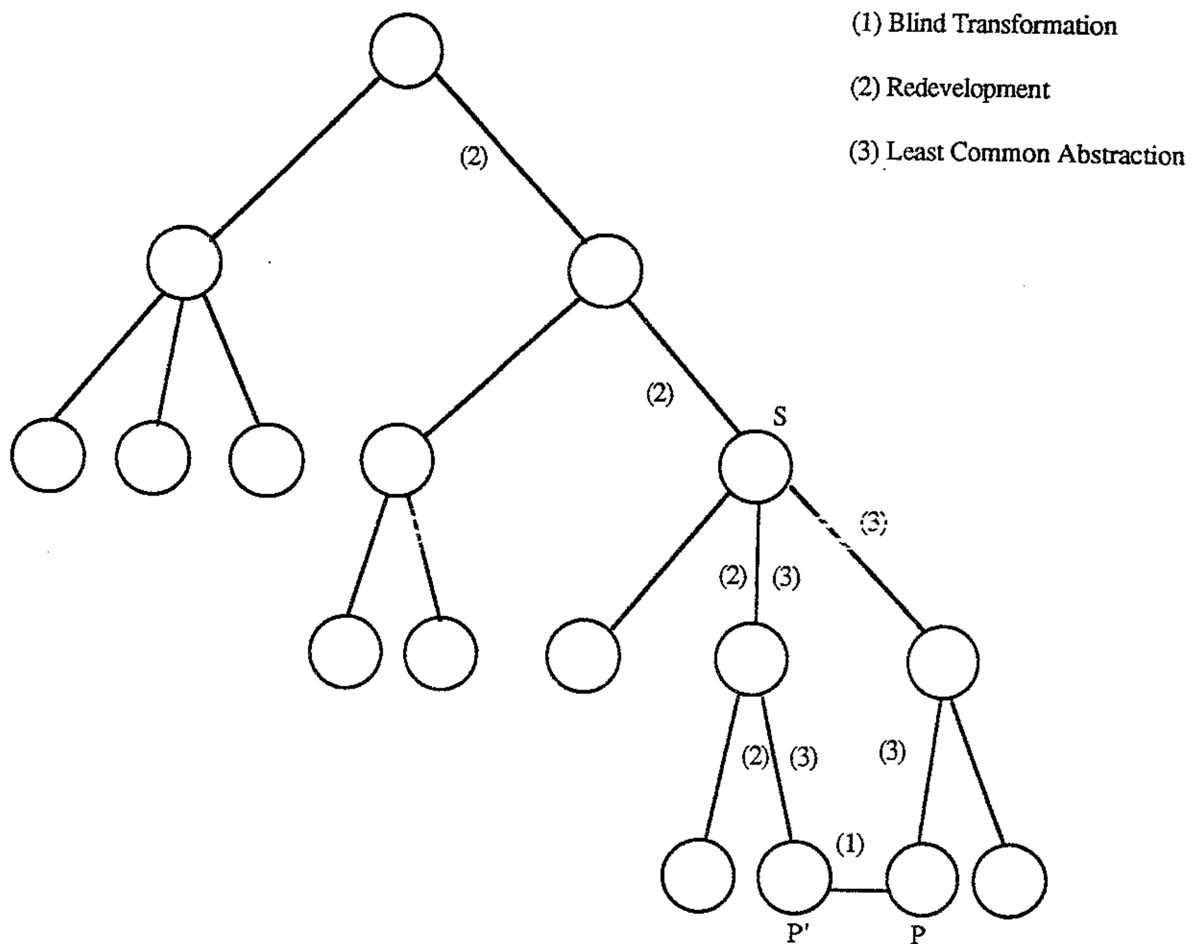

Figure 7. Illustrations of the Approaches to the Maintenance Task (Realization of P' from P)

The forms of maintenance, because of differences in their underlying purposes, relate to the development process in different ways. In terms of development documentation, the source of information needed to initiate the maintenance task is a primary concern. The ARM represents this source as the LCA, and the likelihood of the LCA location at various levels of the DSG is conveyed in the distribution functions for each form of maintenance in the different parts of Figure 8. These distribution functions are illustrative and based on no "hard" empirical data; yet, the rationale given below for their characterizations seems quite plausible. Although the underlying random variable for such a distribution function is discrete, the portrayal of a continuous function simplifies the illustration.

3.2.1 Corrective Maintenance

The correction of an error in the definition of system needs is the purpose of *corrective maintenance.* The error could occur at any point in the development process, but with reference to the use of ARM in Figure 8(a), the assumption seems warranted that errors are less likely early in the process than later. Certainly, all development methodologies seek to realize this outcome.

The two distribution functions shown in Figure 8(b) indicate: (1) a linearly increasing likelihood that corrective maintenance initiates at the more "concrete" specification levels, and (2) a disproportionately higher probability that the initiation occurs at the lower levels of the graph. Either form might describe a particular software system, depending on the development and maintenance methodologies, among several other factors.

3.2.2 Adaptive Maintenance

The addition of functionality to reflect a better understanding of system needs is the purpose of *adaptive maintenance.* Belady and Lehman (1976) note that the plotting of error behavior for OS 360 clearly demonstrates the points where functional improvements are made in the subject system. The number of Software Trouble Reports jumps drastically following each point.

Figure 8(c) shows the hypothesized initiation level probability for adaptive maintenance. The middle levels of the tree form the more likely initiation points. Since nodes nearer the root represent a more basic, and consequently more costly, functional addition. Nodes nearer the leaves of the graph represent less costly functional changes tied more closely to implementation and more likely to be adopted.

3.2.3 Perfective Maintenance

In contrast with adaptive maintenance, *perfective maintenance* has the purpose of improving system functionality in its ability to meet current needs. This is the conventional "product improvement" process whereby convenience changes are realized as well as improved capabilities.

Figure 8(d) shows the initiation level for perfective maintenance, as more likely to be nodes at the middle levels. At these levels functional definition tends to be completed and the transition from design decisions to implementation occurs here.

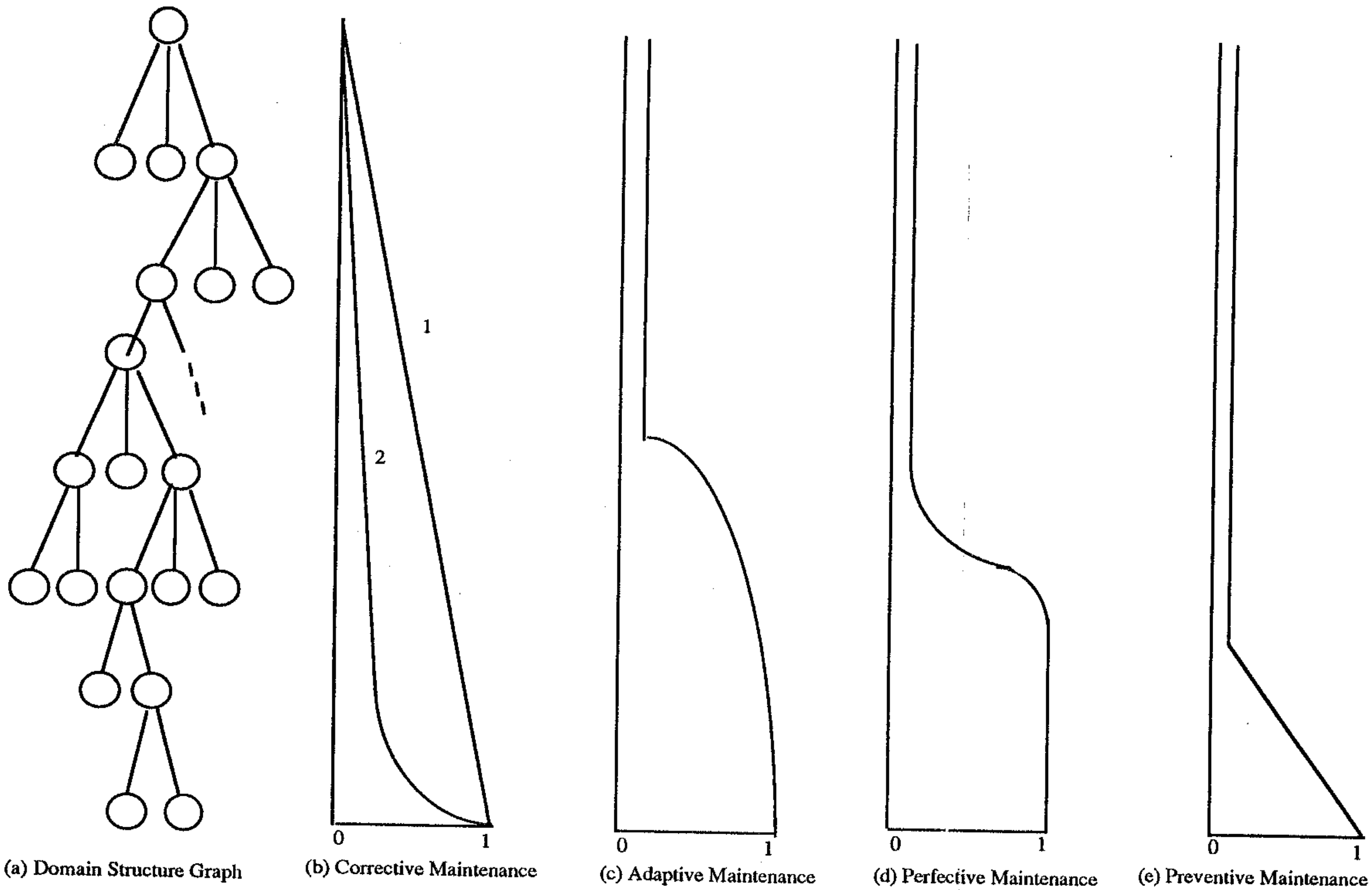

Figure 8. Distribution Functions Illustrating the Location of the Least Common Abstraction for Each Form of Maintenance

3.2.4 Preventive Maintenance

The purpose of *preventive maintenance* is to make the system more robust to withstand increased uncertainty in input or hardware availability. Expansion of inputs or an increase in error tolerance are motivations for this form. The tendency might be to group preventive with perfective maintenance; however, the emphasis in preventive maintenance must be predominately at the implementation levels of the tree. This is shown in Figure 8(e), which depicts the initiation as more likely to be at the lowest two levels.

