

Air Force Institute of Technology

AFIT Scholar

Theses and Dissertations

Student Graduate Works

12-1996

An Approach to Evaluate Software Effectiveness

Timothy J. Schalick

Follow this and additional works at: <https://scholar.afit.edu/etd>



Part of the [Software Engineering Commons](#)

Recommended Citation

Schalick, Timothy J., "An Approach to Evaluate Software Effectiveness" (1996). *Theses and Dissertations*. 5880.

<https://scholar.afit.edu/etd/5880>

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact AFIT.ENWL.Repository@us.af.mil.

AFIT/GCS/ENG/96D-24

DTIC QUALITY INSPECTED 2

AN APPROACH
TO EVALUATE SOFTWARE EFFECTIVENESS
THESIS

Timothy J. Schalick, Captain, USAF

AFIT/GCS/ENG/96D-24

19970409 039

Approved for public release; distribution unlimited

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U. S. Government.

AN APPROACH
TO EVALUATE SOFTWARE EFFECTIVENESS

THESIS

Presented to the Faculty of the Graduate 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 in Computer Systems

Timothy J. Schalick, B.S.

Captain, USAF

December 1996

Approved for public release; distribution unlimited

Acknowledgments

I'd like to thank the Lord, from whom all good things come. I am deeply indebted to my thesis advisor, Major Mark Kanko, without whose patience, persistence and constructive criticism, I would have never been able to complete this document. I'd also like to thank my committee members, Dr Thomas Hartrum and Dr Henry Potoczny, for their insightful comments and suggestions. A tip of the cap goes to my classmate, Captain Marshall "messy" Messamore, a good friend and a great Air Force officer, for his encouragement and sense of humor, but most of all, for his friendship.

Mr Jeff Wiltse and Captain Brian Hermann played a key role in the success of this research as members of the Software Analysis Team at the Air Force Operational Test and Evaluation Center (AFOTEC), the sponsor of this research effort. Mr Nelson Estes, of the C-17 System Program Office here at Wright-Patterson AFB, was instrumental in providing me with data for the demonstration portion of my research.

The individuals I'd like to thank most are my family: Mary, my wife, who has always been there; Sarah, my daughter, who got here a little over a year ago amidst the most tumultuous of circumstances; and Stolie (Stolichnaya) The Wonder Dog, who slept at my feet while I wrote most of this document. Without the monumental efforts of my family surrounding me with love and support, I would not be who I am, or where I am, today. I love you more than you'll ever know, and I dedicate this work to you.

Timothy J. Schalick

Table of Contents

	Page
Acknowledgments	ii
List of Figures.	x
List of Tables	xii
Abstract	xiii
1. Introduction	1
1.1 Background	1
1.2 Problem Statement	3
1.3 Research Objectives	4
1.4 Research Questions	5
1.5 Scope	5
1.6 Assumptions	6
2. Literature Review	8
2.1 Introduction	8
2.2 Literature Directly Addressing Software Effectiveness	8
2.2.1 PRC, Inc. Report on Software Effectiveness.	9
2.2.2 Stanko's Thesis on Software Reliability.	12
2.2.3 Zane's Study of Educational Software Effectiveness	13
2.2.4 Summary	13
2.3 Industry Perspective on Software Effectiveness	14
2.3.1 IBM.	14
2.3.2 Boeing	15
2.3.3 Microsoft	15
2.3.4 Lockheed Martin	15
2.3.5 Summary	15

	Page
2.4 Military Perspective on Software Effectiveness	16
2.4.1 Department of Defense	16
2.4.2 Air Force Test and Evaluation	16
2.4.3 Air Force Developmental Test and Evaluation	16
2.4.4 Air Force Operational Test and Evaluation.	17
2.5 Summary and Conclusion.	18
3. Research Methodology.	20
3.1 Introduction	20
3.2 Original Research Plan.	21
3.3 Difficulties Encountered During Research	22
3.3.1 Lack of Previous Research.	22
3.3.2 Problems with the PRC Report.	23
3.3.3 No References in Industry	24
3.3.4 Summary	24
3.4 Dead Ends for Software Effectiveness Evaluation Methods	25
3.4.1 User's Perspective	25
3.4.2 Static Set of Software Attributes.	25
3.4.3 Dynamic Set of Software Attributes	26
3.4.4 Dynamic/Static Sets of Metrics	26
3.4.5 Effectiveness Viewed as Another Term for Quality	27
3.4.6 Summary	28
3.5 Modified Research Plan	28
3.6 Preliminary Definition of Software Effectiveness.	29
3.7 AFOTEC's Definition of System Effectiveness	30
3.8 Effectiveness Definitions in Other Fields of Study	31
3.8.1 System Effectiveness	31
3.8.2 Information Systems Effectiveness	33
3.8.3 Effectiveness of Teaching Methods.	33
3.8.4 Effectiveness of Strategies for Hardware Reconfiguration	34
3.8.5 Traffic Control Effectiveness	34
3.8.6 Shielding Effectiveness in Materials	34

	Page
3.8.7 Summary	35
3.9 Performance-Based Software Attributes	35
3.9.1 Software Reliability	35
3.9.2 Software Quality	36
3.10 Working Definition of Software Effectiveness	38
3.11 Software Activities Related to Software Effectiveness	39
3.11.1 Testing	40
3.11.2 Verification and Validation	41
3.11.3 Requirements Tracing	42
3.12 Summary	43
3.13 Software Effectiveness Evaluation Approach.	44
4. Evaluating Software Effectiveness through Requirements Traceability	46
4.1 Introduction	46
4.2 Definitions	48
4.2.1 Requirements	48
4.2.2 Design	50
4.2.3 Code	51
4.2.4 Tests	51
4.2.5 Artifact.	52
4.2.6 Element	52
4.2.7 Trace	53
4.2.8 Structure.	53
4.3 Assumptions	54
4.4 Unique Identification of Elements	57
4.4.1 ORD Elements.	59
4.4.2 System Requirements	60
4.4.3 Software Requirements.	60
4.4.4 Design Elements	60
4.4.5 Code Elements.	60
4.4.6 Tests	61

	Page
4.5 Structures Used for Traceability	61
4.5.1 ORD Structure.	61
4.5.2 System Requirements Structure	62
4.5.3 Software Requirements Structure	62
4.5.4 High-Level Design Structure	62
4.5.5 Low-Level Design Structure	63
4.5.6 Validation Tests Structure	63
4.5.7 Code Structure.	63
4.5.8 Unit Tests Structure	64
4.5.9 Summary	64
4.6 Trace Descriptions	64
4.6.1 Introduction.	64
4.6.2 ORD to System Requirements	66
4.6.3 System Requirements to Software Requirements.	66
4.6.4 Software Requirements to High-Level Design	67
4.6.5 High-Level Design to Low-Level Design.	67
4.6.6 Low-Level Design to Code	68
4.6.7 Software Requirements to Validation Tests	69
4.6.8 Code to Unit Tests	69
4.6.9 Summary	70
4.7 Implementing Traces in a Database	72
4.7.1 Introduction.	72
4.7.2 General Forms for Database Entries	73
4.8 Database Operations	80
4.8.1 Introduction.	80
4.8.2 Database Initialization	81
4.8.3 Data Entry.	81
4.8.4 Error Checking	85
4.8.5 Information Retrieval	86
4.8.6 Creation of Traceability Matrices	91
4.8.7 Summary	91
4.9 Calculation of Software Effectiveness	93
4.9.1 Five Components of Software Effectiveness Evaluation.	93
4.9.2 Combining Effectiveness Components for Overall Evaluation . .	93
4.10 Summary of Software Effectiveness Approach.	95
5. Demonstration of the Software Effectiveness Traceability Approach.	97

	Page
5.1 Introduction	97
5.2 The Decision Process to Demonstrate SETA	98
5.2.1 Validation of SETA with Actual Data	98
5.2.2 Validation of SETA with Test Data.	99
5.2.3 Demonstration of SETA with Actual Data	100
5.3 The Search for Traceability Data	100
5.4 Background Information on C-17 Avionics	101
5.4.1 Core Integrated Processor	101
5.4.2 Operating System Utilities.	101
5.4.3 1553 Data Bus	102
5.5 Overview of MIL-STD-1553	103
5.5.1 Background.	103
5.5.2 1553 Data Bus Modes.	103
5.5.3 Software for the 1553 Data Bus	104
5.6 Software Development Terminology Used at the C-17 SPO.	105
5.6.1 Prime Item Development Specification	105
5.6.2 Computer Program Development Specification.	105
5.6.3 Computer Program Product Specification	105
5.6.4 Computer Program Test Procedures	105
5.6.5 Unit Tests	106
5.6.6 Summary	106
5.7 Translation of Element Identification Methods	107
5.7.1 Introduction.	107
5.7.2 Element Identification Method for C-17 SPO Artifacts	107
5.7.3 Translation of Identification Method to SETA	108
5.8 Demonstration of Traceability of 1553 Data Bus Software.	108
5.8.1 Introduction.	108
5.8.2 ORD to System Requirements	109
5.8.3 System Requirements to Software Requirements.	110
5.8.4 Software Requirements to High-Level Design	113
5.8.5 High-Level Design to Low-Level Design	122
5.8.6 Low-Level Design to Code	124
5.8.7 Software Requirements to Validation Tests	131

	Page
5.9 Effectiveness of 1553 Data Bus Software	134
5.9.1 Introduction.	134
5.9.2 Effectiveness Components.	134
5.9.3 Significance of Overall Effectiveness Value.	135
5.10 Summary	135
5.10.1 Introduction.	135
5.10.2 Traceability Must Start Early	136
5.10.3 Additional Benefits of Traceability	136
5.10.4 Limitations Revealed by the Demonstration	137
6. Conclusions and Recommendations	138
6.1 Introduction	138
6.2 Research Summary	138
6.2.1 Overview	138
6.2.2 Limitations of SETA	140
6.2.3 Limitations of Database Implementation	142
6.3 Addressing Objectives and Questions from Previous Chapters	145
6.3.1 Meeting Research Objectives.	145
6.3.2 Answering Research Questions	146
6.4 Additional Benefits of SETA	148
6.4.1 Introduction.	148
6.4.2 Additional Benefits of Database Implementation.	148
6.4.3 Additional Benefits of Tracing Requirements	149
6.5 Practicality of Implementing SETA	153
6.5.1 Requirements Traceability in AFOTEC Documents.	154
6.5.2 Requirements Traceability in MIL-STD-498	155
6.5.3 Requirements Traceability in DoDR 5000.2-R	155
6.6 Recommendations for Future Research.	157
6.7 Final Comments	158
6.7.1 Importance of the Research	158
6.7.2 Software Effectiveness; What's in a Name?	159
Bibliography.	161

	Page
Vita.	167
Appendix A: Acronyms and Abbreviations.	168
Appendix B: Selected Code From 1553 Data Bus Software.	171

List of Figures

Figure	Page
1. Abstract View of Degrees of Traceability	47
2. Example Structures with Traces	54
3. Example of Element Decomposition and Identification Method	58
4. Three Purposes of the Element Identification Method	59
5. Example Section of ORD Structure	62
6. Example of ORD to System Requirements Traces	66
7. Example of System Requirements to Software Requirements Traces.	67
8. Example of Software Requirements to HLD Traces	68
9. Example of HLD to LLD Traces	68
10. Example LLD to Code Traces	69
11. Example of Software Requirements to Validation Tests Traces	70
12. Example of Code to Unit Tests Traces	70
13. Format of Database Entry for Element Decomposition	74
14. Example Database Entries Containing Decomposition Information.	75
15. Format of Database Entry for Trace Information	75
16. Example Database Section Containing Trace Information	78
17. Example Database Section	79
18. Logical Arrangement of Database Sections	81
19. Example Section of Database after Initialization.	82
20. Data Entry Example.	83
21. Database Information Used in Software Effectiveness Evaluation.	87
22. Example Database for Degree of Traceability Calculation	89

	Page
23. Example Portion of the Requirements to HLD Traceability Matrix	92
24. Effectiveness Components for an Example Software System	94
25. System Diagram for C-17 Core Integrated Processor Architecture.	102
26. ORD to System Requirements Database Entries.	110
27. System Requirements to Software Requirements Database Entries	113
28. Partial Decomposition and Description of RO - 3.2.2	114
29. Partial Decomposition and Description of RO - 3.2.8	115
30. Database Entries for Partial Decomposition of RO - 3.2.2	118
31. Database Entries of Partial Decomposition of RO - 3.2.8.	119
32. (Continued)	120
33. Software Requirements to HLD Database Entries.	121
34. HLD to LLD Database Entries.	125
35. LLD to Code Database Entries for 1553 Driver	128
36. LLD to Code Database Entries for POBIT Utility	129
37. LLD to Code Database Entries for MIBIT Utility	129
38. LLD to Code Database Entries for Common_BIT Utility.	130
39. Database Entries for Software Requirements to Validation Tests	132
40. Example Decomposition Database Entries	143
41. Possible Alternate Database Design	144
42. Example of HLD to Requirements Backtraces	150
43. Traces from Example of HLD to Requirements Backtraces	151
44. Example of HLD to Requirements Traces with Requirement Change	153

List of Tables

Table	Page
1. Correspondence Between Software Development Artifacts	106

Abstract

The Air Force Operational Test and Evaluation Center (AFOTEC) is tasked with the evaluation of operational effectiveness of new systems for the Air Force. Currently, the software analysis team within AFOTEC has no methodology to directly address the effectiveness of the *software* portion of these new systems.

This research develops a working definition for software effectiveness, then outlines an approach to evaluate software effectiveness-- the Software Effectiveness Traceability Approach (SETA). Effectiveness is defined as the degree to which the software requirements are satisfied and is therefore application-independent.

With SETA, requirements satisfaction is measured by the "degree of traceability" throughout the software development effort. A degree of traceability is determined for specific pairs of software life-cycle phases, such as the traceability from software requirements to high-level design and low-level design to code. The degrees of traceability are combined for an overall software effectiveness value.

It is shown that SETA can be implemented in a simplified database, and basic database operations are described to retrieve traceability information and quantify the software's effectiveness.

SETA is demonstrated using actual software development data from a small software component of the avionics subsystem of the C-17, the Air Force's newest transport aircraft.

AN APPROACH TO EVALUATE SOFTWARE EFFECTIVENESS

1. Introduction

The Air Force Operational Test and Evaluation Center (AFOTEC) is the single operational test agency for Air Force acquisition programs. AFOTEC is tasked with the evaluation of operational effectiveness and suitability of new systems for the Air Force.

The software analysis team within AFOTEC is responsible for the evaluation of the software portions of systems. Although AFOTEC has standard methodologies to evaluate *system* operational effectiveness, they currently have no standard methodology that directly addresses *software* effectiveness. Because of the increased number of software-intensive systems being introduced to the Air Force, Air Force Test and Evaluation (AF/TE) has suggested that software effectiveness be specifically evaluated by AFOTEC [PRC94]. This research outlines an approach to evaluate software effectiveness. From this outline, AFOTEC's software analysis team can develop a full methodology to evaluate software effectiveness.

1.1 Background

Computer software increasingly affects everyone's life both directly and indirectly. It is part of the tools used in everyday life in the 20th century: appliances, automobiles, and television. Software is also more and more a part of complex systems

that underlie business infrastructure: air traffic control, telephone service, and the stock market. Computer software is also important to the United States armed forces.