3.3 Critique of the Abstraction Refinement Model

The abstraction refinement model (ARM) is a representation of software development and maintenance interdependency drawn from: empirical observation of the development and maintenance processes, research findings from the current literature, and some speculations about the nature of only partially understood phenonoma. As a model in the context used herein, it represents a *theory* to be judged according to two criteria:

> Descriptive validity - does it describe what appears to exist in the real world giving some plausible explanatory basis for it?
>
> Instructive utility - operating from the descriptive basis, can problem points or areas where actions can lead to improvement be identified?

The ARM representation of interdependencies, and its characterization of the forms of software maintenance, seem to provide an adequate, if not complete, descriptive basis. However, the instructive requirement is seen as very underdeveloped. At this point the ARM seems helpful and promising in terms of its potential. Even at that, a better explication of the *raison d'etre* for the reverse engineering role in the maintenance process is possible using it. That subject is taken up in Section 5.

4. Documentation Sets

Document sets support the information needs of both software development and maintenance activities. With respect to meeting the needs of both activities, the standard set and the maintenance set are defined. The standard set is to support the information needs of all software development activities, and the maintenance set is to support the information needs of all maintenance activities. Development of the larger set, i.e., the standard set, is predicated in the belief that a substantial subset of the maintenance set is derivable. The maintenance set is composed of documents from the standard set, coupled with elements added because of the inability to recognize all needs of maintenance during development.

Document sets are collections of document objects, each defined by a standard guideline or

directive. A document object (or simply object) is a unit of information found in the document set. The units of information are considered to vary in size. For example, one object may be the unit of information contained in an entire paragraph, while another may be the unit of information contained in one line of a document. To identify objects and to reflect their information content, identifiers are assigned to each. For example, if a document object is considered to be a unit of information specifying Ada as the required implementation language, then the identifier for the object could be "software language requirement".

4.1 The Standard Set

The contents of the standard set are determined by a review of five documentation standards, see Appendix B, with an emphasis placed on the documentation standard DoD-STD-2167A whose influence is evident in the use of certain key words in this section. See Appendix C for a complete list of these key words. The review of the standards produces approximately 250 objects grouped into 37 separate categories.

Document objects are categorized based on their support of a common identifiable objective. For example, programming language requirements, operating system requirements, and programming standards are document objects placed in the "programming requirements" category. This action assumes the objects support a common objective: defining requirements for implementation. Document objects are further organized by relating the categories to software development phases. A category and a developmental phase are related if completion of the phase produces document objects found in the category. The category/phase relationship provides an effective representation method for the standard set.

4.1.1 The Software Development Phases

The software development phases are described in terms of purpose and composite activities. The activities of each phase are intended to support an ideal development process; more specifically, activities are included which should help assure the development of a high quality, reliable, and maintainable product. Taken from several sources (DoD-STD-2167A, Pressman 1986, Abbott 1986) the developmental phases include:

> *Software Requirements Analysis* - Bridges the gap between computer systems engineering and software design. Specifies engineering requirements, indicates software interfaces, and establishes developmental constraints. Provides the software designers with a representation which can be translated into data, architectural, and procedural design. Defines a complete set of formal qualification requirements, which assure proper development of system components.

Preliminary Design - Produces a convincing data and software architectural design by development of a preliminary design that meets system requirements. Allocates established requirements to the Computer Software Components (CSCs) of each Computer Software Configuration Item (CSCI). Establishes test requirements for conducting CSC integration and testing, and determines the formal qualification requirements which apply to each CSC.

Detailed Design - Focuses on refinements to the data and software architectural design which lead to a detailed data structure and algorithmic representation of the system. Allocates established requirements to the Computer Software Units (CSUs) of each CSC. Establishes actual test cases for conducting CSC integration and testing, CSU testing, and determines the formal qualification tests cases which apply to each CSCI.

Coding and CSU Testing - Translates the detailed design of the system into a programming language realization. Prescribes for tests of coded CSUs according to unit test cases; Includes corrections for any errors which occur because of testing. Creates CSC test procedures to aid CSC testing.

CSC Integration and Testing - In a controlled fashion, integrates CSUs to form CSCs while simultaneously conducting tests to uncover errors associated with interfacing. Testing of CSCs with respect to CSC tests cases. Includes provisions and corrections of errors found in the design documentation and code. Creates testing procedures for each formal qualification test case to aid formal qualification testing.

CSCI Testing - Tests the computer software configuration items with respect to the formal qualification test cases. Performs revisions and error corrections necessary in the design documentation and code.

System Integration and Testing - Integrates system software through a series of validation tests. Includes error corrections and revisions as necessary in the design documentation and code. (This is often considered to fall outside the scope of the software development activity since, in general, system integration is not performed solely by the software developer.)

4.1.2 Category/Phase Relationships

As mentioned previously, the correspondence of document objects to categories is related to the development phases. The relationship between object categories and phases are defined phase by phase. For each phase the related document categories are listed and defined. The numbers serve to identify the category in Figure 12 which follows. For a list of document objects related to each category, see Appendix A.

Requirements Analysis

(1) *Requirements, Adaptation* - Specifies data which can be centrally modified to change the scope of the CSCI operational functions.

(2) *Requirements, Data Base/Files Specification* - Specifies in detail all data bases and files required.

(3) *Requirements, Design* - Specifies design constraints and standards for CSCIs.

(4) *Requirements, Functional Input/Output* - Specifies all input and output requirements for each system function.

(5) *Requirements, Functional Processing/Performance* - Specifies performance factors each system function must satisfy.

(6) *Requirements, Interface* - Specifies requirements for each interface considered to be external to each CSCI.

(7) *Requirements, Programming* - Specifies programming requirements to be met by each CSCI.

(8) *Requirements, Qualification* - Specifies qualification requirements for each CSCI to insure the implemented CSCI meets all specified requirements.

(9) *Requirements, Quality* - Specifies guidelines which must be followed to insure production of a high quality product. (Quality requirements influence the entire developmental process.)

(10) *Requirements, Support* - Specifies the resources required to perform the software development activity.

(11) *Reports, Traceability* - Provides a mapping between the above requirements and the initial requirements found in System/Segment Specifications (SSS), Prime Item Development Specifications (PIDS), or Critical Item Development Specifications (CIDS).

Preliminary Design

(12) *Architecture, CSCI* - Describes overall static structure used in the design of each CSCI. Provides a brief description of the Top Level Computer Software Components (TLCSCs) and their relationships to the Lower Level Computer Software Components (LLCSCs).

(13) *Allocations, CSCI* - Describes how functions and interfaces for each CSCI are allocated to the TLCSCs.

(14) *Design, Preliminary CSCI Interface* - Describes preliminary design for interfaces external to each CSCI.

(15) *Design, Top Level* - Describes each TLCSC included in the top-level design. Includes a detailed description of each TLCSCs (purpose, expected inputs, processing, and outputs).

(16) *Function Control/ Data Flow* - Describes top-level flow of data and execution control within each CSCI.

(17) *Global Information* - Describes data which is common to more than one TLCSC.

(18) *Software Testing, CSC Test Requirements* - Describes test requirements for conducting CSC integration and testing.

(19) *Software Testing, CSCI Formal Qualification Tests* - Describes formal qualification tests to be conducted to comply with qualification requirements.

Detailed Design

(20) *Allocations, TLCSC* - Describes how the functions, performance, and interfaces of each TLCSC are allocated to the LLCSCs.

(21) *Design, Detailed CSCI interface* - Describes a detailed design for the interfaces external to each CSCI.

(22) *Design, LLCSC* - Describes detailed design of each Lower Level Computer Software Component (LLCSC).

(23) *Design, LLCSU* - Describes detailed design of each LLCSU.

(24) *Software testing, CSC test cases* - Describes test cases required for CSC integration and testing.

(25) *Software testing, CSCI formal qualification test cases* - Describes test cases for formal qualification testing.

(26) *Software testing, CSU test requirements and test cases* - Describes CSU testing requirements and actual CSU test cases.

Coding and Unit Testing

(27) *Software testing, CSC test procedures* - Includes actual test procedures required to perform testing of each CSC.

(28) *Software testing, CSU test procedures and test results* - Includes actual test procedures required to perform CSU testing, and the results of performing each test.

(29) *Source code* - Includes actual code produced for each CSU.

CSC Integration and Testing

(30) *Software testing, CSC integration test results* - Includes results of performing each CSC integration test.

(31) *Software testing, CSCI formal qualification test procedures* - Includes actual test procedures required to perform formal qualification testing of each CSCI.

(32) *Updated source code and documentation* - Revisions to the source code and design documentation which undergo design or coding changes based on the results of all testing performed.

CSCI Testing

(33) *Software testing, CSCI Test Results* - Results of performing each CSCI formal qualification test.

(34) *Updated source code and documentation* - Revisions to the source code and design documentation which undergo design or coding changes based on the results of all testing performed.

System Integration and Testing

(35) *System testing, Integration Test Plans* - Includes actual test procedures and test cases used to support system integration.

(36) *System Integration, Test Results* - Results of performing system integration test.

(37) *Updated source code and documentation* - Revisions to the source code and design documentation which undergo design or coding changes based on the results of all testing performed.

4.1.3 Standard Set Specification Tracing

The progression of the development process can be thought of as a series of transformations: abstract specifications to more concrete specifications. Following the assumptions underlying the Abstraction Refinement Model, a specification is a complete description of the software system at some level of development. Seven distinct specification levels exist during system development with each development phase producing a specification less abstract than that of the immediately prior phase. The seven specifications are represented by the simple linear graph shown in Figure 9. The top-most node represents the most abstract specification; the bottom node is the fully tested implemented system. The intermediate nodes represent specifications which are, or have the potential of being, transformed into a fully tested implemented system.

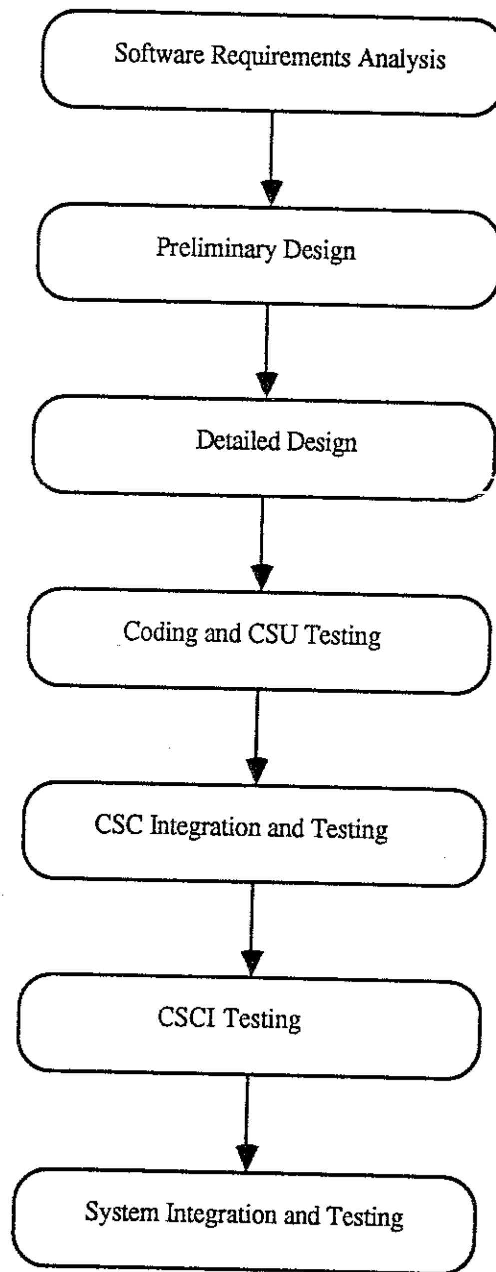

Figure 9. The Seven Specification Levels for the Standard Set

If document objects are considered to form each specification, then transformations between specifications are traceable through their associated document objects. Figure 10 is an example of a transformation between three specifications which is traced through individual document objects. The most abstract specification is composed of six document objects which

are traced to the second level of specification. Lines connecting two document objects imply direct traceability. Two objects are directly traceable between specifications if the object of the successor specification is produced after design decisions are applied to the object of the predecessor specification. The document objects which describe the second specification are further traced to the third specification.

Two types of document object transformations exist between the lower two specifications. In one case, a set of document objects converge to one rectangular node (labeled A). The convergent representation is necessary when individual document objects of a set are not directly traceable, but the set of objects as a whole is traceable. The rectangle A thus represents the convergence of design decisions which transform the set of objects in the predecessor specification into the set of objects in the successor specification. In the other case, two rectangles (labeled B) are merged into one. A merger of two rectangular nodes implies an interdependence exists among design decisions associated with each, i.e., the design decisions associated with one node possibly affect the design decisions associated with the other and vice versa.

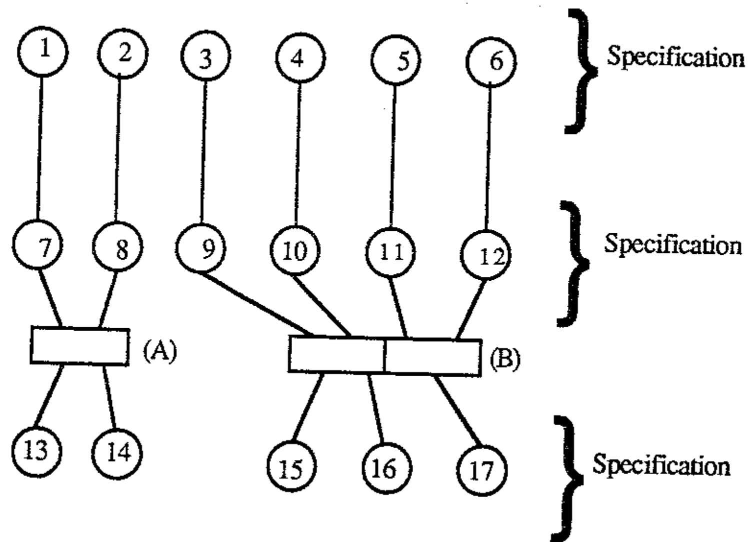

Figure 10. Illustration of Document Objects Traced Through Three Specification Levels. (Rectangles represent the convergence of design decisions.)

In many cases the above description of document object transformations is adequate because most objects of one specification are transformed directly or by sets into document objects of the successor specification. However, such a sufficient transformation does not always occur. For example, objects of one specification are not always transformed into the

successor specification, but to some later specification. Other objects are observed not to necessarily be part of any transformation but appear to affect design decisions. To represent the two additional transformation problems, additional object types are introduced.

A document object at a specification level that is not transformed into a object at the successor specification, but appears in some lower level, is termed a *delayed document object*. Delayed document objects attempt to recognize the need for future specification information that is not vital to current objectives while at the same time supporting the principle of "separation of concerns". Figure 11 is an example of a transformation among three specifications with two delayed document objects. The delayed objects are represented by the dark boundary nodes in the first specification with no lines extending. Delays occur until a future phase of the development process requires the information content of those objects. When a given phase does require the information content of a delayed object, the object becomes part of the specification preceding that phase. For example, the document object labeled "1" in Figure 11 is delayed until the second specification level. This implies the second phase of development requires its information content.

The *umbrella document objects* are not directly involved in any transformation, but they affect the design decisions which produce specifications. Their purpose is to assure a level of uniformity and quality among all specifications. Umbrella document objects expressing software quality requirements, for example, are produced in the requirements phase and are represented by doubly inscribed nodes as shown in Figure 11.

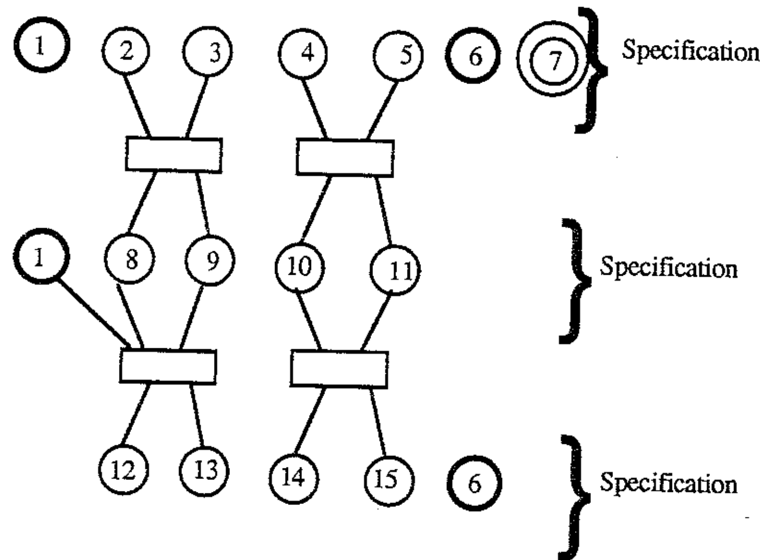

Figure 11. Illustration of Delayed and Umbrella Document Objects in Specifications. (The node (labeled 1) is delayed until the second specification. The node (labeled 7) is an umbrella document object, and is not involved in any transformation but affects design decisions.)

A graph which traces all document objects of the standard set through the seven specification levels is complex and cumbersome; however, a graph that traces document *categories* through the seven levels is reasonable and reveals important information. Minor differences exist between document object and document category trace graphs. With respect to defining a document category trace graph for the standard set, the differences are as follows:

1) The nodes represent document categories instead of document objects. The node level identifies a document category as defined in Section 4.1.2.