The military has unique systems that rely on computer software. Most of the aircraft in the Air Force would not function at all if it were not for computer software [Kit89]. Software also runs the Navy's ships and the Army's tanks. The functionality of all the military's weapons systems, information systems, and communications equipment depend on software. The military's software dependence is made evident by a comparison of hardware cost to software cost as a percentage of total system cost. Over a 23-year study period (1962-1985), life cycle costs for large, complex weapons systems flipped from an 80:20 ratio, hardware to software, to a 20:80 ratio [DAF94]. The trend towards software dependence has continued to the present.

Ironically, with this increased dependency on computer software, it has become all too common (almost a cliché, unfortunately) that software development efforts have resulted in products that are late, over budget, incorrect, or incomplete. The military has not escaped this dilemma. Software functionality, regardless of the type of system (weapon, communication, or information management) has suffered from performance shortfalls, as well as cost and schedule overruns [DAF94]. For example, as of 1989 the C-17 program had schedule delays and cost overruns of half a billion dollars; the worldwide command-and-control information system was turned over to another Air Force agency and restructured after consuming eight years and over a third of a billion dollars [Kit89].

This situation results in a paradox: an increasing dependence on software, accompanied by the inability of software developers to reliably produce usable software. This paradox, along with the exploding costs of correcting and maintaining poor quality software, is the driving force behind the need for software engineering.

Software engineering, the application of engineering discipline to software development, encompasses a broad range of activities including the assessment and evaluation of software. Assessments and evaluations measure attributes of software in terms of its structure and function. Structural attributes include complexity, modularity, and maintainability. Functional attributes include reliability, usability, and efficiency.

AFOTEC's software analysis team (AFOTEC/SAS) has several methodologies to assess software attributes such as reliability, usability, and maintainability. The software analysis team performs these assessments in support of testing for system operational effectiveness. Evaluating software effectiveness as a vital component of the system leads to a more complete evaluation of overall system effectiveness. Presently, software effectiveness is determined by, and dependent on, system effectiveness and is generalized as follows: "if the system works, the software works" [DAF94].

1.2 Problem Statement

AFOTEC does not have a methodology to directly address the evaluation of software effectiveness of the software portion of new systems acquired by the Air Force and currently depends on the evaluation of system effectiveness to determine software effectiveness.

1.3 Research Objectives

The purposes of this research were to define software effectiveness and propose an approach to evaluate software effectiveness. From this approach, AFOTEC's software analysis team could develop a full methodology to evaluate software effectiveness. In order to carry out this research, the effort was divided into the following objectives:

- 1) Develop a working definition of software effectiveness.
- 2) Research software effectiveness, including both military and industrial information on related topics such as software testing, software quality, and software verification and validation. If necessary, draw parallels from discussions of effectiveness in other areas of study.
- 3) Research various approaches to evaluate software effectiveness and identify where these approaches overlap with current Air Force developmental testing policies and practices.
- 4) Recommend an evaluation approach for AFOTEC's Software Analysis Team (AFOTEC/SAS) to develop into a full methodology.

A working definition was developed first, since effectiveness had to be defined before it could be evaluated. The development of the definition occurred simultaneously with the research for software effectiveness evaluation approaches. All the available information on software effectiveness was pooled to create a standard working definition. With the working definition in mind, the various approaches to evaluate software effectiveness were outlined and one approach was recommend to AFOTEC to develop into a full methodology.

1.4 Research Questions

While focusing on the research objectives, this research attempted to answer the following questions:

- 1) Does this software effectiveness evaluation support AFOTEC's system effectiveness evaluation?
- 2) How can current software development practices facilitate the evaluation of software effectiveness?
- 3) Can the effectiveness evaluation be used during the software development process as an early indicator of the software's effectiveness, i.e., before it reaches AFOTEC for operational test and evaluation (OT&E)?
- 4) Can the effectiveness evaluation be used to determine the product's readiness to *enter* OT&E?

1.5 Scope

The primary objective of this research was to outline an approach to evaluate software effectiveness. The approach focused on the software product, not the process in which it is developed. Although information is collected during the software development process to aid in the evaluation of the software's effectiveness, this study did not address any way of evaluating the software development process.

In addition, this research did not suggest any ways of improving the software, if necessary, once the effectiveness evaluation is made. It is up to the stakeholders in the development of the software to decide what changes need to be made, if any, to the software or the development process.

1.6 Assumptions

There were a few assumptions involved in this research concerning the software product under evaluation and the process in which it is developed.

Software development is a broad and diverse activity, and the capabilities of software developers vary widely. This study assumed that the software being evaluated represented a typical system that is evaluated by AFOTEC, i.e., a system with software of significant size and complexity. This study also assumed that the software being evaluated was developed with a defined, repeatable process. A further assumption was this development process provided the various documents referenced in the evaluation approach, such as the Software Requirements Specification (SRS).

1.7 Sequence of Presentation

This research began with an investigation into current methods of evaluating software effectiveness. Unfortunately, the evaluation of software effectiveness was not directly addressed in many documented cases. The few sources that directly addressed software effectiveness are summarized in Chapter 2. Chapter 3 describes the methodology used in conducting this research, and accounts for the changes in the original research plan, due to the lack of information on software effectiveness. The research methodology outlines the modified research plan and constructs the working definition of software effectiveness.

After constructing a definition for software effectiveness, Chapter 4 outlines an approach to evaluate software effectiveness. The evaluation approach utilizes data

gathered during the software development process and results in a quantified value of software effectiveness.

Once the approach is outlined, it must be demonstrated. The evaluation approach is demonstrated in Chapter 5. Conclusions and recommendations for further research are contained in the final chapter, Chapter 6.

2. Literature Review

2.1 Introduction

The original purpose of this chapter was to review the available literature and commercial practices for information on software effectiveness. Based on the information gathered, the intent was to propose a working definition of software effectiveness and outline several approaches to evaluate software effectiveness. Unfortunately, there were very few references to software effectiveness in the literature and in commercial practice. This lack of material led to several changes in the original research plan, including the creation of a working definition of software effectiveness; these changes are described in detail in Chapter 3.

The next section of this chapter summarizes the few direct references to software effectiveness found in the literature. Section 2.3 describes the search for references to software effectiveness in the commercial software development industry. Then, Section 2.4 reviews military software documentation in a search for references to software effectiveness. The final section contains a brief summary and conclusion.

2.2 Literature Directly Addressing Software Effectiveness

A search for previous studies on software effectiveness revealed only three articles that directly address software effectiveness [PRC94, Sta91, Zan92]. In 1994, PRC, Inc. was tasked with describing software effectiveness evaluation methodologies for AFOTEC/SAS. Although the topic of Stanko's thesis was not software effectiveness,

he defines software effectiveness in terms of software reliability [Sta91]. Zane studies the effectiveness of educational software [Zan92]. Two of these three references [PRC94, Zan92] specifically point out the scarcity of previous research in software effectiveness.

2.2.1 PRC, Inc. Report on Software Effectiveness. PRC, Inc., hereafter referred to as PRC, states that software effectiveness is difficult to define, closely related to quality, and based on the performance of the software [PRC94]. PRC defines software operational effectiveness as the degree to which the software supports mission accomplishment [PRC94]. This is not an adequate response to AFOTEC's questions about a definition and evaluation method for software effectiveness, for two reasons: 1) AFOTEC defines *system* effectiveness in terms of mission accomplishment [AFO95], and 2) at present, software effectiveness is determined by and dependent on system effectiveness [DAF94] (hence the reason for this study). By defining software effectiveness in terms of system effectiveness, the solution PRC provides AFOTEC is the practice AFOTEC is trying to get away from, i.e., determining software effectiveness by system effectiveness! The definition of software effectiveness should coincide with the definition of system effectiveness, but should not be defined by system effectiveness.

In reference to quality, PRC states that "the quality attributes of the software do, in fact, reflect the software's operational effectiveness" [PRC94]. This may be true, but apparently the software quality attributes that AFOTEC does address (maintainability, usability, maturity, reliability) are not adequate to define software effectiveness.

PRC also states that, like most software attributes, software effectiveness is based on performance [PRC94].

In their report, PRC presents eight “methodologies” for evaluating software effectiveness. Each methodology is categorized by four characteristics, with each characteristic offering a number of options in implementation. These options are taken in various combinations for each methodology, i.e., one set of options (one option for each characteristic) constitutes one software effectiveness methodology. The four characteristics and their options are summarized below [PRC94]. However, before these options are summarized, the definitions of Operational Test and Evaluation (OT&E) and Development Test and Evaluation (DT&E) are provided:

Operational Test and Evaluation. Testing and evaluation conducted to estimate the system’s military utility conducted in as realistic an operational environment as possible to estimate the prospective system’s operational effectiveness and suitability. In addition, operational test and evaluation provides information on organization, personnel requirements, doctrine, and tactics. Also, it may provide data to support or verify material in operating instructions, publications, and handbooks [AFO95].

Developmental Test and Evaluation. That testing and evaluation used to measure progress, to verify accomplishment of development objectives, and to determine if theories, techniques, and material are practicable and if systems or items under development are technically sound, reliable, safe, and satisfy specifications [AFO95].

The four options from the PRC report are listed below [PRC94]:

1) AFOTEC Software Quality Assurance (SQA) Role. AFOTEC currently has no responsibility in planning, implementing, or monitoring the quality programs during software development. Options for this characteristic include AFOTEC’s involvement in software process improvement, providing an operational test readiness certification monitor, or providing a proactive operational test readiness certification participant.

2) OT&E Task Measurement Method. There are two primary options to evaluate the effectiveness of a system during OT&E. Functional level characteristics data can be acquired and aggregated upward to answer the OT&E critical operational issues and measures of effectiveness as well as other required issues. The other option is to evaluate the system, and its software, from the user's perspective, i.e., user satisfaction.

3) OT&E Data Sources. Data for the OT&E can be obtained from the information derived during the DT&E efforts or it can come exclusively from the OT&E mission exercises. While the latter method might be preferred, the DT&E data may provide the opportunity to significantly reduce redundancy in testing.

4) OT&E Scenario Development. The DT&E scenarios are designed to demonstrate the compliance with the software's approved specifications, not to specifically stress the software to identify its breaking points. Also, DT&E does not necessarily target the software's critical paths as part of the specification compliance demonstration. These are extremely important to operational users. For some of the software effectiveness evaluation methodologies, the use of realistic operational scenarios to stress the software's operational profile processing capabilities and critical paths will be emphasized.

As stated earlier, the software effectiveness evaluation "methodologies" outlined by PRC were developed from various combinations of the options for these four characteristics. In fact, PRC does not outline specific methodologies for evaluating software effectiveness. The combinations of these four characteristics merely describe circumstances in which AFOTEC should be more capable of evaluating the effectiveness of software, as PRC defines it. PRC concludes:

Software quality measures are related to readiness for OT&E and the users' satisfaction with the software. The options for evaluating software operation test readiness and effectiveness can be characterize by the level of AFOTEC involvement in software quality measurement and evaluation of the software user's satisfaction during OT&E [PRC94]:

PRC also states that software evaluations presently conducted by AFOTEC, such as suitability, maturity, and maintainability, can be used as early indicators of software

effectiveness [PRC94]. This is true if AFOTEC further defines effectiveness in terms of these attributes that they already assess.

In summary, the PRC study provides no new information to AFOTEC. PRC's definition of software effectiveness is derived directly from current AFOTEC terminology. In addition, the eight "methodologies" PRC offers are not methodologies at all. Finally, in criticism of their own software effectiveness "methodologies", PRC states that each of the eight methodologies does not adequately assess software effectiveness.

2.2.2 Stanko's Thesis on Software Reliability. Stanko defines software operational effectiveness as being "... based on reliability as derived from test, or execution time" [Sta91]. In his thesis, entitled *A Standardized Software Reliability Measurement Methodology*, Stanko presents the following definitions:

Software System Effectiveness. A measure of the percentage of time the software system operates correctly (no failures) versus the total attempted operational time [Sta91].

Failure. The inability of a system or system component to perform a required function within specified limits [Sta91].

Stanko presents several different reliability models to assess their usefulness to evaluate software effectiveness. Stanko concludes that "software reliability provides one way of measuring the operational effectiveness of the weapon system software," and this can be used to calculate the total weapon system effectiveness [Sta91].

Although Stanko's thesis doesn't directly define software effectiveness as meeting functional requirements, his research states the following: software effectiveness is "based on reliability"; reliability is defined as the probability of failure-free operation;

failure is defined as the inability to perform a required function within specified limits [Sta91]. Therefore, Stanko implicitly defines software effectiveness as the degree to which the software meets its functional requirements.

Although the focus of Stanko's thesis was not software effectiveness, but rather software reliability, his definitions imply a point that will be useful in developing a working definition of software effectiveness; to be effective, the software must meet its functional requirements.

2.2.3 Zane's Study of Educational Software Effectiveness. In his study of the effectiveness of educational software, Zane contacted educational software developers and asked them to provide any documentation that supported the claims they made with respect to their software and improved learning performance [Zan92]. Zane defines educational software effectiveness as a measurement of improved learning performance through the use of the software. Of the 34 software development companies contacted, none provided any data to substantiate their claims of increased learning capability from using their products. Zane concludes that although software effectiveness is one of the most important attributes to measure in software, especially educational software, it is rarely done [Zan92]. The most significant determination in Zane's paper is that he explicitly defines effectiveness by comparing actual performance to expected performance.

2.2.4 Summary. The main theme in all three studies is that software effectiveness is based on the actual performance of the software compared to the expected performance of the software. PRC defines software effectiveness as the degree of software support towards mission accomplishment [PRC94]. Stanko defines software effectiveness based

on reliability [Sta91], and reliability is certainly focused on performance. Finally, Zane defines software effectiveness as the degree to which learning improves in relation to the claims made by the software's developers [Zan92]. Improved learning is a means of evaluating how well the software performs. In accordance with these few direct references to software effectiveness, any definition of software effectiveness must be rooted in software performance. Software performance is addressed in the development of the working definition of software effectiveness in Chapter 3.

2.3 Industry Perspective on Software Effectiveness

Four companies were contacted and questioned about their use of the term "software effectiveness" in software development. The four companies covered a broad spectrum of software development efforts, including: commercial (Microsoft), corporate (IBM), aviation (Boeing), and defense (Lockheed Martin). Each company was asked the same initial question: "How do you define software effectiveness, and how does your company address it, if at all?" Without exception, each company answered this question with another question. Each company asked (in one way or another): "What do you mean by *effectiveness*?" Additional details from the individual interviews are documented below.

2.3.1 IBM. Margaret Hedstrom, former member of IBM's Software Quality Engineering Group, stated that IBM does not specifically address software "effectiveness" by that term [Hed96]. IBM is concerned with quality throughout development and after delivery, with quality meaning "more than bug-free" [Hed96]. The overriding issues for IBM's software development efforts include: functionality,

usability, and meeting customer expectations. After product delivery, IBM addresses customer satisfaction by administering follow-on customer surveys [Hed96].

2.3.2 Boeing. John Vu, Senior Principal Scientist at Boeing's Software Engineering Research and Technology Division, stated that Boeing has "no official position" on software effectiveness [Vu96]. The critical issues in Boeing's software development efforts include: meeting requirements, performance, minimizing defects, and developing software on time, within budget.