2) Dark boundary nodes represent delayed document categories instead of delayed document objects.

3) Doubly inscribed nodes represent umbrella document categories instead of umbrella document objects.

4) Heavy boundary nodes represent document categories produced but never used again in development.

5) Merged rectangles represent the same convergence of information but with respect to document categories rather than objects.

Figure 12 is the complete graph which traces the standard set document categories through the seven specification levels.

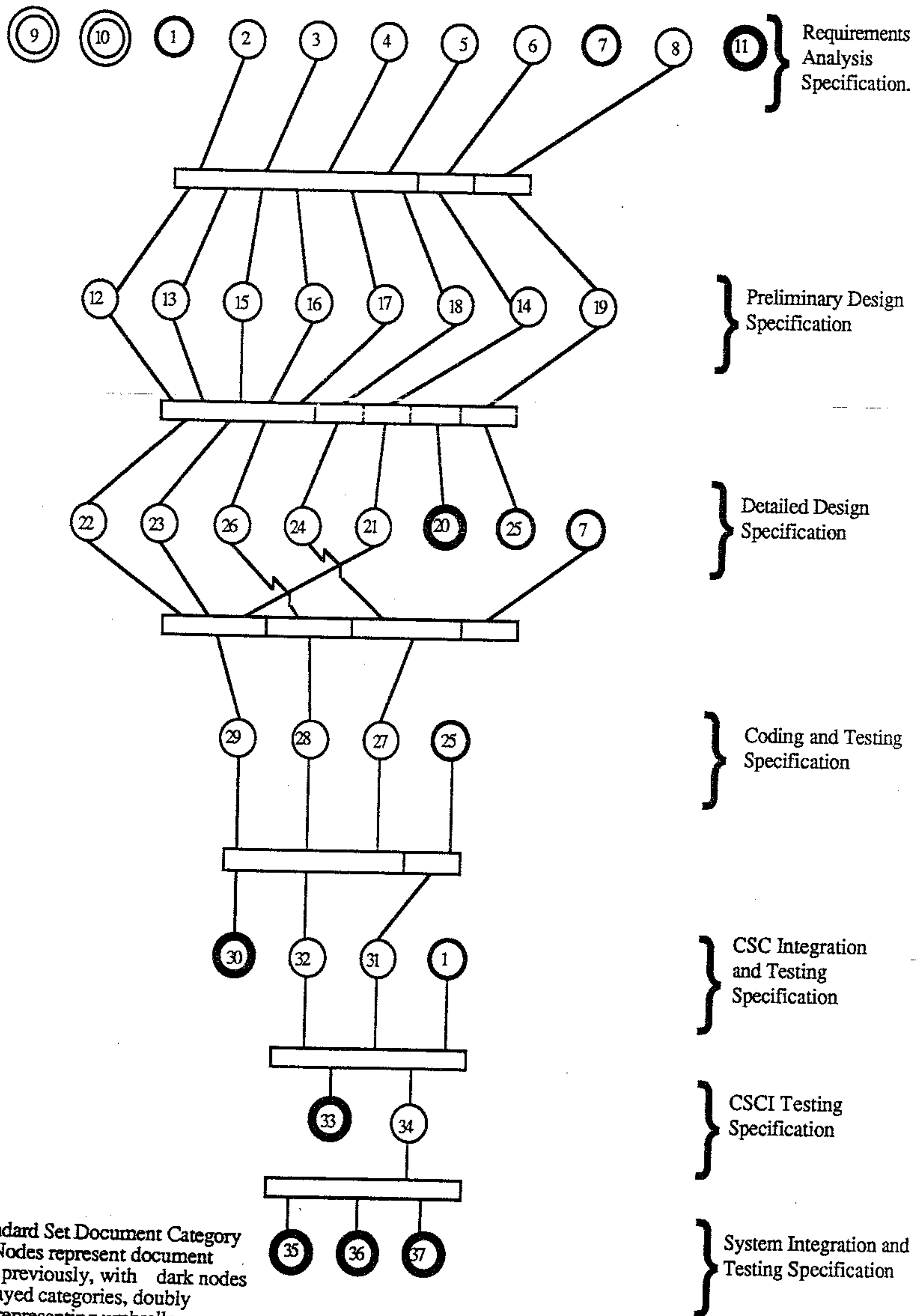

Figure 12. Standard Set Document Category Trace Graph. (Nodes represent document categories listed previously, with dark nodes representing delayed categories, doubly inscribed nodes representing umbrella categories, and extra dark nodes representing categories not further needed in the trace. The edges represent design decisions, and the rectangles represent the convergence of design decisions.)

4.2 The Maintenance Set

Unfortunately, software engineering has been slow to characterize the nature and extent of the maintenance dependency on the software development process. This oversight is manifested in systems that are difficult to maintain. Certainly, maintenance can be viewed as a continuation of the development process. Thus, just as any other development phase, maintenance requires documentation support, and in its own right produces documentation. However, this acknowledgement represents only a start.

The development documentation essential to the maintenance phase is defined as the Maintenance Set. The Maintenance Set is a dynamic collection of document objects divided into two distinct subsets: the Maintenance Kernel and the In-service Set. The Maintenance Kernel is conceivably a static collection of document objects that comprise initial baseline support to maintenance activities; i.e., it is a collection of objects generated throughout the seven phases of software development. The In-service Set is a collection of document objects that are produced during the maintenance phase. Because maintenance is a continuing process, the In-service Set is continuously updated to reflect the evolution of the system.

The remainder of this section describes the definition of the Maintenance Kernel: (1) the ideal, and (2) the realistic set.

4.2.1 The Ideal Maintenance Kernel

The ideal Maintenance Kernel provides the baseline information needs of all maintenance activities. The minimum content of the ideal is defined below (Collofello 86):

> *Requirements/Specification Information* - high and low level system behavior which includes what a system does and how it is done.
>
> *Architectural and Low Level Design Information* - design principles and design decisions, and other information to convey why certain design decisions are made.
>
> *Processing Information* - resource requirements such as hardware and support software.
>
> *Declaration, Control and Data Flow Information* - knowledge of a program's control flow, its invocation hierarchy, data flow, data aliasing, loop termination conditions, entry and exit assertions for procedures, all other syntactic elements which contribute to an understanding of a program's run-time behavior.
>
> *Traceability for Software Life Cycle Products* - traces between specification, design, and source code traces.

Test Environment - diagnostic and regression test cases and procedures.

Anticipated Features for Enhancement - anticipated additional features that are yet to be implemented.

All the above information is included in objects that are members of the previously defined Standard Set. Thus, the ideal Kernel is conceivably defined as the Standard Set.

Realistically, several constraints impair the production of an ideal; the ideal requires the supporting Standard Set to be of very high quality. Such quality is achieved in few project efforts. However, the importance of quality underscores the need to measure document quality so that objects can be excluded from use because of their deficiencies (see Stevens 1988).

In addition to high quality, the Maintenance Kernel must be readily available to maintenance personnel. Quick access to information in a large document collection requires support by an automated system. An automated system is required also to maintain the Kernel and the In-service Set, produced through maintenance activities. Such a system should be cost effective, i.e., it should support only vital, frequently used maintenance support information. Based on requirements of quality assessment and quick access to correct information, the total Standard Set is an unrealistic Kernel.

4.2.2 The Realistic Maintenance Kernel

The realistic Kernel is defined with attention to the extent and quality of source information (the Standard Set) and projected costs. To define the Kernel, the four forms of maintenance are examined and document objects are selected with considerations of the likelihood that maintenance could originate with information at a given level in the ARM. The hypothesized distribution functions are used as guides in the selection. More specifically, the need for a document object to perform a particular form of maintenance establishes a relationship between the Standard Set specification (see Section 4.1.3) containing the object and the form shown below.

Corrective Maintenance (corrects errors in definition of system)

Detailed System Design Specification.
Coding and CSU Testing Specification.
CSC Integration and Testing Specification.
CSCI Testing Specification.
System Integration and Testing Specification.

Adaptive Maintenance (adds functionality to better meet system needs)

Detailed System Design Specification.
Coding and CSU Testing Specification.
CSC Integration and Testing Specification.

Perfective Maintenance (improves system functionality to meet current needs)

Coding and Testing Specification.
CSC Integration and Testing Specification.
CSCI Testing Specification.

Preventive Maintenance (makes system more robust to cope with uncertainties)

CSCI Testing Specification.
System Integration and Testing Specification.

Past research shows that the effort involved in the maintenance activity is not evenly distributed over all four maintenance forms. The uneven distribution of effort contributes to the decision to include a specification in the Kernel. For example, perfective maintenance might absorb 55 percent of the total maintenance effort, and preventive maintenance only ten percent (Schneidewind 1987). If the Kernel cannot support all of the information needs of both types of maintenance, then more space should be allocated to support the needs of perfective maintenance. Fortunately, the information needs of these two forms are similar, so that the loss of information support for the preventive maintenance function may exact little penalty.

A procedure could be defined to place the choice in the form of an assignment problem and extend the selection to document objects rather than categories. The decision process might reflect each object's contribution to each form of maintenance in terms of probability of access, document quality or a combined expression. A total storage capacity would constrain the number of distinct objects in the Kernel. Figure 13 might reflect the division of Maintenance Kernel capacity among the seven categories.

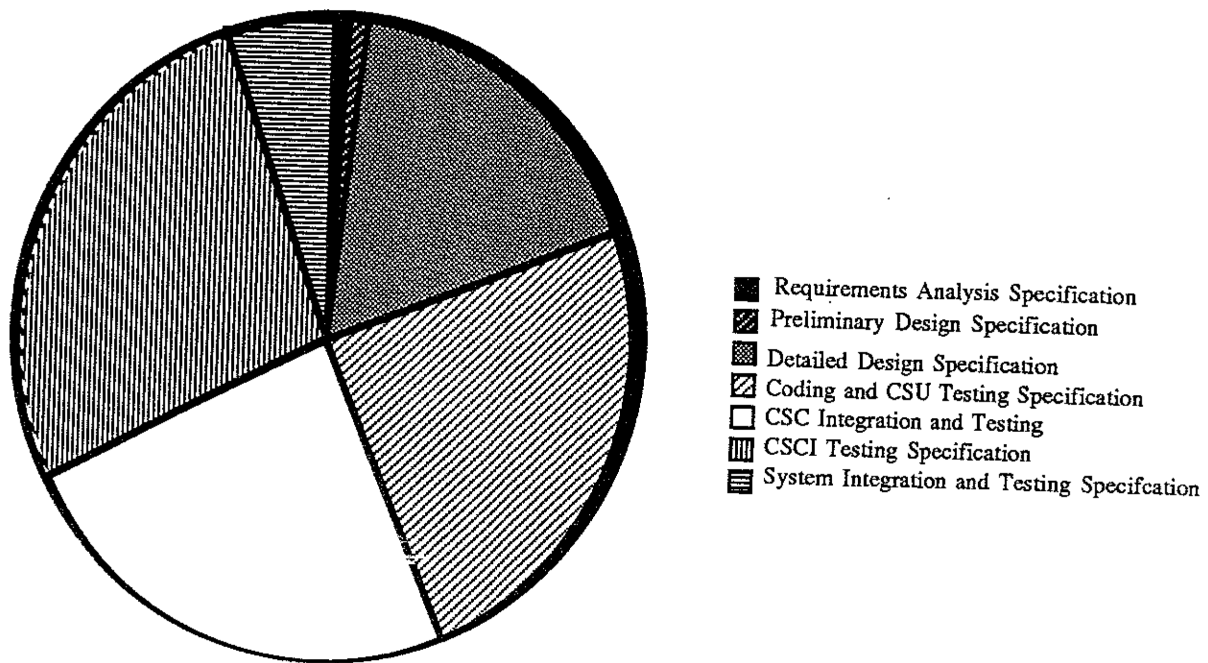

Figure 13. Development Document Composition of the Realistic Kernel

Figure 13 reflects the conclusion that document objects from the Requirement Analysis and the Preliminary Design Specifications are less useful in maintenance than documents from later phases. However, the above derivation of the Kernel set is hypothetical, only to serve as a guide. Other factors also affect the definition of the Kernel such as a special emphasis on a particular maintenance form, the capability to assess document object quality, and the capability to augment inadequate specifications.

5. Documentation in the Maintenance Phase

The prior sections establish the means for explaining the interdependency between development and maintenance phases and the critical role of documentation. A basis for the credibility of the Abstraction Refinement Model (ARM) is found in the categorization of documentation. Clearly, the maintenance function is limited by the quality of the effort performed in the development function. Despite the tendency to view documentation as a development requirement, prior studies have noted the importance of documentation in the maintenance function itself. *Reverse engineering* has long been considered a pragmatic attempt to realize higher quality from a faulty or inadequately documented development process (Rekoff 1985, p. 244). These ideas are expanded in the following paragraphs.

5.1 Persistent Problem: Incomplete Development Information

The ARM clearly characterizes the maintenance function as defining the path from a current realization P to a desired P'. While easy to describe, the actual definition of such a path is extremely difficult. This difficulty stems in part from the loss of information regarding design decisions during development. Typically, errors discovered at one level can not be traced to preceding specifications for correction at that point. The problem can be caused by deficiencies in methodology or the lack of ability to apply a methodology or both. DoD Standard 2167A details the requirements for capturing design decisions in a set of documents governing the acquisition process. Whether this standard exerts significant positive influence remains to be seen.

A second source of difficulty in defining the inverse path is the loss of application domain knowledge. In explanation, as the development process proceeds to successively lower levels in ARM, concerns become dominated by implementation constraints and dictates. The tools used at this level, e.g., program design languages, module interconnection languages, etc., focus on implementation issues. Abbott (1987) notes this deficiency in "knowledge abstraction" and suggests that only in the areas of logic programming and expert systems has the application domain knowledge been successfully captured.

5.2 Unrecognized Problem: Maintenance Documentation

Without a doubt, the maintenance function itself injects errors into the software system. Such errors require further changes, and the cycle continues even in a "fixed" environment (one in which no functionality improvements are attempted).

In attempting to remove errors from the system, the maintenance process requires the creation and use of documents which are themselves representations of the system or parts of the system or descriptions of faults and possible courses of action. Communication among maintenance personnel is effected in the language of these documents, and the education of new hires is accomplished through studying these documents. Deficiencies in maintenance documentation then influence the effectiveness of the maintenance process thereby causing further deficiencies in the software system being maintained.

Thus the maintenance process itself is a source of problems which must be corrected by the same group that creates them. Tools for improving the performance of the maintenance function have been slow to emerge. Recognition of the importance of tools such as ADDS and MicroADDS is inhibited because of a basic lack of understanding of both the development and maintenance processes.

5.3 Pragmatic Solution: Reverse Engineering

For the purposes of the current study, reverse engineering is defined as activities having the objective of assimilating information to enable effective and efficient system maintenance. (See (Rekoff 1985) for a definition more specific to hardware systems.) With respect to ARM, the reverse engineering objective is represented in the identification of the least common abstraction (LCA). Maintenance must also be concerned with those activities in generating the path from the LCA to the target realization (P').

5.3.1 The Ideal Process

The ideal software development process precludes the necessity for reverse engineering; however, such a process would produce error-free software, precluding corrective maintenance as well. (Note that the other forms - adaptive, perfective and preventive - remain as concerns.) Recognizing the fallacy of assuming a process to be close to ideal, reverse engineering is employed to produce the information necessary to define the inverse path from P to the LCA. Again, an ideal view is that such transformations could be made exactly to accomplish the process of "design recovery" (Arango, et. al. 1985).

5.3.2 The Achieveable Process

Even if all the design decisions and application domain knowledge are captured, the inverse transformations cannot be made exactly. ((Arango, et al., 1985) present a contrary view.) This contention is made in the recognition that the forward transformations in the development process are not necessarily invertible, for, if they were, the automatic programming problem would have been solved long ago. An analogy helps to understand this claim.

> Define division for a, b $\in$ N, as $a \div b = \lfloor a/b \rfloor$, where N is the set of natural numbers and $\lfloor\ \rfloor$ is the floor operation.
>
> Multiplication is typically considered the inverse operation of division, for example
>
> $$(4 \div 5) \times 5 = 4,$$
>
> but with division as defined above
>
> $$4 \div 5 = \lfloor 4/5 \rfloor = 0$$
>
> $$\text{so } 4 \div 5 \ \times 5 = 0 \neq 4$$
>
> or the "standard form" of multiplication does not represent an inverse operation.

The loss of information in the application of the floor operator, in the above example, is caused by the requirement that the quotient belong to the natural numbers. The "forcing" of an object into a mold, such as a rational into a natural number or a specification into a design, contributes to a loss of information about the original form of the object. This is characterized functionally by stating that the transformations are onto but not one-to-one, or equivalently (in this use), the function is not invertible.

The lack of a true inverse, or the information necessary to define the inverse path, motivates the attempt to define an approximate inverse. This approximation may construct a different path than that taken during development, in fact the path may not even be in the original domain structure graph. Such a path is called a *shadow path* (or chain).

Figure 14 illustrates the relationship that can exist between the development path and the shadow path constructed in the absence of the necessary information to define the development path inverse. The shadow path is a chain in the semi-lattice which either reaches the least common abstraction or a non-LCA ancestor of the existing (P) and objective (P') realizations. This path is formed by picking the best edges along which the inverse approximation is made based upon available information. Three particular cases of shadow path constructions are shown in Figure 14. Figure 14(a) depicts the unlikely result of actually reaching the LCA through construction of a shadow path in the absence of the inverse path. Figure 14(b) shows a result that leads to a common ancestor (CA) and produces paths not contained in the original development graph. Figure 14(c) shows the result of reaching a common ancestor but achieving the development process through the LCA to achieve P'.

6. The Abstraction Refinement Model in the ADDS Context

The Automated Documentation Description System (ADDS) utilizes a PSL/PSA-based representation to provide four major views of the complex AEGIS software system (McConnell 1987):

(1) system structure,

(2) data structure,

(3) data derivation (data flow), and

(4) system control and dynamics.