2.3.3 Microsoft. Greg Enslow, Technical Sales Representative of the Developer Tool Sales Team, stated that Microsoft does not directly address software "effectiveness", but Microsoft *is* concerned with software quality [Ens96]. Microsoft has created various methods to address quality in the software products they develop.

2.3.4 Lockheed Martin. Gerald Nieto, Manager of the Application Division's Project Management Support, stated that "effectiveness is not a term that's used all the time, but it's occurring more frequently" [Nie96]. Lockheed Martin relates software effectiveness to process improvement, reduced development time, reduced costs, and the software's contribution to the whole product. Although Lockheed Martin has begun collecting metrics on what it terms as "effectiveness", the focus is on the software development process, rather than the effectiveness of the software product itself [Nie96].

2.3.5 Summary. Without exception, no company contacted used the term "software effectiveness" in the development or evaluation of their software products. The absence of the term "software effectiveness" in industry is reiterated by PRC in their study [PRC94]. While not scientific and not conclusive, this small survey of large,

commercial software developers indicates that the term “software effectiveness” is not widely used in the development of software in industry.

2.4 Military Perspective on Software Effectiveness

2.4.1 Department of Defense. There is no reference to software effectiveness in Department of Defense (DoD) Regulation (DoDR) 5000.2-R, which contains the mandatory procedures for major defense acquisition programs and major automated information system acquisition programs [DoD96b]. Although there are many references to *system* effectiveness and the important role software plays in major systems acquired by the military, there is no mention of *software* effectiveness with regard to definition or evaluation method.

2.4.2 Air Force Test and Evaluation. There is no reference to software effectiveness in Air Force Instruction (AFI) 99-103, *Test and Evaluation Process*, which “directs and describes the Air Force Test and Evaluation Process and its relationship to the systems acquisition process within the policies” of the Department of Defense [AFI94c]. As with DoDR 5000.2-R above, there are many references in AFI 99-103 to system effectiveness and the important role software plays in major systems. AFI 99-103 also emphasizes the importance of testing software as an integral part of the system [AFI94c], but there is no mention of software effectiveness.

2.4.3 Air Force Developmental Test and Evaluation. AFI 99-101, *Developmental Test and Evaluation*, contains one reference to software effectiveness [AFI94a]. In a list of what developmental test and evaluation (DT&E) programs test, under the system performance section, AFI 99-101 states: “Assess the system's software (including its

effectiveness, suitability, and interoperability aspects) and identify limiting factors” [AFI94a]. This is the only reference to software effectiveness in AFI 99-101. Although there is information contained on many other aspects of system DT&E, including system effectiveness, there is no definition of software effectiveness and no guidance on how to assess software effectiveness.

2.4.4 Air Force Operational Test and Evaluation. There is no reference to software effectiveness in AFI 99-102, *Operational Test and Evaluation*, which “provides guidance and procedures for operational test and evaluation in the Air Force” [AFI94b]. Also, there is no reference to software effectiveness in AFOTEC Instruction (AFOTECI) 99-101, *Management of Operational Test and Evaluation*, which “provides the specific guidelines and procedures for the Air Force Operational Test and Evaluation Center (AFOTEC) conduct of operational test and evaluation (OT&E) on Air Force systems” [AFO95]. AFOTEC Pamphlet (AFOTEC P) 99-102, *Software Operational Assessment Guide*, contains one reference to software effectiveness [AFO94]. Stating that *software* operational assessment areas must mirror the *system* operational assessment areas, AFOTEC P 99-102 lists “impacts affecting operational effectiveness” as one of five standard areas to be assessed. Although AFOTEC P 99-102 contains information on many other aspects of software assessments, there is no definition of software effectiveness and no guidance on how to assess software effectiveness.

The *Software Operational Assessment Guide* [AFO94] is the eighth volume in a series of OT&E guidelines (AFOTEC Pamphlet 99-102) prepared by AFOTEC/SAS. Although there are guidelines to assess software attributes such as maintainability,

usability, maturity, and reliability, there is no guideline to assess software effectiveness. At present, AFOTEC/SAS determines software effectiveness with the assessment of the software attributes mentioned previously, aided by the evaluation of system effectiveness.

2.5 Summary and Conclusion

Information that directly addresses software effectiveness is quite sparse. Of the three sources found in the literature [PRC94, Sta91, Zan92], the only useful information they offered was that software effectiveness is based on performance. Although interviews with only four software development companies hardly constitutes a complete scientific survey, it is generalized that software effectiveness is not addressed in industry [Ens96, Hed96, Nie96, Vu96]. Of the four military software documents reviewed, including documents from the DoD, Air Force, DT&E, and OT&E communities, only two references to software effectiveness were found [AFI94a, AFO94]. These references to software effectiveness in the military documentation were not accompanied by a definition of or an evaluation method for software effectiveness.

This lack of information dictated that changes had to be made to the original research plan. With little information on software effectiveness in the available literature, industry, and military documentation, it was necessary to develop a working definition of software effectiveness by other means. Once a working definition was established, an approach to evaluate software effectiveness could be outlined. The changes to the research plan, along with the development of a working definition of software effectiveness, are described in Chapter 3. Using the software effectiveness definition

developed in Chapter 3, an approach to evaluate software effectiveness is outlined in Chapter 4.

3. Research Methodology

It is vital to document the research plan undertaken in any study. The research plan reveals the steps (and missteps) taken along the path in the attempt to solve the problem at hand. The plan is important to the reader of the research in that the plan must follow a logical process and common sense or the results of the research effort have little meaning. The research plan is also important to the author of a research effort to document the original plan, the problems encountered following the original plan, and the modified plan. Another benefit to the author of a research effort is the documentation of the trials, mistakes, and dead ends encountered during the research. In the end, more may be learned from what does *not* work than from what works according to the research plan.

3.1 Introduction

This chapter documents the research plan, hereafter referred to as the plan, and its evolution throughout the research effort. This chapter is a direct result of the problems encountered during the research into software effectiveness. The first section below outlines the original plan to investigate software effectiveness and develop an approach to evaluate software effectiveness. The next section describes the difficulties encountered while following the original plan. In an attempt to develop an approach to evaluate software effectiveness, these difficulties led to several dead ends which are described in Section 3.4. As a result of these dead end approaches, it was necessary to modify the original plan; this modified plan is outlined in Section 3.5.

The first step of the modified plan is accomplished in Sections 3.6 through 3.9 with the development of a working definition of software effectiveness. Since the information reviewed in Chapter 2 is inadequate to develop a working definition of software effectiveness, a preliminary definition is formed in Section 3.6, starting with a “clean slate” and just using the word *effective*. This preliminary definition of software effectiveness is refined in Sections 3.7 and 3.8, by examining “effectiveness” in various contexts. The definition is further refined by Section 3.9, which summarizes two other software attributes that are comparable to software effectiveness. Following these four development sections, 3.6 through 3.9, a working definition of software effectiveness is provided in Section 3.10.

After presenting the working definition for software effectiveness, the focus turns to the *evaluation* of software effectiveness. Section 3.11 outlines software development activities that assess software attributes, since these activities may aid in the development of an approach to evaluate software effectiveness. Next, Section 3.12 contains a chapter summary and Section 3.13, the final section, concludes with the main idea in the proposed approach to evaluate software effectiveness, which is described in detail in Chapter 4.

3.2 Original Research Plan

The original research plan came from a series of objectives outlined by AFOTEC [Pro95]. After providing some background information as to the need for a methodology

to evaluate software effectiveness, AFOTEC/SAS outlined some basic objectives which were adapted to the objectives listed in Section 1.3 and repeated below:

- 1) Develop a working definition of software effectiveness.
- 2) Research software effectiveness, including both military and industrial information on related topics such as software testing, software quality, and software verification and validation. If necessary, draw parallels from discussions of effectiveness in other areas of study.
- 3) Research various approaches to evaluate software effectiveness and identify where these approaches overlap with current Air Force developmental testing policies and practices.
- 4) Recommend an evaluation approach for AFOTEC's Software Analysis Team (AFOTEC/SAS) to develop into a full methodology.

This original plan seemed fairly straightforward, assuming information on software effectiveness was available.

In summary, the original plan was to review all the available information and develop a working definition for software effectiveness. Supposedly, in the course of the research, several approaches to evaluate software effectiveness would be encountered. The software effectiveness approaches would then be outlined, compared, contrasted, and one approach would be recommended to AFOTEC/SAS to develop into a full methodology. AFOTEC's outline of objectives implied that information on software effectiveness was readily available, but this was not the case.

3.3 Difficulties Encountered During Research

3.3.1 Lack of Previous Research. Although the research outline provided by AFOTEC implied the availability of information on software effectiveness, there were

very few sources that directly addressed software effectiveness. These few sources also emphasized a scarcity of information on software effectiveness. Also, AFOTEC delayed in providing one source that directly addressed software effectiveness [PRC94], so as not to influence the start of this research. AFOTEC provided the PRC report after an initial search for information on software effectiveness came up empty. Unfortunately, this document proved problematic.

3.3.2 Problems with the PRC Report. The report from PRC, entitled *Software Effectiveness Evaluation Methodology Study Task Report of Concept Options*, was intended to be an important reference in this research effort, but actually provided little information on software effectiveness. This document defined software effectiveness with the same terminology (mission accomplishment) as AFOTEC's definition of system effectiveness, and did not provide any new insight into software effectiveness.

The methodologies the PRC document offered were not methodologies at all. PRC's "methodologies" were merely eight options of suggested changes in quality assurance roles, measurement methods, and data sources that AFOTEC could undertake. By making these changes, AFOTEC would supposedly be better able to assess software effectiveness. PRC went on to criticize their own suggestions for effectiveness approaches by stating that none of them would adequately assess software effectiveness.

Finally, the PRC document had structural problems in the way it was written. Although the PRC document had an extensive bibliography, there were no citations inside the document to indicate what information came from which source. Even though the bibliography listed some sources that provided some relevant background

information, there was one source in the bibliography that could not be located. The phantom document had authorship credited to AFOTEC/SAS and was entitled *Software Operational Effectiveness Assessment for C-17*. This document should have provided some insight into a software effectiveness evaluation method, but no one at AFOTEC had ever heard of the document, much less written it. The same answer came from the C-17 System Program Office at Wright-Patterson AFB; no one had ever heard of the document.

3.3.3 No References in Industry. The difficulties in the research continued when it was determined that there were no references to software effectiveness in industry. While only four companies in industry were surveyed, they covered a broad range of software development activities in size and scope. If software effectiveness was being addressed anywhere in industry, it should have been encountered in one or more of the following companies that were surveyed: IBM, Microsoft, Boeing, and Lockheed Martin. All the companies contacted stressed that they were concerned with quality, customer satisfaction, and meeting budget and schedule constraints. Unfortunately, none of the companies used the term "software effectiveness", and it is therefore assumed that software effectiveness is not addressed at any great length anywhere in industry.

3.3.4 Summary. The scarcity of information directly addressing software effectiveness was disappointing, but not completely debilitating. The nagging question was why. Why was there virtually no reference material on software effectiveness? AFOTEC is the single responsible agency for operational test and evaluation for new systems (and their software) acquired by the United States Air Force. AFOTEC has enough concern about software effectiveness to sponsor a thesis on the topic; why is it

that no one else in the software development community seems to be concerned about software effectiveness?

With very little reference material for guidance, a definition for software effectiveness would have to be developed by other means. Assuming a general definition of effective as another software attribute meaning “adequate” or “appropriate”, a few evaluation approaches were considered and eventually rejected (for good reasons) as dead ends.

3.4 Dead Ends for Software Effectiveness Evaluation Methods

3.4.1 User's Perspective. Influenced by numerous horror stories of unmet user expectations in software development, software effectiveness was initially viewed exclusively as user satisfaction. The end-user would determine whether the software was “effective” or not.

This approach was rejected because there are considerations with the development of the software that are hidden from the user. These considerations would alter the “effectiveness” of the software in its final form. For example, consider software maintainability. Suppose a user initially determines that a software system is effective. His or her opinion would likely change if a requested software change proposal took too long to complete, or was not completed at all.

3.4.2 Static Set of Software Attributes. A fixed set of attributes could compose the overall attribute of software effectiveness. Assuming functionality and reliability are necessary attributes in any software development effort, other attributes would be added to create a composite definition of software effectiveness.

This approach was rejected because it is not application-independent. With the many types of software products in development, it is impossible to select a fixed set of attributes to form a definition of software effectiveness. For example, user-friendliness would certainly be a desirable attribute to determine effectiveness in a database application. However, to determine software effectiveness in an embedded system such as a cruise missile, user-friendliness would not be considered a necessary, or even a desirable attribute of the missile guidance software.

3.4.3 Dynamic Set of Software Attributes. A dynamic set of attributes could compose the overall attribute of software effectiveness. Selecting software attributes “cafeteria-style” to determine software effectiveness would certainly remove the problem of application dependence. User-friendliness would be selected as an attribute to determine software effectiveness for the database application, but not as an attribute for the on-board cruise missile software. Survivability may be selected as a necessary attribute to determine software effectiveness for the cruise missile software, but survivability is hardly necessary for a database application.

Unfortunately, this approach was rejected because it is too subjective. In determining software effectiveness, *what* attributes are selected will be influenced by *who* is making the selection. Opinions may vary greatly as to what is (or is not) an important attribute to determine software effectiveness for each application under consideration.

3.4.4 Dynamic/Static Sets of Metrics. Perhaps the quantitative nature of metrics could provide an evaluation approach for software effectiveness. By collecting metrics on the software during the development process, this data could possibly serve as an

“early indicator” of software effectiveness, which would appeal to AFOTEC. This approach seemed the most promising of all considered so far.

Unfortunately, the metrics approach was rejected for the exact same reasons as the attribute approach. A static set of metrics would not be application-independent and a dynamic set of metrics would be too subjective.

3.4.5 Effectiveness Viewed as Another Term for Quality. From the general definition of software effectiveness, described above as “adequate” or “appropriate”, could software effectiveness just be another term for quality? This would certainly explain the scant information on software effectiveness in literature and in practice. Perhaps all the effort and consideration that would have been applied to software effectiveness has already been expended on another attribute closely related to “adequate” or “appropriate”, such as software *quality*. Perhaps software effectiveness can be evaluated by addressing software quality.

This approach was also rejected, for a number of reasons. First, quite simply, it is not what AFOTEC/SAS requested. The focus of this research is software effectiveness, not software quality. Although the connection between software effectiveness and quality must be considered as part of this study, effectiveness will not be defined in terms of quality. Secondly, defining software effectiveness in terms of quality is merely substituting one difficult-to-define term for another, so nothing is gained. In addition, quality implies consideration not only of the software product, but the process with which it was developed. Software effectiveness concerns the software product exclusively; at best it could be defined as a subset of quality attributes. Lastly, if software effectiveness

is defined as a subset of quality attributes, this approach becomes the first approach that was rejected: a set of attributes to define software effectiveness.

3.4.6 Summary. All of the software effectiveness evaluation approaches above were considered and rejected for various reasons. This “thrashing” of ideas stemmed from a loose interpretation of the term effectiveness. It was necessary to go “back to the drawing board” to develop a working definition of software effectiveness; the development of the working definition is covered extensively in Sections 3.6 through 3.9 below.