Initially developed in the late 1970s, ADDS provides several forms of assistance to maintenance programmers, most prominently in overcoming their most difficult problem: localized understanding (Letovsky and Soloway 1985). However, when neither the program nor the

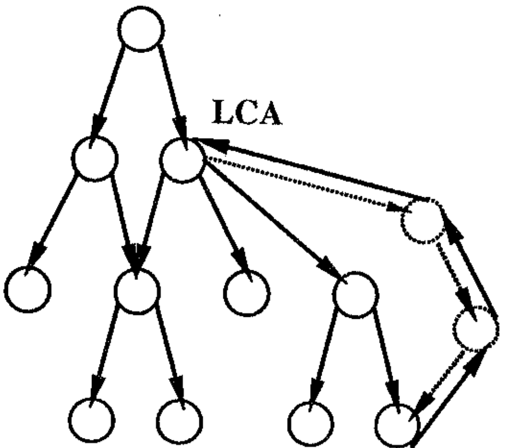

Figure 14(a). Direct Path to the Least Common Abstraction

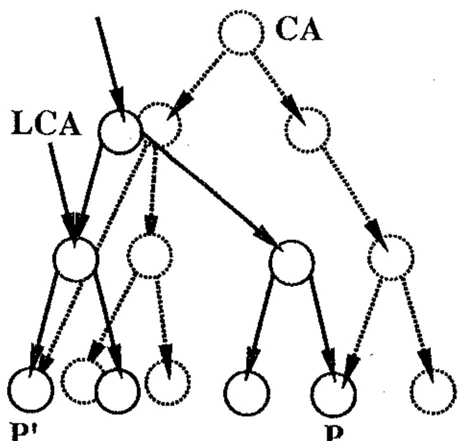

Figure 14(b). Shadow Path Throughout the Maintenance Function

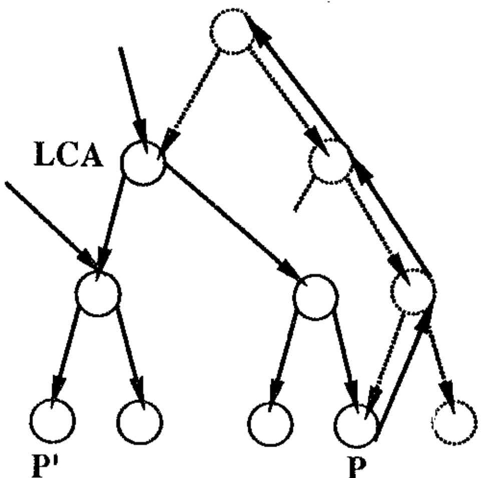

Figure 14(c). Shadow Path to a Common Ancestor, Thence to LCA

Figure 14. The Shadow Path Illustration of the Reverse Engineering Role

documentation informs the programmer that specific pieces of code interact with other pieces of code that are located some distance away, then the strategy of forming a local understanding of the program, which, in turn, can lead to ineffective program modifications.

The later developed MicroADDS utilizes GPTHORS and ZYINDEX commercial software to locate and access information from documents produced in the development process, e.g., Program Performance Specifications, and in the maintenance process (synthesized documents created through PSL/PSA and added utilities). While ADDS originated as a reverse engineering tool, the addition of MicroADDS expands the functionality to meet more comprehensive maintenance needs.

6.1 Documentation Guidelines and Specification Languages

The Abstraction Refinement Model (ARM) assists in providing a definitional answer to a question central to this effort:

> Assuming a predefined set of documentation guidelines and Ada-like specification languages, to what extent can Document Quality Indicators (DQIs) provide a basis for the automatic analysis of documentation?

Documentation standards and guidelines are essential to the development process. Likewise, the adherence to standards and guidelines during development is equally important for its provision of information needed for effective performance of the maintenance function. Predefined structure and format, coupled with a prescribed objective of capturing design decisions, admit the possible construction of the inverse path to the least common abstraction (LCA) that is a goal of reverse engineering. Even in the absence of the inverse path identification, the construction of closer (less removed) shadow paths becomes more feasible.

While the ARM adds nothing in the identification of DQIs, the measurement of document quality takes on added significance in the characterization of the reverse engineering function. Measurements of document quality provide direction as to the need for improvement through document synthesis as well as the potential for realizing improvement. Moreover, the trace of quality indicators in the development process provides guidance as to the comparative difficulty in accomplishing the different forms of maintenance.

The presence of Ada-like specification languages is perceived to add little to, and possibly detract from, maintenance effectiveness. Specifications should admit graduated levels of abstraction. Specification languages should focus on describing "what behavior is sought," not "how that behavior is produced." Derivations from programming languages are unlikely to maintain a purity of focus. In fact, the use of such a derived specification language could create a noticeable gap in the specification refinement progression depicted by the ARM. The presence of

such a gap could inhibit reverse engineering and reduce maintenance effectiveness.

6.2 The Roles of ADDS/MicroADDS in Software Maintenance

The evolution of ADDS/MicroADDS as a maintenance tool is also indicative of the transition from batch processing, mainframe, hardcopy reporting tools to interactive, workstation, screen-oriented tools. ADDS remains as a document synthesis (reverse engineering) tool that progressively is viewed as providing some of the data for MicroADDS, which is perceived to be the primary tool.

6.2.1 The Current and Planned Roles

Although a subset of users prefer to consult hardcopy reports generated by ADDS (Arthur, et al. 1988), MicroADDS is viewed as the emerging tool of preference. This perception is further substantiated in the planned enhancements and extensions shown in Figure 15. The Baseline 2 modifications are intended to improve the information organization, search and retrieval functions. Baseline 3 calls for an improved user interface in addition to enhanced query construction and search capability. Note that added source documents are indicated in each baseline. Baseline 4 focuses on hypertext features and networking, with a reference to enhanced natural language analysis.

MicroADDS DEVELOPMENT PLAN

(OCT 86) (JUNE 88) (DEC 89) (JAN 91)

CATEGORIES	BASELINE 1	BASELINE 2	BASELINE 3	BASELINE 4
USER INTERFACE	GPTHORS MENU SYSTEM HELP SCREENS		WINDOWS/MULTITASKING	WINDOW/PAGING
ZyINDEX FEATURES	STANDARD ZyINDEX FEATURES	MACROS THESAURUS FIELDS MOVE INDEX LIST MOVE FILES CHANGE DRIVE WILDCARD MODE DISK LABEL INFO.	AUTO. LABEL READING COMMAND FILE CAP. RESEARCH ON SUBSET INDEX 'ALL WAYS' INCREASE WILD CARD MODE CAPS. EXCEPTIONAL CHARACTER SEARCH SHOW NUMBER OF HITS PER FILE RECORD DELIMETER (00-FF) CHECK THES. & WILD CARD LISTS AGAINST INDEX LIST	HYPERTEXT
FILE EDITING CAPABILITY		POLYDESK III	IMPROVED WP TO MANAGE DOCUMENTATION CHANGES	
INFORMATION TYPES	PPS, SOURCE, ADDS	DATA DIC. (ADAR.)	DBDD (ADDS)	
GENERAL FEATURES		LIMITED USER INDEXING		NETWORKING PC BASED ADDS ENHANCED NATURAL LANGUAGE ANALYSIS
PROCESSORS SUPPORTED	PC-XT PC-AT	PC-AT PC-386	PC-AT PC-386	PC-386

Figure 15. The MicroADDS Development Plan

6.2.2 Alternative Roles

The MicroADDS Development Plan provides for improved support of access to objects in the Maintenance Document Set, described in Section 4. Although the Plan calls for additional source data (the Program Performance Specifications), the focus is primarily on the "front end:"

(1) improving the user interface,

(2) adding flexibility and capability to the search functions, and

(3) establishing vocabulary control.

These enhancements are needed, particularly the first, but they clearly represent an intent to limit the functionality of the system to primarily that of a reverse engineering tool.

Consideration should be given to redefinition of the ADDS/MicroADDS role. The alternatives suggested below represent role expansions that vary in extent and the attendant costs:

(1) Expansion of the source documents to include a larger number of objects identified in the Maintenance Set. Of particular interest should be the document objects generated in the performance of maintenance (the In-service Set).

(2) Addition of editing capability and access to test tools and data to enable source code modification, testing *and documentation* in an integrated environment. Described in past reports (Arthur, et al. 1988) as making ADDS/MicroADDS a *proactive tool*, this alternative would extend support from reverse engineering to the complete maintenance function as described in the ARM.

(3) Development of an *educational interface* to complement the current *reference interface* and the proposed enhancements to it. This interface would utilize existing data, and objects to be added, in exposing the user to:

   (i) the organization and components of the AEGIS Combat System,

   (ii) the architecture of the AEGIS Computer System,

   (iii) the software development process, and its relationship to software maintenance and the ADDS/MicroADDS system,

   (iv) the AEGIS software maintenance process, and

   (v) the role and proper use of ADDS/MicroADDS in software maintenance.

Admittedly, each of these alternatives represents a major expansion of the current role. However, the perceived needs and potential benefits suggest that consideration be given to them in the intermediate and longer term planning.

6.3 Improving the AEGIS Maintenance Function

The Abstraction Refinement Model and the document set categorization and correspondent mapping in ARM emphasize the importance to *maintenance* of documentation in the *development* process. That importance is manifested in:

(1) the extent to which development documents are accessible by maintenance personnel and the access costs, and

(2) the quality of the development documents.

The effectiveness and efficiency of software maintenance is highly dependent on the development documentation, which defines the Maintenance Kernel (see Section 4.2).