However, before developing the working definition of software effectiveness, a modified research plan is outlined in the next section, since the difficulties and dead ends encountered during the research have changed the original research plan.

3.5 Modified Research Plan

The modified research plan retained as much as practically possible from the objectives outlined by AFOTEC/SAS. The objectives of the modified research plan are listed below:

- 1) Develop a working definition of software effectiveness.
- 2) Research effectiveness in other areas of study, including AFOTEC’s definition of system effectiveness. Also research other performance-based software attributes and activities such as software quality, software reliability, software testing, and software verification and validation.
- 3) Since there are no software effectiveness methodologies to review, develop one approach to evaluate software effectiveness to recommend to AFOTEC/SAS to develop into a full methodology.
- 4) Demonstrate the operation of the recommended approach to evaluate software effectiveness.

This modified plan was not drastically different than the original research plan. The working definition of software effectiveness had to be developed by other means since the previous work on the topic was extremely limited. These limited resources also called for the development of a software effectiveness evaluation approach. The output products of this modified plan were identical to the original research plan: a working definition of software effectiveness and a recommended approach to evaluate software effectiveness. The working definition of software effectiveness is developed in the next four sections, with the actual definition presented in Section 3.10. The approach to evaluate software effectiveness is explained in detail in Chapter 4.

3.6 Preliminary Definition of Software Effectiveness

Software effectiveness is difficult to define precisely. Without a firm definition, software effectiveness is impossible to evaluate. The *Random House Webster's College Dictionary* defines effective as "adequate to accomplish a purpose; producing the intended or desired result" [Ran92]. In short, effective means "fit for use."

The dictionary definition of effective (and the short definition, "fit for use") denotes performance or functionality, brought out by the words "accomplish" and "producing" (and "use"). Effectiveness, as the noun form of the adjective "effective", describes a property of an object or process. In short, effectiveness means "degree of fitness for use."

To associate software with the basic definition above, software effectiveness is

the degree to which the software performs as required and expected.

Software effectiveness must answer the questions: “Does the software do what it is supposed to do?” and “Does the software *not* do what it is *not* supposed to do?” This preliminary definition of software effectiveness is referred to in subsequent sections in its short form: “performs as required”. To satisfy the first goal of this thesis, a working definition for software effectiveness will be developed with software performance in mind.

AFOTEC tests systems for system effectiveness. Since this software effectiveness approach is being developed for AFOTEC, it is logical to consider their definition of system effectiveness in refining the preliminary definition of software effectiveness.

3.7 AFOTEC's Definition of System Effectiveness

AFOTECI 99-101 refers to operational effectiveness and suitability as the two benchmarks of OT&E and defines *system* operational effectiveness as follows:

System Operational Effectiveness. The degree of mission accomplishment of a system when used by representative personnel in the environment planned or expected (e.g., natural, electronic, threat, etc.) for operational employment of the system considering organization, doctrine, tactics, survivability, vulnerability, and threat (including countermeasures, initial nuclear weapons effects, nuclear, biological, and chemical contamination threats) [AFO95].

From the definition of system operational effectiveness, the mission in “degree of mission accomplishment” is divided into operational and support tasks that are necessary for the achievement of a military objective. Critical Operational Issues (COIs) are questions about the system’s operational tasks that must be answered “yes” or “no” as to

whether the issue has been addressed or not [AFO95]. COIs can be composed of one or more Measures of Effectiveness (MOEs) or Measures of Performance (MOPs). MOEs are defined by the operational command as measures of operational capability in terms of engagement or battle outcome and MOPs are defined as quantitative or qualitative measures of the system's capabilities or characteristics [AFO95]. The "operational capabilities" of the MOEs and the "capabilities or characteristics" of the MOPs describe the operational requirements of the system. Therefore, for AFOTEC, system operational effectiveness generally measures the degree to which the operational requirements are met. This generalization supports the preliminary definition of software effectiveness, "performs as required", but adds no new information. Perhaps effectiveness definitions in other areas of study can aid in refining the preliminary definition of software effectiveness.

3.8 Effectiveness Definitions in Other Fields of Study

3.8.1 System Effectiveness. A "system" is a collection of individually working components, contained in a complex structure, and designed to function together. Examples of some large systems include power plants, factories, and satellite communication networks.

System effectiveness is based largely on performance [Sei69]. System components may include hardware, software, input and output data, and even subjective things such as operator experience and management pressure. With this many variables to consider, an effectiveness assessment of such a complicated system requires a divide

and conquer strategy. Seiler breaks down overall expected effectiveness of an entire system into the following components [Sei69]:

$$E(E) = [E(P) C_p] [E(A) C_a] [E(R) C_r] [E(S) C_s] \quad (1)$$

where $E(E)$ = expected effectiveness of the system
 $E(P)$ = expected performance of the system (the basic design)
 C_p = statistical confidence in performance
 $E(A)$ = expected availability of the system
 C_a = statistical confidence in availability
 $E(R)$ = expected reliability of the system
 C_r = statistical confidence in reliability
 $E(S)$ = expected survivability of the system
 C_s = statistical confidence in survivability

Each expected value component is broken down in a similar manner, until the components are quantifiable. For example, the reliability component of the system, $E(R)$, is decomposed into the reliability of the hardware, the reliability of the software, etc. The statistical confidence in reliability, C_r , is also broken down and matched with each component of the hardware, software, etc. The reliability of the hardware is then broken down further into reliability components of the hardware (disk platter, power supply, circuitry, etc.), again with their associated statistical confidence. Once all the components of the system are decomposed to the point where they can be assigned explicit values, they are assembled back into equation 1 to calculate a composite value describing the system's overall effectiveness.

Decomposition may prove useful in the evaluation of software effectiveness. A software "system" can be broken down into manageable components which could be individually evaluated for effectiveness.

3.8.2 Information Systems Effectiveness. In measuring the effectiveness of information systems, performance is the most important factor [Ash94, Eva88, Hua95, Sco95]. Performance of information systems is measured by different attributes such as user satisfaction, information quality, and reliability. This definition coincides with the preliminary definition of software effectiveness, “performs as required”.

Ashqar defines information systems effectiveness from two different perspectives: the user’s and the system’s [Ash94]. From the user’s perspective, Ashqar measures user satisfaction; from the system’s perspective, Ashqar measures performance in terms of resource utilization, cost, and efficiency [Ash94]. These different perspectives offer insight to a refinement of the preliminary definition of software effectiveness; if the software satisfies the user, yet is inefficient and costly in terms of system resources, can the software still be considered effective?

In generalizing the evaluation of information systems effectiveness, actual performance is measured against the expected performance of the information system, which coincides with the recurring theme in the sources that directly address software effectiveness.

3.8.3 Effectiveness of Teaching Methods. In a comparison study of teaching methods, Gillis evaluates effectiveness by assessing student learning [Gil85]. One teaching method consists of computer-based instruction, and the other method involves a human instructor. Both methods are used in teaching composition writing. Following the instruction, the students write compositions which are graded independently to assess which teaching method was more helpful in aiding student learning.

The salient point is that teaching effectiveness is measured by the degree of student learning (quantified by the grades on their compositions after the instruction), i.e., by student performance. This effectiveness measure also shows how well each teaching method fulfills the expectations of the evaluators.

3.8.4 Effectiveness of Strategies for Hardware Reconfiguration. Schwab describes reconfiguration strategies for real-time, very large scale integrated circuit processing arrays [Sch95]. Fault-tolerant designs of real-time systems require hardware reconfiguration to continue to function properly. Schwab evaluates these strategies for their effectiveness in successful reconfiguration. Effectiveness is determined by how well these reconfiguration strategies work to support the fault-tolerant circuit design.

3.8.5 Traffic Control Effectiveness. To combat traffic congestion, a number of intelligent-vehicle highway systems (IVHS) have been introduced [Lo94]. These systems gather real-time data on automobile traffic flow and provide guidance to motorists so they may respond to traffic conditions appropriately. Lo outlines an evaluation method for these systems to determine their effectiveness [Lo94]. Lo defines effectiveness as the degree of impact on the transportation system due to the application of IVHS. The IVHS is designed to improve traffic flow and Lo's impact evaluation method determines how well traffic flow is aided by the IVHS.

3.8.6 Shielding Effectiveness in Materials. Radford discusses various composite materials and evaluates their effectiveness in shielding against electromagnetic interference (EMI) [Rad94]. Shielding effectiveness is based on measured electrical conductivity in the material during exposure to EMI. The EMI is absorbed and directed away from the component or structure the material is shielding. In shielding against EMI,

effectiveness is a quantified measure using a standard, accepted formula in the study of composite materials. The specific formula is unimportant in regards to software effectiveness; what is important is the effectiveness of the material is determined by how well the material performs in shielding against EMI.

3.8.7 Summary. The overriding theme in reviewing effectiveness in other fields of study is effectiveness is defined by how well something is done or indicates a level of accomplished tasks. Although these studies provide little new information to refine the preliminary definition of software effectiveness as “performs as required”, these studies substantiate the preliminary definition. Ashqar’s study provides insight into a refinement for the preliminary definition of software effectiveness; effectiveness can be looked at from different perspectives. In addition to the perspectives of the user and the system offered by Ashqar, other perspectives for *software* effectiveness may include the software developer, tester, or maintainer.

From the preliminary definition “performs as required”, software effectiveness can be viewed as an attribute of software that is based on performance. To further refine the preliminary definition of software effectiveness, it may be beneficial to examine other software attributes that are based on performance.

3.9 Performance-Based Software Attributes

3.9.1 Software Reliability. In reference to “performs as required”, software effectiveness has a connection with software reliability because both attributes are based on software performance. Although both terms are based on performance, each term views performance from opposite perspectives. In generalizing “performs as required”,

software effectiveness is the degree of satisfactory performance, while reliability is the degree of absence of unsatisfactory software performance.

Stanko defines software effectiveness in terms of reliability [Sta91], and implies through his definitions that to be effective, software must meet its functional requirements. However, this definition does not quite capture the preliminary definition of software effectiveness as “performs as required”. Reliability only addresses the performance half of “performs as required”; there may be other “requirements” that have little to do with actual performance. For example, reliability does not take into consideration non-functional attributes such as maintainability or constraints such as program size. Effectiveness can capture non-functional attributes and constraints if “performs as required” includes performance aspects as well as meeting the requirements of the software.

In summary, the refined definition of software effectiveness must contain reliability’s definition within it, i.e., “performs as required” is taken to mean the software performs adequately *and* meets software requirements.

3.9.2 Software Quality. Quality is defined by Robert Glass as “the degree of excellence of something” [Gla92]. Glass’ definition is certainly broad enough to contain reliability, as well as meeting the non-functional requirements described previously. Glass’ definition also coincides with “performs as required”, the preliminary definition of software effectiveness. In general, a high quality software product implies that it performs adequately and would therefore be effective. Also, software that is effective would generally imply that it is a quality product. However, software quality is also

difficult to define [Gla92], but unlike software effectiveness, there is a large volume of information that defines software quality and how to evaluate it.

Information on the definition and evaluation of software quality may aid in the definition and evaluation of software effectiveness. One traditional approach to define and evaluate software quality is to use a divide and conquer strategy and define quality in terms of attributes of the software product [Bow85, Cla92, Gla92, ISO91, War87]. In this manner, each attribute is assessed separately (much like Seiler's calculation of system effectiveness above), then combined to yield an overall appraisal of the software's quality. Another traditional approach to define and evaluate quality is to use software metrics. Many studies attribute software quality to a successful metrics program and vice versa [Oiv93, Ros94, She90, Wal91, Wel93]. These approaches, using a set of attributes or a set of metrics to define and evaluate software quality, were previously considered and rejected (in Sections 3.4.2 through 3.4.4) as ways to define and evaluate software effectiveness.

Finally, while software reliability concerned too narrow a scope by focusing on software performance, software quality covers too broad a scope by considering the software development process in addition to the quality of the software product.

Considering the refined preliminary definition of software effectiveness as adequate performance and meeting requirements begs the question: "can software performance be defined within the requirements?" The answer is "yes" and the working definition of software effectiveness is now established as simply "meeting the software requirements".

3.10 Working Definition of Software Effectiveness

Software Effectiveness. The degree to which the software requirements are satisfactorily met.

In this definition, the term “degree” refers to the percentage of software requirements that are satisfactorily met.

This definition addresses the questions: “Does the software do what it is supposed to do?” and “Does the software *not* do what it is *not* supposed to do?”, since what the software is supposed to do is defined by its requirements. This definition also coincides with AFOTEC’s definition of *system* effectiveness, the effectiveness definitions reviewed from other areas of study, and the definitions of software reliability and quality.

AFOTEC defines system operational effectiveness as the degree of mission accomplishment. With a mission broken down into tasks to support a military objective, these tasks can be viewed as mission “requirements” and mission accomplishment is merely “meeting” the mission tasks. Therefore, meeting requirements for software effectiveness is analogous to mission accomplishment for system operational effectiveness. Also, mission tasks are eventually decomposed into MOEs and MOPs which are measures of the system’s “capabilities or characteristics” [AFO95]. Capabilities and characteristics are the precise terms used in the *IEEE Standard Glossary of Software Engineering Terminology* to define software requirements [IEE90].

In reviewing other fields of study, effectiveness is determined by a comparison of measured performance to expected performance [Ash94, Eva88, Gil85, Hua95, Lo94, Rad94, Sch95, Sco95, Sei69]. This certainly agrees with the working definition of

software effectiveness, since the software's expected performance is documented in the software requirements. Particularly noteworthy is Ashqar's study of informational systems effectiveness and his view of effectiveness from multiple reference points [Ash94]. Perspectives for software effectiveness may include the developer, user, and maintainer, and these unique viewpoints are addressed in the software requirements.

At a minimum, software requirements must include adequate functionality and reliability. Without reliable functionality, all other software attributes have little meaning. Therefore, the definition of software effectiveness as meeting requirements is supported by software reliability, since some level of reliability is always assumed to be a software requirement.

The definition of software effectiveness coincides with software quality, since it is implied that a quality software product meets all its requirements.

In summary, all the reviewed material on effectiveness supports the definition of software effectiveness as meeting requirements. Since effectiveness is a software attribute, it may be beneficial to examine software development activities that evaluate software attributes, especially attributes that emphasize the satisfaction of software requirements.

3.11 Software Activities Related to Software Effectiveness

Now that software effectiveness is defined as a software attribute indicating the degree to which the software requirements are met, the focus turns to the *evaluation* of software effectiveness. During the software development process, certain activities are

conducted to evaluate software. Some of these activities are concerned with general software evaluation, such as testing; other activities, such as verification and validation, specifically evaluate the satisfaction of software requirements. An examination of these software evaluations may be useful in developing an approach to evaluate software effectiveness.

3.11.1 Testing. Software testing is often mistakenly identified as an activity that is conducted to show the software works properly. In actuality, software is tested to show where the software does *not* work properly [Gla92, Het88, Hum89, Pre93]. Software testing is ordinarily divided into two areas: requirements testing and design testing [Gla92, Het88]. With software effectiveness defined as meeting requirements, testing for requirements satisfaction may provide insight into developing an approach to assess software effectiveness.

Testing for requirements satisfaction implies at least three things: requirements are stated in such a manner that they are testable, tests are designed that show the requirements are met satisfactorily, and these tests are executed on the software. Hetzel outlines the paradox of stating a requirement by determining how to test the requirement and working backwards [Het88]. By concentrating on how the requirement will be tested, Hetzel states that ambiguity is reduced in specifying the requirement and the resulting requirement is testable [Het88]. With this method of designing software requirements by designing the tests for the requirements, the end result is a set of requirements and a set of tests to test those requirements. What remains to be done is the execution of those tests on the software, once the software is developed.

Considering the definition of software effectiveness, testing for requirements satisfaction is certainly comparable to evaluating software effectiveness. Also, for the tests designed to validate the software requirements, the degree to which the tests are passed satisfactorily is comparable to the degree to which the requirements are met. Therefore, the evaluation of software effectiveness is analogous to testing for satisfaction of software requirements. Another software activity that emphasizes the satisfaction of software requirements is verification and validation (V&V).

3.11.2 Verification and Validation. V&V is a software engineering discipline that examines the software during and after development to increase the likelihood that the resulting software product functions correctly and meets the user's expectations. Specifically, Wallace and Fujii define V&V as follows:

Verification and Validation. V&V comprehensively analyzes and tests software to determine that it performs its intended functions correctly, to ensure that it performs no unintended functions, and to measure its quality and reliability. Verification involves evaluating software during each life-cycle phase to ensure that it meets the requirements set forth in the previous phase. Validation involves testing software or its specification at the end of the development effort to ensure that it meets its requirements (that it does what it is supposed to do) [Wal89].

It is worth noting that this definition incorporates much of the terminology used in developing the working definition of software effectiveness, including:

- performs intended functions correctly (does what it is supposed to do)
- performs no unintended functions (does *not* do what it is *not* supposed to do)
- quality
- reliability

- meets requirements

In regards to software effectiveness, the task of meeting requirements is addressed in the “validation” portion of V&V. Validation is defined as a process of evaluating software at the end of the development process to ensure the software requirements are met [IEE86, IEE90, Lew92, Pre92, Sca94]. Therefore, the evaluation of software effectiveness is analogous to software validation, since validation ensures the software requirements are satisfied. The software evaluation in the validation process is accomplished by testing the software. Consequently, software validation and the evaluation of software effectiveness are also related because validation primarily involves testing, which is related to software effectiveness as established in the previous section.

Both testing and validation address the issue of requirements satisfaction well into the software development process. A working version of the software is needed to test or validate that the software requirements have been met. However, requirements satisfaction can be addressed indirectly during the software development process by “tracing” the requirements throughout the products of the software development process.

3.11.3 Requirements Tracing. The products of the software development process include the requirements, design, code, and tests. Requirements tracing establishes a relationship: 1) from an individual requirement, through the design, to the code, and 2) from the requirement to a test that validates the requirement. The purpose of the traces is “to establish a relationship between two or more products of the development process” [IEE90].

Assuming completely defined software requirements, complete requirements tracing implies requirements satisfaction. If a requirement is not considered in the design it cannot be satisfied, since the requirement is unlikely to be implemented in the code if it is not in the design. Since requirements tracing indicates the satisfaction of software requirements, it can be used to evaluate software effectiveness.

Traceability is an attribute of the connection between the software development products and is defined as follows:

Traceability. The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match [IEEE90].

Requirements traceability is analogous to software effectiveness. Traceability is the degree of connectivity between software development products. Complete traceability indicates complete requirements satisfaction. Software effectiveness is the degree to which the requirements are met satisfactorily. Therefore, the evaluation of requirements traceability may serve as an evaluation of software effectiveness.

3.12 Summary

As stated in the beginning of this chapter, it is vital to document the research plan undertaken in any study. This study is no different. Faced with an initial set of tasks to accomplish and armed with a set of assumptions, the research began. Sometimes the incorrect path was followed, but the elimination of an incorrect answer is as much of a learning experience as discovering a correct answer.

This chapter described the research process undertaken in this study. The original research plan, as suggested by AFOTEC/SAS, was outlined. Then, the problems encountered and dead ends pursued were detailed. Following the documentation of these problems, the modified plan was outlined. Limited references in publications and in practice led to the development of a definition for software effectiveness as part of the modified plan.

After defining software effectiveness, three software development activities were reviewed to provide insight into a possible method to evaluate software effectiveness. The most promising software development activity that may be used to develop an approach to evaluate software effectiveness is complete requirements traceability to determine requirements satisfaction.

The research and definition of software effectiveness satisfied the goal of the literature review, as well as the first and second objectives of the modified research plan, as outlined in Section 3.5. Next, the main thrust of the approach to evaluate software effectiveness is described below.

3.13 Software Effectiveness Evaluation Approach

During a visit to AFOTEC to clarify research objectives, the sponsor emphasized the importance of an early indicator of software effectiveness. An early indicator of software effectiveness serves many purposes. First, an early indicator provides information on the effectiveness of a particular piece of software *before* it gets to OT&E and in fact may serve as a qualifier to enter OT&E. Secondly, the early indicator helps

AFOTEC answer questions from Air Staff directed towards software effectiveness.

Lastly, an early indicator of software effectiveness is useful to many agencies involved in the software development process, including the personnel within the DT&E community, as well as the system program office (SPO).

AFOTEC approved of the effectiveness approach that was presented: to evaluate software effectiveness through requirements traceability. The traceability approach serves as an early indicator of software effectiveness, since traceability is maintained throughout the development of the software. Traceability also allows AFOTEC to maintain their independence during software development; AFOTEC does not dictate how something is to be developed, only that the developers maintain traceability throughout the effort.

The approach to evaluate software effectiveness through requirements traceability is described in detail in Chapter 4.

4. *Evaluating Software Effectiveness through Requirements Traceability*

4.1 *Introduction*

This chapter outlines an approach to evaluate software effectiveness using requirements traceability. Requirements traceability is the capacity to establish and monitor a connection between a software requirement and its counterparts in the software development process such as design, code, and tests. In this approach, assuming correct implementation of a software requirement, complete traceability of a requirement implies satisfaction of the requirement. This implication connects requirements traceability to the working definition of software effectiveness established in Chapter 3: the degree to which the software requirements are satisfactorily met. The software requirement must be traced through the design to the implementation, and the implementation must be tested to determine if the requirement has indeed been satisfied. In summary, software effectiveness is determined by the degree of complete traceability of: 1) the software requirements, through the design, to the code, and 2) from the software requirements to the tests that validate those requirements. Figure 1 portrays an abstract view of the various software development products and the degrees of traceability between them that are used to determine the software's effectiveness. The degrees of traceability, as well as their use in determining software effectiveness, are explained in the sections that follow.

Throughout this chapter, the focus is on the evaluation of software effectiveness. However, traceability is discussed beyond the connections between the *software* development products, and includes the system requirements. In addition, AFOTEC is

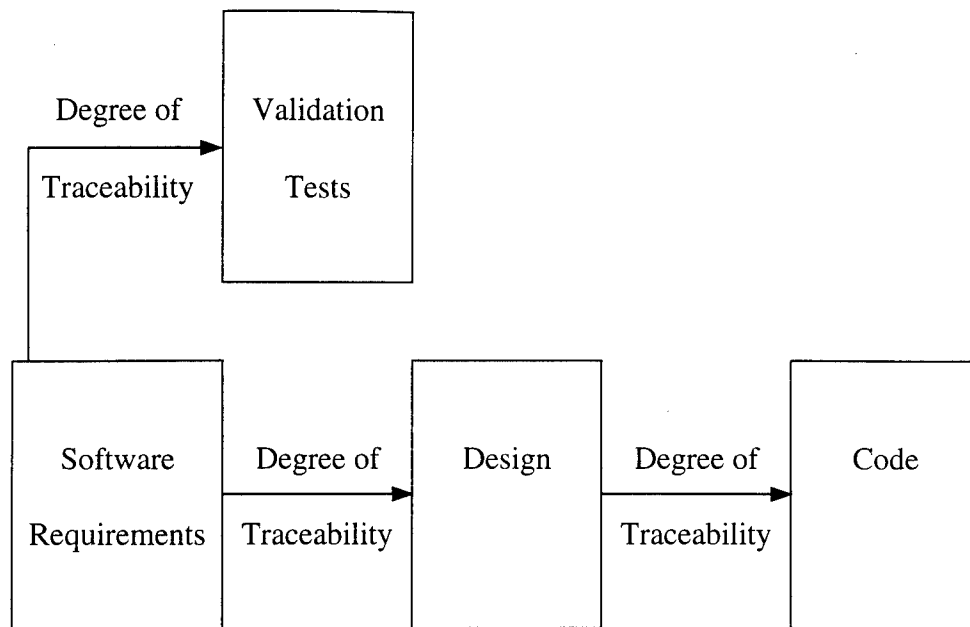


Figure 1. Abstract View of Degrees of Traceability

primarily concerned with traceability from the Operational Requirements Document (ORD), their governing document for OT&E. AFOTEC is also concerned with traceability from the code modules to the unit tests, since this traceability gives insight into the breadth of unit testing. Therefore, the “flow” of traceability in the sections that follow is from the ORD, to the system requirements, to the software requirements, to the design, and finally to the code. As described previously, traceability also exists from the software requirements to validation tests.

Traceability from the ORD and to the unit tests is included in this discussion in deference to AFOTEC. However, this additional traceability has nothing to do with the software effectiveness evaluation. The software effectiveness evaluation is only concerned with traceability and the following software development artifacts: software requirements, design, code and requirement validation tests.

After providing the definitions and assumptions that lay the groundwork to discuss the evaluation approach, the identification methods and logical structures used in the trace process are presented in Sections 4.4 and 4.5. Section 4.6 describes all the traces discussed in this research effort. Section 4.7 implements the traces in database form. After outlining the database implementation of the traces, Section 4.8 describes the operations that are used to initialize, populate, and extract information from the database. A method to calculate an overall effectiveness rating for the software being evaluated is outlined in Section 4.9. The final section, 4.10, provides a brief summary of the evaluation approach and the effectiveness calculation, and some concluding remarks.

4.2 Definitions

The following definitions are provided to facilitate the discussion of the effectiveness evaluation approach outlined in this chapter. Although some of these terms are generally understood in the software engineering community, subtle differences in their interpretation abound. The definitions are repeated here for clarity and to establish a foundation to discuss the effectiveness evaluation approach.

4.2.1 Requirements. Within this study, there many references to requirements. All references to requirements (whether from the ORD, system requirements, or software requirements), will either be identified explicitly or will be obvious from the context as to what type of requirement is being discussed. The ORD, system requirements, and software requirements are discussed below.

AFOTEC tests exclusively to the ORD to determine system effectiveness. The ORD is defined by AFOTEC as:

Operational Requirements Document. A document prepared by the respective using command describing the pertinent quantitative and qualitative performance, operation, and support parameters, characteristics, and requirements for a specific candidate weapon system. The ORD documents how a system will be operated, deployed, employed, and supported and provides initial guidance for the implementing, supporting, and participating command and agencies. The Air Force requires a mandatory attachment, called a requirements correlation matrix (RCM) [AFO95].

In other words, the ORD is a very high-level overview of system capabilities and characteristics. For example, the ORD for the C-17 aircraft includes the following capabilities: 1) “180° turn on a 90’ paved runway in approximately 3 maneuvers”, and 2) “airdrop 102 personnel” [Des95].

System requirements address *what* the system is supposed to do. The hardware and software requirements are determined from the system requirements. A system requirement is a capability or characteristic the system must possess upon development. System requirements specify the complete details of a system that will meet the operational requirements in the ORD.

The software requirements address *what* the software is supposed to do. A software requirement is a capability or characteristic the software must possess upon development. All the software requirements completely describe the necessary functions and attributes of the software product in its final form. There are many different kinds of software requirements that originate from many different sources. In this research, software requirements are classified into two different classes and four different types.

The two classifications of software requirements are functional and nonfunctional. Functional requirements are characterized by a capability the software must perform. Functional requirements are further identified as any software activity that takes input, processes data, or provides output. Put simply, a nonfunctional requirement is any requirement that is *not* a functional requirement. Nonfunctional requirements are often characterized by software attributes such as reliability, maintainability, and usability.

Software requirements are further divided into four different types: original, derived, interface, and constraint. Original requirements are the initial requirements defined by the user or generated by the refinement of system requirements into hardware and software requirements. Interface requirements describe the software's interaction with other software or hardware components, or human operators. Derived requirements are additional requirements that necessarily arise as a result of analysis of and/or development from the original requirements. For example, an original software requirement to calculate a position in three-dimensional space may result in a derived requirement for a specialized math function to multiply matrices. If the derived requirements are not satisfied, the original software requirements cannot be satisfied. Constraints are limiting factors on the software or its operating environment, such as programming language, program size, or a particular hardware platform required to run the software.

4.2.2 Design. The software design addresses *how* the software is going to do the tasks outlined by the software requirements. It is quite common to use the word "design" to describe the creative process as well as the document resulting from the process; such is the case in the use of the word "design" in this approach. The design is often

developed in several steps, with each step representing a further-refined and more detailed version of the previous step.

In this approach, the design process is divided into two subtasks: high-level design (HLD) and low-level design (LLD). The HLD defines the overall architecture, components, and interfaces of the software. The HLD is the “first cut” at describing how the software will meet the requirements. The LLD is a more detailed version of the HLD. The main purpose of the LLD is to refine and expand the HLD to the extent that the design is in sufficient detail to be implemented in a programming language. The LLD is the “final cut” at describing how the software will meet its requirements.

4.2.3 Code. The code refers to the implementation of the LLD elements in a programming language, and is also known as source code. This code consists of the computer instructions necessary to carry out the functions described in the requirements. This code does not include any other software necessary to create and support the executable software in its target environment such as batch files, command files, or data files. Typically, each LLD element is expressed at a level of detail such that it is directly implemented into one computer software unit (CSU) of code. CSUs are the smallest logical elements of code that perform a unique function or procedure and are separately executable and testable [DAF94]. CSUs with related functions are often grouped together, forming computer software components (CSCs).

4.2.4 Tests. In this approach, the primary purposes of software tests are to ensure: 1) the software requirements are satisfactorily met (validation tests), and 2) the code is robust (unit tests). Tests are conducted by executing the software under controlled conditions, observing the behavior of the software, and making an evaluation

about the software. Individual tests can be made up of several test cases which consist of inputs, execution conditions, and expected outputs.

4.2.5 Artifact. In common usage, an artifact refers to any product of the development process such as a system specification, software requirements specification (SRS) or design specification. Artifacts vary between different development processes, but typically contain complete information for one of these products; for the SRS, this information includes a unique identifier and a textual description for each software requirement.

In this study, the common definition of the term “artifact” is narrowed, referring *only* to the complete set of unique identifiers for the software elements within each product of the development process. For the purposes of traceability, there are six *software* artifacts: software requirements, HLD, LLD, code, validation tests and unit tests. For example, in regards to traceability, the LLD artifact contains only the unique identifiers for all the LLD elements for the software being developed.

There are two *system* artifacts discussed in this study: the ORD and the system requirements. These system artifacts and the unit tests (of the software artifacts) are used to provide the additional traceability requested by AFOTEC, but are not part of the software effectiveness evaluation.