As the maintenance phase progresses, the documents produced in the performance of maintenance assume proportionately greater importance. However, adaptive and perfective maintenance functions in particular make access to the development documentation a recurring requirement. Therefore, *a major factor in maintenance effectiveness and efficiency is the quality of development documents*. The ability to influence or dictate development document quality should be of high priority for managers of software maintenance. Even the reverse engineering function is crucially dependent on development documentation, for the quality of a synthesized document is tied to that of the original(s) - "one cannot fashion a silk purse from a sow's ear," (Jonathan Swift, *Polite Conversation*, 1733).

7. Concluding Recommendations

The initiation of ADDS in the late 1970s represents a farsighted approach to coping with the AEGIS software maintenance responsibilities. The addition of MicroADDS parallels the technology transition from batch, mainframe, hardcopy support to interactive, workstation, screen-oriented. This project has addressed the prospects for supporting AEGIS software maintenance needs influenced by the pull of emerging technologies. Frankly, the effort has been forced to address: (1) the lack of understanding of software maintenance, (2) the role of reverse engineering within maintenance, and (3) the relationship to the software development process and development documentation.

The construction of ARM has served to clarify the understanding of all three issues noted above. Benefitted by this improved understanding, we offer the following recommendations:

(1) Adopt the characterization of the four forms of maintenance so as to explain to management the breadth and extent of activities that appear to concern some NSWC managers that the activities are viewed as "bug removal."

(2) Use the ARM to explain the role of reverse engineering, the software maintenance function, and the maintenance - development relationships to management, development personnel, and trainees.

(3) Review the availability of development documents, e.g., Interface Design Specifications; and using the Maintenance Set defined in Section 4, identify candidates for the Maintenance Kernel.

(4) Using the structure and characterization of document quality from a prior report (Stevens, et al. 1988) and the supporting M.S. Thesis (Stevens 1988), examine the candidates from (3) above. Based on availability and cost considerations, define the Maintenance Kernel.

(5) Review the potential for improving documents produced in the development process. Take actions wherever possible to effect improvements since the quality of development documentation is inherently a limiting factor on the quality of synthesized documents and the effectiveness and efficiency of software maintenance.

(6) Review the software maintenance process to assess the potential for improving documentation produced in maintenance. Present improvement proposals to management showing the dynamic nature of the Maintenance Set and the impact of the recurring process on effectiveness and efficiency. Note the transitional nature of agency responsibility *and control* (from RCA to NSWC).

(7) Consider the three alternative roles identified for ADDS/MicroADDS and assess the cost/benefit tradeoffs of role redefinition. Revise the ADDS/MicroADDS Development Plan as warranted by the assessment.

(8) Re-examine the ADDS/MicroADDS Development Plan, without regard to role redefinition, and consider the potential improvements in the reverse engineering tool by (a) deriving more formal representations from development documents, (b) extracting design decision information through a chronological trace, e.g., high-level design → detailed design → PPS → program.

(9) Do not pursue the use of an Ada-derived program design language except as a component in a vertical specification trace. Its utility for maintenance support is judged to be limited.

References

Abbott, R.J. (1987) "Knowledge Abstraction," *Communications of the ACM, 30*, 8 (August), 664-671.

Abbott, R.J. (1986) An Integrated Approach to Software Development, John Wiley & Sons, Inc., 1986.

Arango, G., I. Baxter, and P. Freeman (1985) "Maintenance and Porting of Software by Design Recovery," In: *Proceedings of the Conference on Software Maintenance - 1985*, IEEE Computer Society Press, 42-49.

Arthur, J.D., R.E. Nance, and K.T. Stevens (1988) "Automated Design Description System (From the User's Perspective)," Sustems Research Center, Briefing at NSWC, 4 February.

Belady, L.A. and M.M. Lehman (1976) "A Model of Large Program Development," *IBM Systems Journal*, v. 15, no.3, 225-252.

Benington, H.D. (1983) "Production of Large Computer Programs," In: *Proceedings of ONR Symposium on Advanced Programming Methods for Digital Computers* (Washington, D.C., June 28-29). Office of Naval Research Symposium Report ACR-15. Also available in *Annals of the History of Computing 5*, 4 (Oct.), 350-361.

Boehm, B.W. (1976) "Software Engineering," *IEEE Transactions on Computers C-25*, 12 (Dec.), 1226-1241.

Boehm, B.W. (1986) "A Spiral Model of Software Development and Enhancement," *ACM Software Engineering Notes 11*, 4 (Aug.), 14-24.

Collofello, James S. (1986) "An Analysis of the Technical Information Necessary to Perform Effective Software Maintenance," *Fifth Annual International Phoenix Conference on Computers and Communications* (Scottsdale, AZ. 26-28 March 1986), Computer Society Press, pp. 420-424.

Giddings, R.V.(1984) "Accommodating Uncertainty in Software Design," *Communications of the ACM 27*, 5 (May), 428-343.

Hoare, C.A.R. (1985) "The Mathematics of Programming," *Foundations of Software Technology and Theoretical Computer Science*, LNCS #206, (ed. S.N. Maheshwari), Springer-Verlag, New York, New York, 1-18.

Hoare, C.A.R., I.J. Hayes, H. Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin (1987) "Laws of Programming," *Communications of the ACM, 30*, 8 (Aug.), 672-686.

Jenkins, A. Milton (1983) "Prototyping: A Methodology for the Design and Development of Application Systems," Discussion Paper #227, School of Business, Indiana University, Bloomington, Indiana.

Lehman, M.M. ( ) "Programs, Life Cycles, and Laws of Software Evolution," In: *Proceedings of the IEEE 68*, 9 (Sept.), 1060-1076.

Liu, C.C. (1984) "A Look at Software Maintenance", *Datamation*, 22,11, (Nov. 1976), 51-5. Little, Brown & Co., Boston, MA.

McConnell, David E. (1987) "Tools for Supporting P3I: Automated Design Description System (ADDS)," Seminar Presentation Handout, Virginia Tech.

Neighbors, J.M.(1980) "Software Constructions Using Components," TR-160, University of California, Irvine.

Neighbors, J.M. (1984) "The Draco Approach to Constructing Software from Reusable Components," *IEEE Transactions on Software Engineering, SE-10*, 5, (Sept.), 564-573.

Pressman, Roger S. (1986) *Software Engineering: A Practical Approach*, McGraw-Hill Book Company.

Rekoff, M.G. (1985) "On Reverse Engineering," IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-15, 2, (Mar.-Apr.), 244-252.

Stevens, Todd K. (1988) "A Taxonomy for the Evaluation of Computer Documentation", Thesis for Computer Science Department of Virginia Tech.

Stevens, Todd K., J.D. Arthur, and R.E. Nance (1988) "A Taxonomy for the Evaluation of Computer Documentation," SRC-88-008, Systems Research Center, Virginia Tech, 15 June.

Stevens, K.T., J.D. Arthur, and R.E. Nance (1988) "Prospects for Automated Documentation Supporting Software Quality Assurance: A Taxonomy for the Evaluation of Documentation," Systems Research Center, Virginia Tech, Presentation of 31 May.

Appendix A.

The following document objects were determined from a review of five documentation standards. See Appendix B for the documentation standards used.

Allocations, CSCI.

a) Functional allocation tables, CSCI to TLCSC(1).
b) Interface allocation tables, CSCI to TLCSC(1).
c) Memory and processing time tables,CSCI to TLCSC(1)
d) Requirments allocation tables, CSCI to TLCSC(1).

Allocations, TLCSC.

a) Functional allocation tables, TLCSC to LLCSC(1).
b) Interface allocation tables, TLCSC to LLCSC(1).
c) Performance allocation tables, TLCSC to LLCSC(1).

Architecture, CSCI.

a) Diagrams, top-level architecture(1,5).
b) Relationships, TLCSCs and LLCSCs(1).
c) TLCSCs, Identification and Purpose (1,3,5).

Data Bases/Files Specifications(1,2,3,4,5).

a) Data Base Access Protection (1,2,5).
b) Data Base Access Protocols(1,2,4,5).
c) Data Base Amendment Method(1,2,5).
d) Data Base Construction Method(1,2,5).
e) Data Base File Origin(4).
f) Data Base File Size(4).
g) Data Base File Type(4).
h) Data Base Files, External data (3).
i) Data Base Files, Input data(3).
j) Data Base Files, Library data(3).
k) Data Base Relational Structure (1,2).
l) Data Base Storage media (2,4).
m) Data Base Structure Diagram(1).

Design, Detailed CSCI interface.

a) Item fields, Identification and purpose of variable (1,2).
b) Item fields, Identification and purpose of constant (1,2).
c) Item fields, Value ranges(1,2).

Design, LLCSC.

a) Design characteristics(1).
b) Diagrams, data flow and control flow (1,2,3,4).
c) Input data, Specifications(1,2,4).
d) Limitations(1,2,3,4,5).
e) Local Data Bases/Files, Specifications(1).
f) LLCSCs, Description and purpose(1,4).
g) Output data, Specifications.(1,2,4)
h) Process Control(1).
i) Resource utilization(1).
j) States Tables(1)
k) Tables, Local data definition(1).