4.2.6 Element. An element is a single item from the set of unique identifiers in an artifact, such as a particular system requirement, a specific validation test, or an individual component of the design. Each element of every artifact is uniquely identified within the development process. Elements may be decomposed into smaller subelements (that are also uniquely identified) for clarity.

4.2.7 *Trace*. By definition, a trace establishes a relationship between two or more elements of the development process. In most cases a trace records an element's transformation from one form to another, such as a software requirement to a high-level design element or a low-level design element to code. In all other cases the traces exist for coverage, such as making sure all requirements or all portions of code are traced to tests. In this study, traces between elements exist in only one direction. The elements of a trace are referenced in terms such as: "a trace exists *from* a software requirement *to* a validation test". For example, although there are traces between software requirements and validation tests, there are no traces from the validation tests to the requirements they satisfy. The directional connotation of the terms *from* and *to* is graphically represented by arrows in all subsequent figures.

4.2.8 *Structure*. A structure is a logical, graphical representation of an artifact and is created in this research for illustrative purposes only. Structures contain the elements of an artifact in hierarchical order. Arrows are drawn between elements within structures to graphically depict traces. Since a single element may be traced to one or more elements in another artifact, two types of arrows are used to represent a trace. A trace to a single element is represented by a line with a single arrowhead; a trace to multiple elements is represented by a line with multiple arrowheads. Both trace representations are shown in Figure 2.

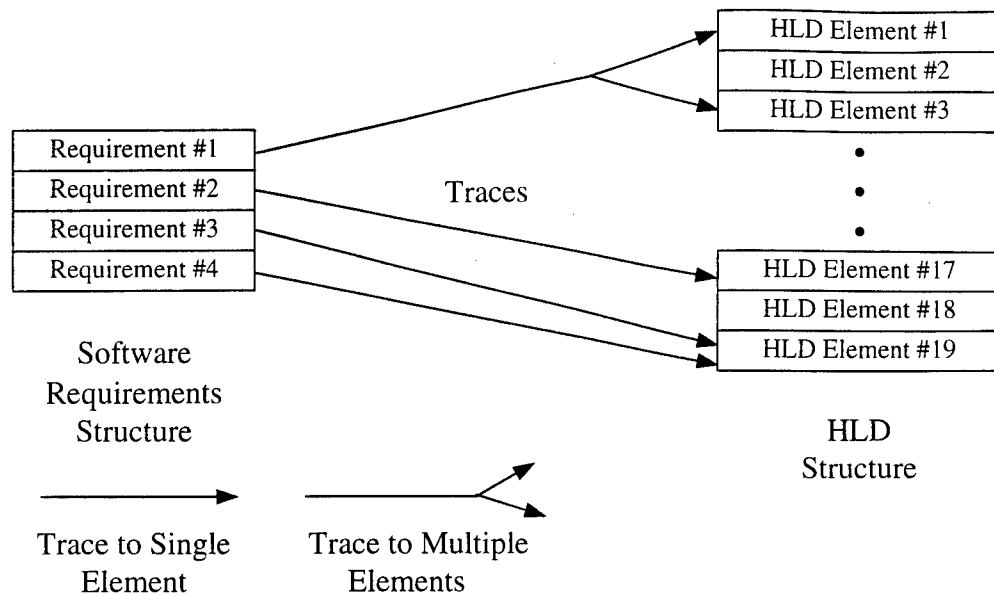


Figure 2. Example Structures with Traces

4.3 Assumptions

The approach to evaluate software effectiveness relies on the assumptions listed below. The assumptions focus on the artifacts and traces involved in the software effectiveness evaluation. No assumptions are made concerning the additional traceability requested by AFOTEC, and it is not implied that this traceability approach can be extended to evaluate *system* effectiveness.

1) Software requirements are complete and accurate.

By far the largest and most significant assumption, complete and accurate requirements ensure that the software has a firm foundation from which to make the effectiveness evaluation. Without this assumption, the evaluation may not represent the actual “effectiveness” of the software. Using incomplete or inaccurate requirements, the evaluation will not reflect the capabilities of the software in at least two scenarios: 1) The

software is rated as “highly” effective but is unsuitable for the user due to poorly defined requirements, even though the stated requirements are completely satisfied; 2) The software has a “low” effectiveness rating but it is unknown whether the inadequacies of the software stem from poorly defined requirements, a deficient software development process, or both.

2) Software requirements are quantifiable, measurable, and testable.

As stated previously, software effectiveness is based on the degree of satisfaction of the requirements. Assumption two facilitates the satisfaction evaluation by ensuring that requirements have specific criteria to determine satisfaction. Without this assumption, the software requirements cannot be assessed as to whether they are satisfied or not, severely degrading the effectiveness evaluation.

3) Decomposition of elements is complete and correct. This assumption is similar to the first assumption, concerning complete and correct software requirements. According to the first assumption, decomposition of requirements is complete and correct. The first assumption must be extended to the decomposition of all other development artifacts. Without the third assumption, “poor” software effectiveness could not be solely attributable to the lack of traceability, since the perceived lack of effectiveness may be due to inadequate decomposition of the design. The third assumption allows software effectiveness to be based *solely* on traceability of the artifacts throughout the development process.

4) Traces are complete and correct.

All defined traces must actually exist, and all traces are connected properly.

5) Complete traceability between elements implies that traced elements are complete and correct counterparts in the software development process.

This assumption is also similar to the first assumption concerning complete and accurate requirements. Since the effectiveness evaluation is measured by the degree of traceability of the requirements, the elements that are connected by the traces must be correct and complete counterparts to each other. E.g., for the traces that connect the software requirements to the design, it is assumed that each requirement is completely and correctly addressed by the design element or elements the requirement is traced to. In other words, the presence of the trace indicates that the requirement has been successfully implemented in the design, i.e., the design correctly and completely satisfies the requirement.

6) The ability to trace software requirements is in place from the onset of software development, and the traces are updated as software is developed, used, and maintained.

The ability to trace requirements must be considered an integral part of the software development plan. It is necessary to document traceability from the first stages of software development since it is virtually impossible to “catch up” once the traces are not kept current. Documenting traceability cannot be an add-on, an afterthought, or a separate activity considered unrelated to software development.

The effectiveness measurement is intended to be used during development as a “snapshot” of current progress and throughout the software’s use to aid in software maintenance. It is vital that the traces between the various elements of software artifacts

are updated as the software is developed, used, and maintained. If the traces are not updated, any evaluation of the software that is based on the traces will be inaccurate, since the outdated traces will not represent the current state of the software. In essence, the effectiveness evaluation is only as accurate as the most recently documented traces between software elements.

4.4 Unique Identification of Elements

All elements are uniquely identified using character codes and a nested numbering scheme. This identification method uses character codes for two primary reasons: 1) to identify from which artifact the element comes and, 2) to distinguish the different types of elements within an artifact. For example, “RSYS” may be used to identify system requirements; within the software requirements, “RO” and “RD” may be used to respectively identify original and derived requirements.

In addition to the character codes, this identification method uses a numerical nesting notation to organize the different elements as they are decomposed into more explicit detail. For example, the software requirement RSW - 12 may be decomposed into subrequirements RSW - 12.1, RSW - 12.2, and RSW - 12.3. This decomposition may continue as necessary, with RSW - 12.1 broken down further into RSW - 12.1.1, RSW - 12.1.2, etc. Refer to Figure 3 for clarification of the identification and decomposition of elements.

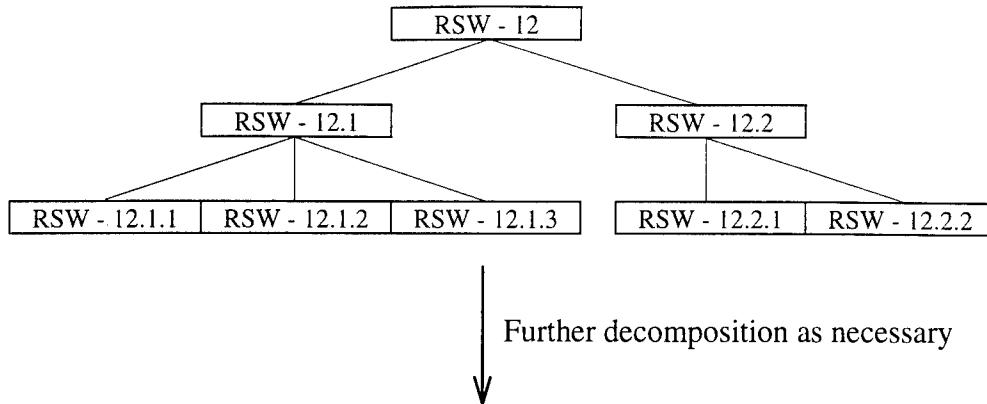


Figure 3. Example of Element Decomposition and Identification Method

The identification method was developed for three purposes, which are listed below and shown in Figure 4:

- 1) to document the decomposition of elements within the structure
- 2) to document traces between elements
- 3) to connect the element back to the artifact that contains its textual description

Note that Figure 4 shows decomposition of the elements by indenting subelements within the structure. Element decomposition will be illustrated within the structures in this manner with all the remaining examples.

It should be noted that all the character codes for elements described in this study were chosen arbitrarily and are *for illustrative purposes only*. In actual practice, software developers will certainly use codes unique to their development process. For example, a software developer may use design element codes that are assigned from each functional area or design team. Similarly, requirements may be identified by functional area or analysis team. The character codes used to identify the different elements in this study are described below.

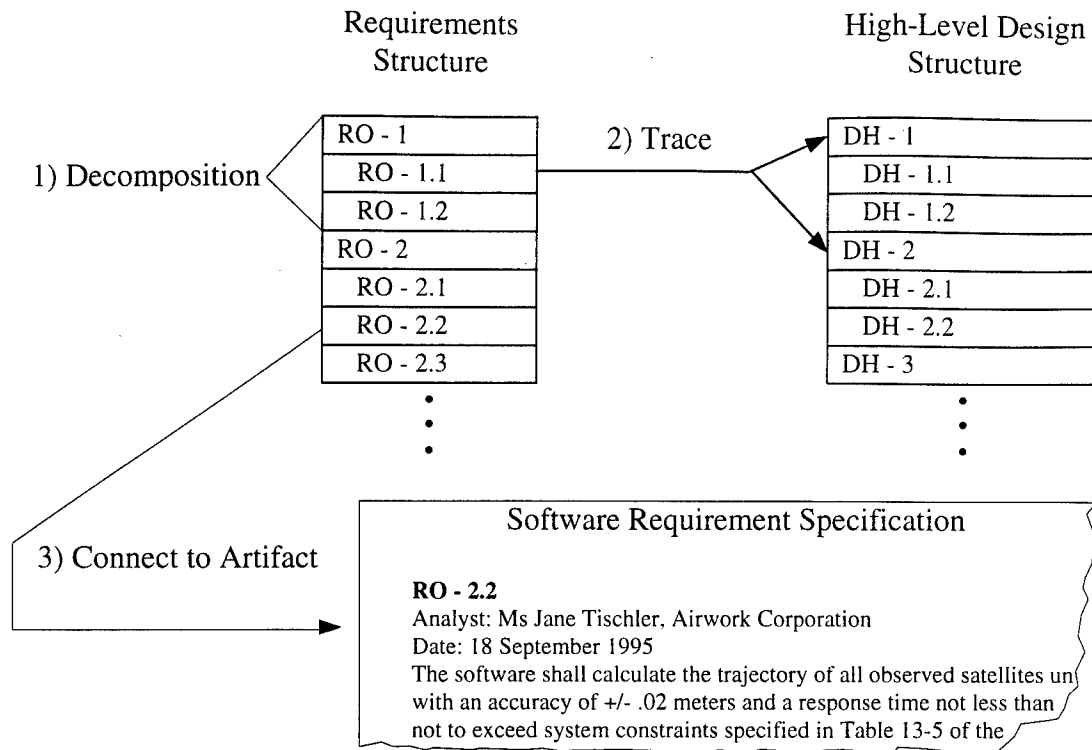


Figure 4. Three Purposes of the Element Identification Method

4.4.1 ORD Elements. The capabilities and characteristics of the system that are listed in the ORD are identified by the following code and the nested numbering scheme as described above:

ORD - system capability or characteristic from the ORD

It should be noted that the actual ORD for the C-17 has a slightly different notation to identify the system capabilities, using lower case letters for decomposition. For consistency, this method will be translated into the identification method used in this study. For example, a capability described in ORD paragraph 4d(3)(a) will be identified as ORD - 4.4.3.1, substituting the letter's position number in the alphabet for the letter identifying the ORD subparagraph.

4.4.2 *System Requirements.* Detailed system requirements are identified by the following code and the nested numbering scheme as described above:

RSYS - system requirement

4.4.3 *Software Requirements.* Software requirements are identified by any of following codes and the nested numbering scheme as described above:

RO - original requirement

RD - derived requirement

RC - constraint requirement

RI - interface requirement

4.4.4 *Design Elements.* Both HLD and LLD elements are annotated with a code and the same nested numbering system as described above. Although these software elements represent different software artifacts and are contained in different structures for traceability illustrations, both types of design elements are described here because of their close relation to each other. The design element codes are described as follows:

DH - high-level design

DL - low-level design

4.4.5 *Code Elements.* Source code elements are identified by either of the following codes and the same nested numbering system as described above. The source code elements are labeled with the standard military software development notation (CSC, CSU). The source code element identifiers are described as follows:

CSC - computer software component

CSU - computer software unit

The computer software configuration item (CSCI) identifier is not used to describe elements of code. The level of detail of a CSCI (in code) is comparable to the lowest-level element of the HLD and the upper-level elements of the LLD, and actually overlaps the design. Perhaps a different approach may decompose the LLD elements into CSCIs and then trace the CSCIs to CSCs and CSUs. In this approach, the term CSCI is omitted for clarity, and the lowest-level LLD elements are traced directly to the CSCs or CSUs.

4.4.6 Tests. The tests are identified by either of the following codes and the same nested numbering system as described above. Although these software elements represent different software artifacts and are contained in different structures for traceability illustrations, both types of design elements are described together here because of their close relation to each other. The test identifiers are described as follows:

- TR - validation test to ensure requirement is satisfactorily met
- TU - unit test for one CSU

4.5 Structures Used for Traceability

As stated previously, a structure is a logical representation of an artifact and is created in this research for illustrative purposes only. Structures contain the unique identifiers of the elements that make up the artifacts of the development process. The structures used to illustrate traceability are described below.

4.5.1 ORD Structure. If necessary, capabilities and characteristics in uniquely identified paragraphs in the ORD are decomposed into individual capabilities and characteristics. For example, if a paragraph in the ORD contains two or more capabilities that the system in development must possess, these capabilities can be identified using the

nested numbering scheme within the ORD structure. Refer to Figure 5 for an example section of the ORD structure.

•
•
•
ORD - 3.4.2
ORD - 3.4.2.1
ORD - 3.4.2.2
ORD - 3.4.3
ORD - 3.4.4
ORD - 3.4.4.1
ORD - 3.4.4.2
ORD - 3.4.4.3
•
•
•

Figure 5. Example Section of ORD Structure

4.5.2 System Requirements Structure. If necessary, system requirements are decomposed into further detail and are contained in a structure similar to Figure 5. Subsystem requirements are indented below the system requirement they are decomposed from. System requirements are decomposed into sufficient detail to distinguish software requirements and hardware requirements.

4.5.3 Software Requirements Structure. Software requirements are decomposed, if necessary, into further detail and are contained in a structure similar to Figure 5, with subrequirements indented below the software requirement they are decomposed from. Software requirements are decomposed into sufficient detail to be addressed by the software design.

4.5.4 High-Level Design Structure. The HLD elements capture the “big picture” of the software’s architecture and are contained in a structure similar to Figure 5. If necessary, the HLD elements are decomposed into subdesign elements for clarity. This

decomposition should not go into detail beyond the complexity level of a CSCI. A CSCI is described as performing a “common end-use function” and, in implementation, may “contain 100,000 lines-of-code” or more [DAF94].

4.5.5 Low-Level Design Structure. The LLD is intended to capture all the explicit details necessary to “flesh out” the lowest-level HLD elements and are contained in a structure similar to Figure 5. If necessary, the LLD elements are decomposed into subdesign elements for clarity. This decomposition should continue until subdesign elements consist of significant detail to be implemented in a programming language. The complexity of the lowest-level LLD element should correspond to a CSU, the LLD element’s counterpart in code.

4.5.6 Validation Tests Structure. This structure contains the tests that ensure the requirements have been satisfactorily met by the software and are contained in a structure similar to Figure 5. Tests can be decomposed into further detail such as input, execution conditions, and expected output. For example, a validation test TR - 3 that is created to ensure requirement RO - 3.1 is satisfied may be decomposed into five test cases, TR - 3.1 through TR - 3.5. Each test case of TR - 3 may be decomposed further into unique identifiers for input, execution conditions, and expected output, with respective identifiers: TR - 3.1.1, TR - 3.1.2, TR - 3.1.3, TR - 3.2.1, etc.

4.5.7 Code Structure. Software code is often modularized into manageable units. These software units, known as CSUs, are the implementation of the LLD elements and are contained in a structure similar to Figure 5. CSUs are not decomposed further, since they perform a single function and represent the smallest unit of compilable, executable

code [DAF94]. However, to group similar functions, this approach allows several LLD elements to be contained in one CSC, then the CSC is decomposed into CSUs.

4.5.8 Unit Tests Structure. This structure contains the tests that ensure the individual CSUs have been tested at the unit level, and are contained in a structure similar to Figure 5. “Proof” of robustness for code of substantial complexity is impossible; however, the purpose of these tests is to ensure (to the greatest degree possible) that the code is robust. The unit tests can be decomposed into further detail such as input, execution conditions, and expected output, as in the decomposition of the validation tests.

4.5.9 Summary. The structures described above only exist conceptually. Each structure contains the unique element identifiers for the development artifact it represents, such as the software requirements or the validation tests. These structures provide the basis from which connections can be made to other artifact structures using the traces described below. Traces are made between individual elements of specific structures to establish a relationship between the elements.

4.6 Trace Descriptions

4.6.1 Introduction. Section 4.6 describes all the traces that connect the development artifacts, such as the traces from the software requirements to the HLD elements. The traces between the elements in the structures that represent the development artifacts are the crux of the software effectiveness evaluation. For the software effectiveness evaluation, the traces establish a connection from the needs and expectations of the user (the software requirements) to the implementation (the code). In

addition, the traces establish a connection from the software requirements to their validation tests for the software effectiveness evaluation.

In this study, a trace is always established *from* a single element in a structure. As shown in Figure 2, a trace may be connected *to* a single element or multiple elements. Whether a trace is connected to a single element or multiple elements, it is still referred to as *a trace* (singular). A trace from a single element permits the familiar 1 to 1 and 1 to n relationships. To allow m to 1 and m to n relationships, *multiple* traces are used. For example, multiple software requirements may be addressed by a single HLD element or several requirements may be validated by many different tests.

Traces are established *from* a single, decomposed lowest-level element in a structure *to* the highest-level element(s) of the structure *being traced to*. If necessary, these high-level element(s) are then decomposed to their lowest level of detail, and *these* lowest-level elements are then traced to the highest-level elements of the next structure *being traced to*. This tracing then continues to the next appropriate development structure. The exception to this “lowest-to-highest” rule occurs in the traces from the LLD to the code; traces are established between the lowest-level LLD elements and the CSUs (which are already at their lowest level and not decomposed any further). The traces between the different structures are described below and displayed in simplified examples in Figures 6 through 12. These figures are simplified for illustrative purposes; the fact that element identification numbers are similar in some cases, such as ORD - 1.2.1 tracing to RSYS - 1 in Figure 6, does not imply that there must be a correlation in identification numbers between traced elements.

Complete traceability is the goal; in this study, completely traced software requirements mean all the software requirements are satisfied and the software is effective. Therefore, the degree of traceability of the software requirements throughout the software development artifacts determines, by definition, the degree of software effectiveness.

4.6.2 ORD to System Requirements. These traces connect the capabilities and characteristics described in the ORD to the requirements of a system that will accomplish these capabilities and characteristics. An example of ORD to system requirements traces is shown in Figure 6.

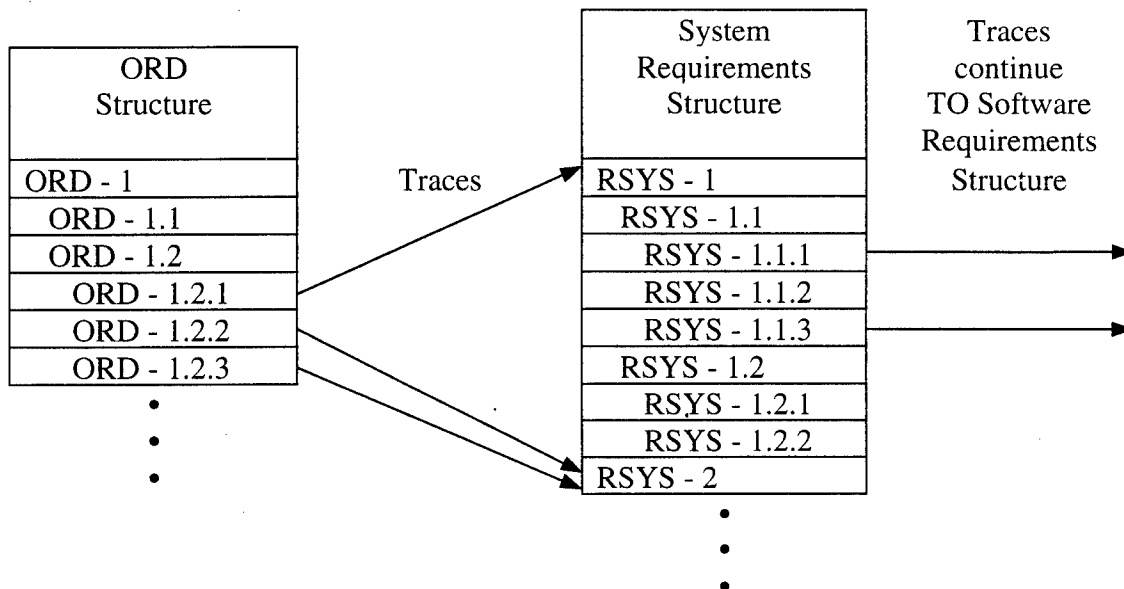


Figure 6. Example of ORD to System Requirements Traces

4.6.3 System Requirements to Software Requirements. As system requirements are decomposed into further detail, the system engineer determines which components of the system will be addressed by hardware and which will be addressed by software.

These traces connect the lowest-level decomposed system requirements to the highest-level software requirements. An example of system requirements to software requirements traces is shown in Figure 7.

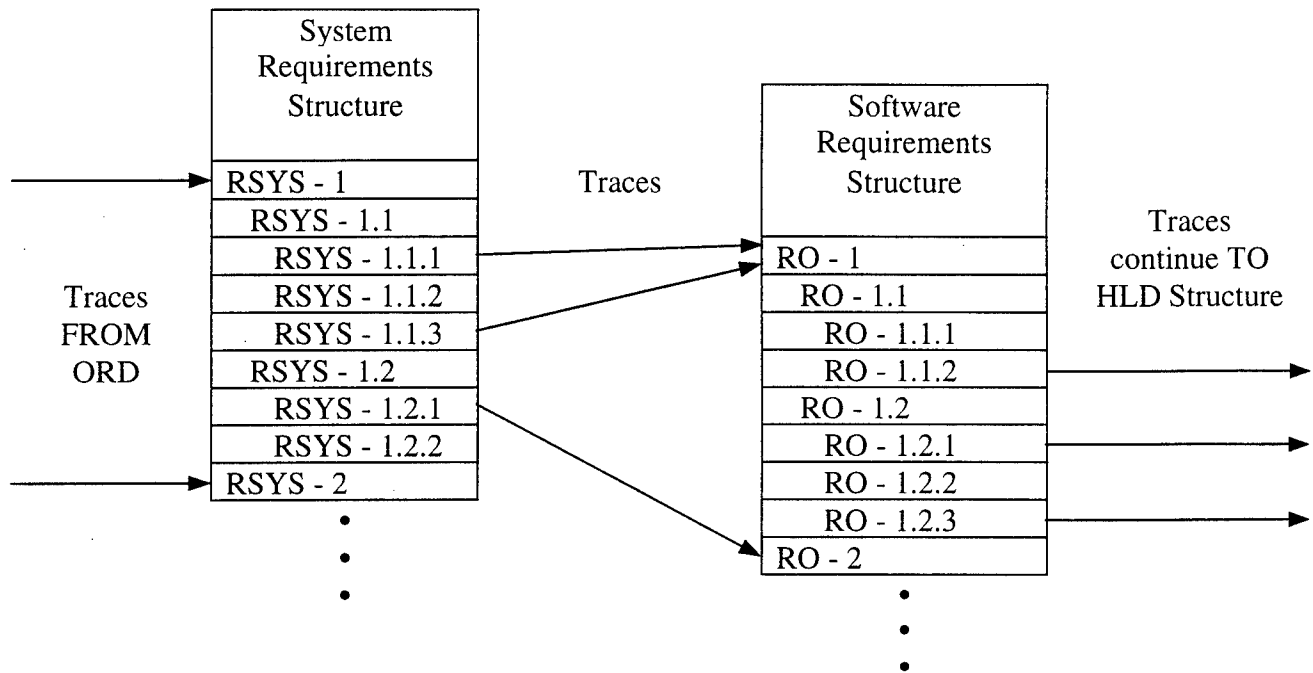


Figure 7. Example of System Requirements to Software Requirements Traces

4.6.4 Software Requirements to High-Level Design. These traces connect the lowest-level decomposed software requirements to the highest-level HLD elements. An example of software requirements to HLD traces is shown in Figure 8.

4.6.5 High-Level Design to Low-Level Design. These traces connect the decomposed lowest-level HLD elements to highest-level LLD elements. An example of HLD to LLD traces is shown in Figure 9.

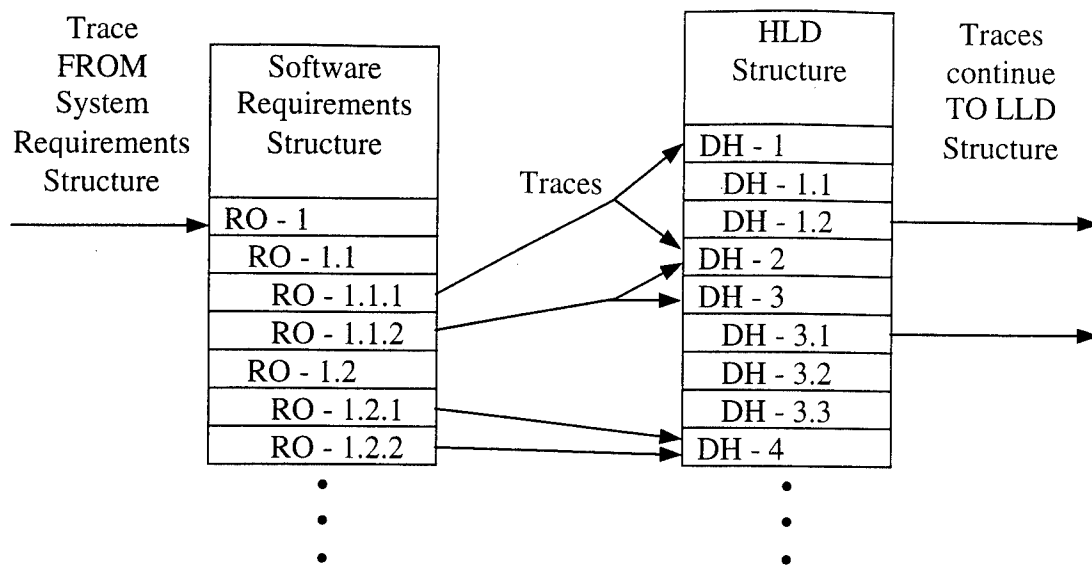


Figure 8. Example of Software Requirements to HLD Traces

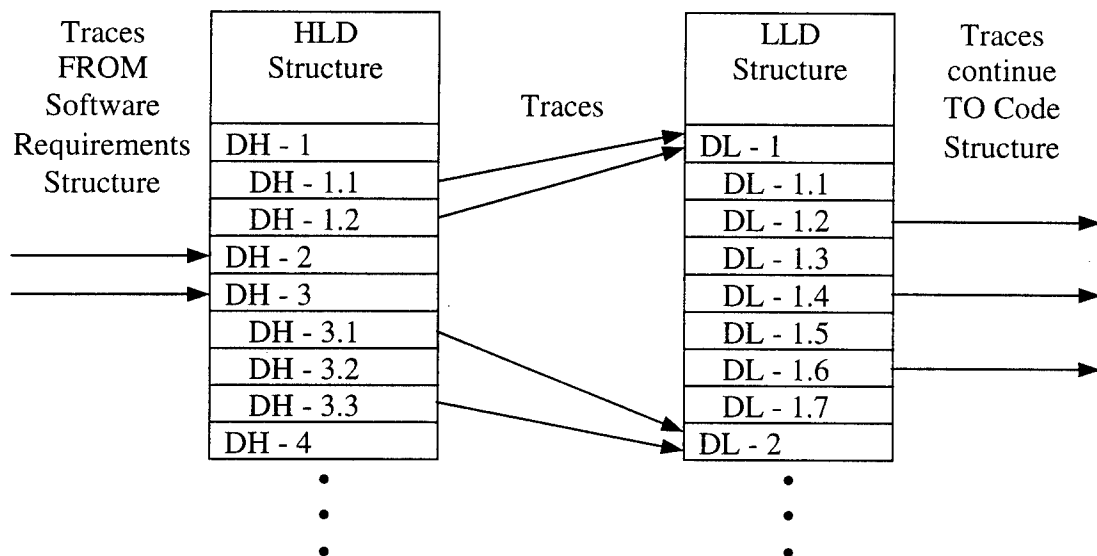


Figure 9. Example of HLD to LLD Traces

4.6.6 Low-Level Design to Code. These traces connect the lowest-level LLD elements to the code elements, either CSCs or CSUs. LLD elements are decomposed into explicit detail with complexity similar to a CSU, and are often traced directly to a CSU.

However, similar functions represented by LLD elements may be combined and traced to a CSC, *then* decomposed into CSUs. An example of LLD to code traces is shown in Figure 10.