Design, LLCSU.

a) Design characteristics(1).
b) Diagrams, data or control flow (1,2,3).
c) Input data, Specifications(1,2,4).
d) Limitations(1,2,3,4,5).
e) Local Data Bases/Files, Specifications(1).
f) LLCSCs, Descripition and purpose(1,4).
g) Output data, Specifictions(1,2,4).
h) Process Control(1).
i) Resource Utilization(1).
j) States Tables(1).
k) Tables, Local data definition(1).

Design, Preliminary CSCI interface.

a) Interface Item Summary (1).
b) Interface Item Formats(1).
c) Tables, Interface data summary and interface format(1,3).

Design, Top Level.

a) Algorithms, Identification and Purpose (1,3,4).
b) Error Handling Specifications(1,4).
c) Error Recovery Procedures (2).
d) Functional Block/Diagrams(1).
e) Input data, Specifications(1,2,4).
f) Interrupts, Identification and Purpose.(1,2).
g) Interrupts, Specifications(1,2).
h) Local data, Specification(1,2).
i) Local data bases/Files, Specifications(1).
j) Output data, Specifications(1,2,3).
k) Special Control Specifications(1).

Functional Control/Data Flow Information.

a) Tables, State/TLCSC (1,2,4).
b) Diagrams, Control flow and TLCSC dataflow(1,2,3,5).

Global Information.

a) Data bases/Files, Access procedures.(1).
b) Data bases/Files, Identification and Purpose (1,2,5).
c) Data bases/Files, Structure(1,2,3,4,5).
d) Format, Global data(1).
e) Global Specifications(1).
f) Tables, TLCSC/Global Data relations(1).

Reports, Traceablility.

a) Tables, Requirements trace(1).

Requirements, Adaptation.

a) Environment, System (1,2,5).
b) Capacities, System (1).
c) Tables, System adaptation(1).

Requirements, Design.

a) Allocation Tables, Memory and processing times (1,3).
b) Constraints, Design (1,2).
c) Standards, Design (1,2).

Requirements, Functional Input/Output.

a) Error detection, Function Input (2).
b) Functions, Description and purpose (1,2,3,4,5).
c) Input data Frequency (1,2).
d) Input data Sources (1).
e) Tables, Input data (1,2).
f) Tables, Output data (1,2).
g) Tables, State/Functional (1).

Requirements, Functional Processing/Performance.

a) Allocation Tables, Performance requirements to functions (1).
b) Events, Timing and Sequencing (1,2).
c) Error detection (1,2,3).
d) Error recovery (1,2,3).
e) Operations, Description and purpose(1).
f) Requirements, Logical algorithms and equations (1,3,4).
g) Restrictions or limitations of functions(1).
h) Tables, Functional parameter effect/operations (1).

Requirements, Interface.

a) Definitions, Interface (1,2,4).
b) Diagrams, Block (1).
c) Tables, Iterface Identification and summary tables (1).

Requirements, Programming.

a) Compiler/Assemblers (1).
b) Languages; Programming (1,3,4,5).
c) Operating Systems (3,4)
d) Standards; Programming (1,2,4).

Requirements, Qualification.

a) Demonstration methods(1).
b) Qualfication test, Additional information(1).
c) Qualification test, Analysis(1).
d) Qualification test, Methods(1).
e) Qualification test, Visual inspections(1).
f) Special Qualification Requirements(1).

Requirements, Quality.

a) Additional Quality(1,4,5).
b) Correctness(1,2).
c) Data Security(4,5).
d) Efficiency(1,2).
e) Flexibility(1,2).
f) Integrity(1,2).
g) Interoperability(1,2).
h) Maintainability(1,2).
i) Portability(1,2).
j) Reliability(1,2).
k) Reuseability(1,2).
l) Testability(1,2).
m) Usability(1,2).

Requirements, Supportive.

a) Auxillary programs (3).
b) Devices, Phyical and Logical(1,2,3).
c) Equipment(1,2).
d) Facilities (1,2,4).
e) Hardware configurations(2,3).
f) Personnel, training requirements(1).
g) Remote Terminals, Purpose and user instruction (4,5).
h) Software support (1,4).

Software testing, CSC test cases.

a) Results, Expected(1,3,4,5).
b) Test Data (1,3,4,5).

Software testing, CSC test procedures (1,2).

Software testing, CSC integration test results (1,2).

Software Testing, CSC test requirements.

a) Factors, Testing(2).
b) Methodologies and Tools, Testing(1,2,4).

Software Testing, CSU test procedures and test results (1,2).

Software testing, CSU test requirements and test cases.

a) Acceptance criteria (2,4).
b) Results, Expected(1,3,4,5).
c) Mehodologies and Tools, Testing(1,2,3,4).
d) Testing, Considerations (2).
e) Test Data (1,3,4,5).

Software testing, CSCI Formal qualification tests.

a) Test Requirements, Formal(1).
b) Test Requirements, Informal(1).

Software testing, CSCI Formal qualification test cases.

a) Description and purpose of formal tests(1,2).
b) Intialization requirements(1,2).
c) Input data specification(1,2).
d) Results, Intermediate(1,2).
e) Results, expected (1,2).
f) Results, evaluation criteria(1,2).
g) Testing assumptions and constraints(1,2).

Software testing, CSCI Formal qualification test procedures.

a) Procedures, Analysis (1).
b) Procedures, Set-Up (1).
c) Results, Expected (1).

Software testing, CSCI Formal qualification test results(1).

Source code, CSU (1,2,3,4,5).

System testing, Integration test plans.

System testing, Integration test results.

a) Test Plans(1).
b) Test Data(1)
c) Test Procedures(1).

Updated Source code and documentation (1).

Appendix B.

The following documentation standards were used to define the documentation sets.

DoD-STD-2167A, **Defense System Software Development**, United States Department of Defense, June 4, 1985.

The Institute for Electrical Engineers, **Guidelines for the Documentation of Software in Industrial Computer Systems**, The Institute of Electrical Engineers, 1985.

Secretariat American Nuclear Society, **American National Standard Guidelines for the Documentation of Digital Computer Programs**, ANS-10.3/N413-1974.

American National Standards Committee X3, Information Processing Systems, **Guide for Technical Documentation of Computer Projects: Information Processing Systems.** Technical Report X3TR-6-82, Computer and Business Manufactures Association, 1982.

Poshman, A. W., **Standards and Procedures for System Documentation**, American Management Associations, 1984.

Appendix C.

The following are definitions of key words used which originated from DoD-STD-2167A.

Computer Software Component (CSC) - A distinct part of a computer software configuration item (CSCI). CSCs may be further decomposed into other CSCs and Computer Software Units (CSUs).

Computer Software Configuration Item (CSCI) - A configuration item of computer software.

Computer Software Unit (CSU) - An element specified in the design of a Computer Software Component (CSC) that is separately testa.

Formal Qualification Testing (FQT) - A process that allows the contracting agency to determine whether a configuration item complies with the allocated requirements for that item.

System/Segment Specification (SSS) - System level requirements specifications.

Prime Item Development Specification (PIDS) - System level requirements specifications.

Critical Item Development Specification (CIDS) - System level requirements specifications.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: Unclassified

1b. RESTRICTIVE MARKINGS:

2a. SECURITY CLASSIFICATION AUTHORITY:

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE:

3. DISTRIBUTION / AVAILABILITY OF REPORT: Unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S): Systems Research Center, Blacksburg, Virginia 24061 — SRC-89-001

5. MONITORING ORGANIZATION REPORT NUMBER(S):

6a. NAME OF PERFORMING ORGANIZATION: Systems Research Center

6b. OFFICE SYMBOL *(If applicable)*:

6c. ADDRESS *(City, State, and ZIP Code)*:
320 Femoyer Hall
Virginia Tech
Blacksburg, Virginia 24061

7a. NAME OF MONITORING ORGANIZATION: Naval Surface Warfare Center, Dahlgren, Virginia 22448

7b. ADDRESS *(City, State, and ZIP Code)*:

8a. NAME OF FUNDING / SPONSORING ORGANIZATION: Naval Surface Warfare Center

8b. OFFICE SYMBOL *(If applicable)*:

8c. ADDRESS *(City, State, and ZIP Code)*: Dahlgren, Virginia 22448

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER:

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*: Documentation Production Under Next Generation Technologies, Final Report

12. PERSONAL AUTHOR(S): Richard E. Nance, Benjamin J. Keller, Dave Boldery

13a. TYPE OF REPORT: Final

13b. TIME COVERED: FROM 6/15/88 TO 10/15/88

14. DATE OF REPORT *(Year, Month, Day)*: 15 January 1989

15. PAGE COUNT: 49

16. SUPPLEMENTARY NOTATION:

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*:

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

This report describes the development of the abstraction refinement model as a basis for linking the development and maintenance tasks in software systems. Documentation is critical in both efforts, and the reliance on development documentation during maintenance is characterized by the model and through a characterization of the development documentation requirement stipulated under DoD-STD-2167A. The abstraction refinement model enables a coherent characterization of the reverse engineering requirements generally caused by a faulty or inadequately documented development process. Within the context of the model, the automated documentation design system (ADDS) is characterized, and the system is evaluated with regard to its current capabilities versus future potential. A set of recommendations regarding ADDS concludes this report.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION: Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL:

22b. TELEPHONE *(Include Area Code)*:

22c. OFFICE SYMBOL:

DD Form 1473, JUN 86 — *Previous editions are obsolete.*