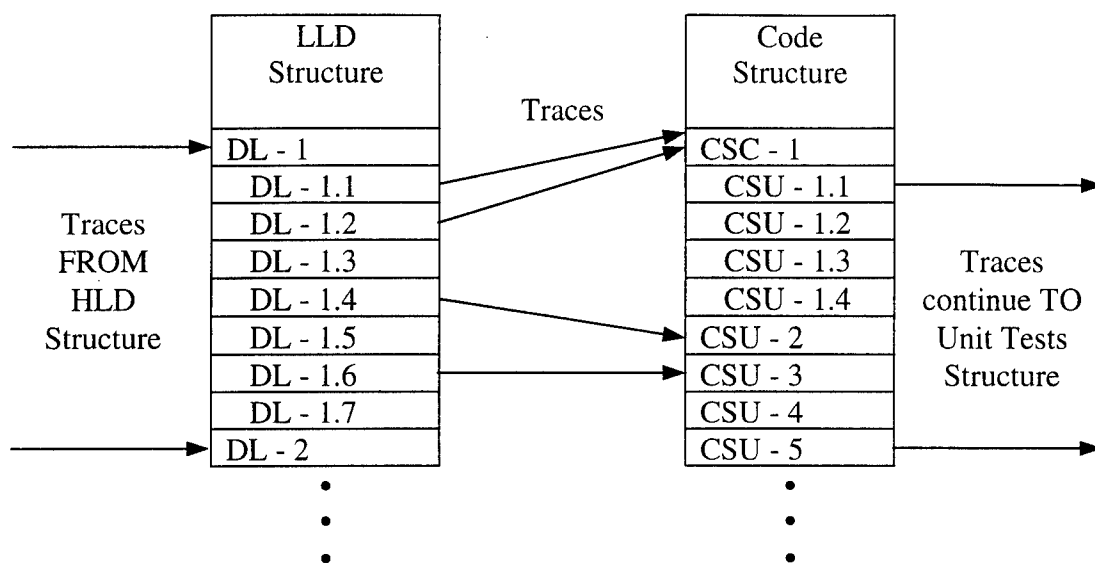


Figure 10. Example LLD to Code Traces

4.6.7 Software Requirements to Validation Tests. These traces connect the software requirements to the validation tests, as defined in Section 4.2.4, that show the requirements have been satisfactorily met. An example of software requirements to validation tests traces is shown in Figure 11.

4.6.8 Code to Unit Tests. These traces connect the lowest-level code elements (CSUs) to unit tests that “exercise” the code and, as described in Section 4.5.8, ensure (to the greatest degree possible) that the code is robust. An example of code to unit test traces is shown in Figure 12.

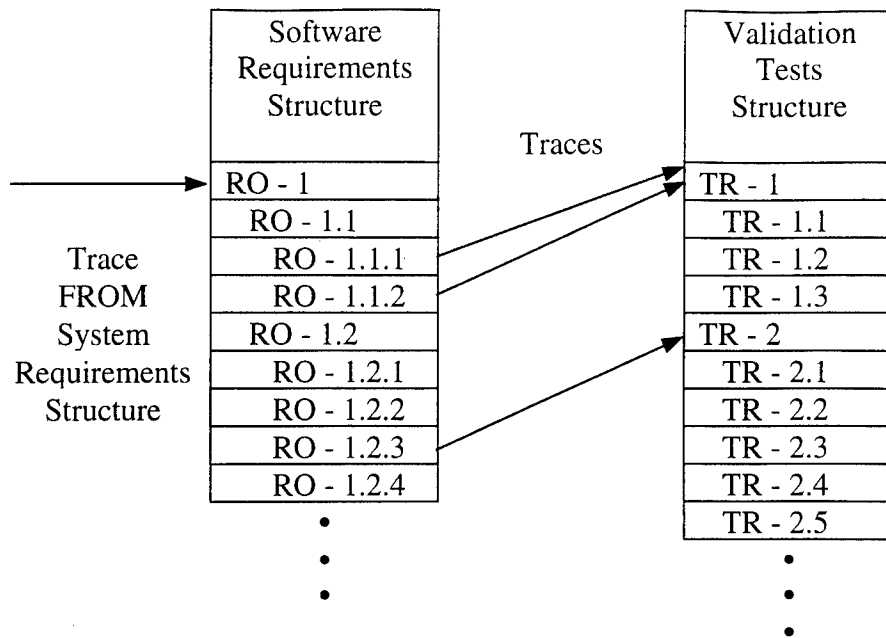


Figure 11. Example of Software Requirements to Validation Tests Traces

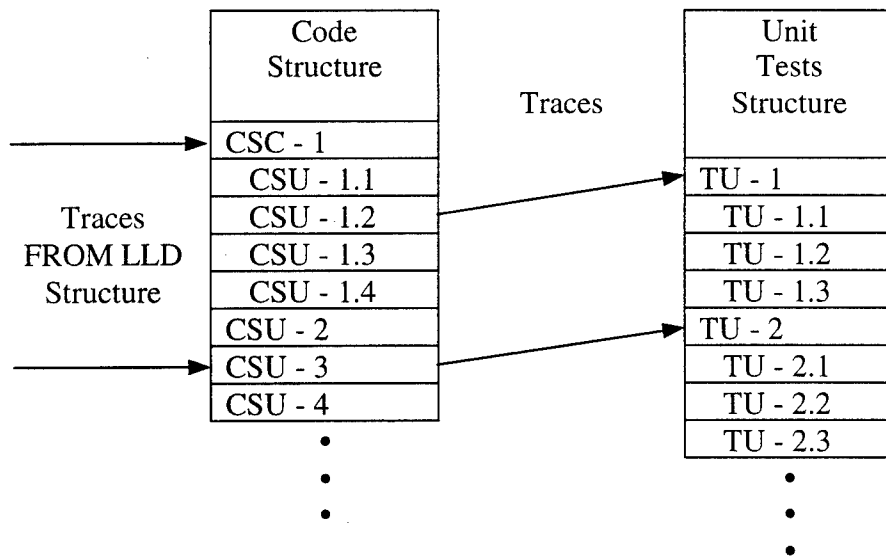


Figure 12. Example of Code to Unit Tests Traces

4.6.9 *Summary.* The traces described above provide the connection between elements of specific development artifacts. The unique identifiers from the artifacts are

contained in the logical structures described in Section 4.5. The traces exist between the elements contained in the structures for one of two reasons:

- 1) To track elements during the development process, such as the traces that connect the software requirements to the HLD elements.
- 2) For accountability, i.e. coverage, to ensure *all* software requirements are traced to validation tests and *all* CSUs are traced to unit tests.

It should be noted that there are no traces between ORD capabilities and tests and system requirements and tests. These traces may be appropriate in another evaluation approach, but they are unnecessary in the software effectiveness evaluation outlined in this research.

So far, structures have been described as existing logically, containing the unique element identifiers from the artifacts of the development process. The traces have existed logically also, represented in figures as arrows connecting elements contained in the structures. The structures and arrows in this study were created for illustrative purposes, to explain traceability between the artifacts of the development process, and the illustration ends here. There will be no more discussion of tracing “the unique element identifiers contained in the structures that represent the development artifacts”. Traces exist between artifact elements, and will be discussed in this manner from this point on.

To shift from the logical world of structures (for artifacts) and arrows (for traces) to the real world, artifacts and traces are implemented in a database in the following section. All of the traces described in Section 4.6 are included in the database described below.

4.7 Implementing Traces in a Database

For clarification, the “implementation” described in this section shows how the traces previously described can be put into practical use or fulfilled by a database. The database approach described in this section is an example implementation and is not a “database” in the standard sense, but an abstract organization of information; in this case, the information is the decomposition information and the trace information. In addition, this implementation *may* be compared to the standard definition of a relational database, in that there are two types of relations: decomposition relations and trace relations. Finally, this “database implementation” should not be taken as a model of efficiency or optimization. There are dozens of commercial products available that allow the storage and retrieval of traceability information, many of which have been evaluated by the Air Force’s Software Technology Support Center [DAF95]; this example is given only to provide some insight into how the traceability information that is mentioned above can be documented and used.

4.7.1 Introduction. Section 4.7 outlines an implementation to document, then quantify, the traces between artifacts of the development process. The traces are implemented in a database, and traceability information is obtained from searching the database. The trace database, hereafter referred to as the database, contains the elements for all the artifacts and all the traces between the elements. The database also contains all the decomposition information for the highest to lowest-level elements within each artifact.

It is important to note that although the decomposition information (via the numbering scheme) for the elements is contained in the database, these relationships are *not* considered traces. In short, elements are decomposed *within* artifacts and are traced *between* artifacts. Traces are established *from* the lowest-level elements of one artifact *to* the highest-level elements of the next artifact in the development process. For example, lowest-level software requirement RO - 2.2.3 is traced to highest-level HLD element DH - 2. These highest-level elements are eventually decomposed to *their* lowest-level elements, where they are traced to the next artifact in the development process.

Element decomposition *could* be followed just by using the nested numbering scheme described previously, thus leaving the decomposition information out of the database entirely. However, just using the numbering scheme to keep track of element decomposition creates a dependency that inhibits reuse. For example, suppose a design element can be decomposed into subdesign elements that are available from a reuse library. The design element identifiers from the reuse library can be stored in the database as decomposition information, even though the identification numbers are unrelated to the design elements they are “decomposed” from. By keeping the decomposition information in the database, element reuse is possible, and traces are not “lost” within the artifact as elements are decomposed.

In summary, *all* the elements for *all* the artifacts and *all* their respective traces are stored in the database, as well as each element’s decomposition information; however, decomposition information is not to be confused with trace information.

4.7.2 General Forms for Database Entries. There are two types of database entries, and each type of entry has three fields for data. One type of database entry

contains the element decomposition information, with one entry for each level of decomposition. For example, one database entry would be needed to decompose software requirement RO - 1 into RO - 1.1, RO - 1.2, and RO - 1.3. The other type of database entry contains trace information, with one entry for each trace, as in software requirement RO - 1.2.1 being traced to HLD element DH - 1.

The format for the element decomposition entry is shown in Figure 13.

(1)	(2)	(3)
Element to be Decomposed	Complete Flag	List of Subelements

Figure 13. Format of Database Entry for Element Decomposition

The three attribute fields, referred to as columns (1), (2), and (3) in Figure 13, are defined as follows:

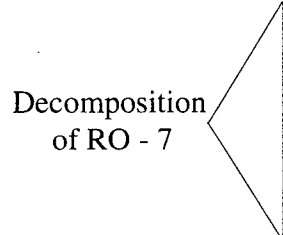
(1) Element to be Decomposed. This element cannot be directly traced to another element in another artifact; therefore, it must be decomposed into subelements. The element in column (1) is decomposed into two or more subelements that will be listed in column (3) of the same entry.

(2) Complete Decomposition Flag. The number in this field is either a one (complete decomposition) or a zero (incomplete decomposition), indicating whether the element in column (1) has been completely decomposed. In this study, since all decompositions are assumed to be complete and correct, this field will *always* be set to 1. In a situation where not all elements are decomposed completely, as in the development

of new software, this field can be used to monitor the decomposition of elements and artifacts.

(3) List of Subelements. This field contains the list of subelements that are decomposed from the element in column (1) of the same entry.

An example section of decomposition database entries is shown in Figure 14.



RO - 7	1	RO - 7.1 RO - 7.2
RO - 7.1	1	RO - 7.1.1 RO - 7.1.2 RO - 7.1.3
RO - 7.2	1	RO - 7.2.1 RO - 7.2.2
RO - 8	1	RO - 8.1 RO - 8.2 RO - 8.3 RO - 8.4

Figure 14. Example Database Entries Containing Decomposition Information

In this figure, RO - 7 is decomposed into its lowest-level elements, followed by the initial decomposition of RO - 8.

The second type of database entry contains trace information; the format for the trace information entry is shown in Figure 15.

(1)	(2)	(3)
Element Traced FROM	Complete Flag	Element(s) Traced TO

Figure 15. Format of Database Entry for Trace Information

The three attribute fields, referred to as columns (1), (2), and (3) in Figure 15, are defined as follows:

(1) Element Traced FROM. The element in this field is the origin of the trace for this database entry. For example, in a software requirement to HLD element trace, the trace is established *from* the software requirement *to* the HLD element(s). In this example, the software requirement identifier would be entered in column (1).

(2) Complete Trace Flag. The number in this field is either 1 (complete trace) or 0 (incomplete trace), indicating whether the element in column (1) has been completely traced to the element(s) in column (3). The field is set to 0 during database initialization and is only set to 1 when the element or elements that constitute a complete trace for this entry are listed in column (3). (Operations on the database, including initialization and maintenance are discussed in greater detail in Section 4.9). Once again, a “complete” trace connects one element to one or more elements in another artifact; this connection wholly and succinctly satisfies the element being traced *from* to the element(s) being traced *to*.

(3) Element(s) Traced TO. The element or elements in this field are the target of the trace for this database entry. For example, in a software requirement to HLD element trace, the trace is established *from* the software requirement *to* the HLD element(s). In this example, the HLD element(s) would be entered in column (3).

In certain database entries, this column contains “N/A”, which stands for “not applicable”. The most common use of N/A in column (3) occurs when, by definition, no

trace exists from the lowest-level decomposed element in column (1). For example, once the tests (both validation tests and unit tests) are decomposed into their lowest-level elements, there is no further decomposition or tracing in this implementation. Test elements are not traced *to* any other development artifacts; therefore, N/A is entered in column (3) for these trace entries.

The other (much less frequent) use of N/A in column (3) occurs when *no trace logically exists* from the element in column (1). For example, suppose there is a software requirement that states the software must be coded in the Ada programming language. This software requirement has no logical trace to any HLD element; therefore, N/A is entered in column (3) for this trace entry. In these rare instances, traceability begins in the ORD and stops before being completely traced through to the code. N/A is entered in column (3) for this trace entry so as to distinguish such a situation from an incomplete trace. It is important to point out that N/A is *never* entered in column (3) of the element decomposition entries; N/A is only entered in column (3) of the trace entries, and only when necessary.

An example section of database entries containing trace information is shown in Figure 16. In this figure, the lowest-level elements of RO - 7 are traced to the highest-level elements of the HLD.

In the database, the two types of entries are distinguished by the element identifiers in columns (1) and (3). If columns (1) and (3) contain identifiers from the *same* artifact, the database entry contains decomposition information. If columns (1) and (3) contain identifiers from *different* artifacts, the database entry contains a trace.

Software
Requirements
Traced to HLD
Elements

RO - 7.1.1	1	DH - 6 DH - 7
RO - 7.1.2	1	DH - 7
RO - 7.1.3	1	DH - 6
RO - 7.2.1	1	DH - 7 DH - 8
RO - 7.2.2	0	

•
•
•
•
•
•
•

Figure 16. Example Database Section Containing Trace Information

In the database, the two types of entries are distinguished by the element identifiers in columns (1) and (3). If columns (1) and (3) contain identifiers from the *same* artifact, the database entry contains decomposition information. If columns (1) and (3) contain identifiers from *different* artifacts, the database entry contains a trace.

There is a problem distinguishing the two types of entries if column (3) is blank. During development, if an element in column (1) has not been decomposed or traced from yet, column (3) will be blank. Consider RO - 7.2.2 in Figure 16. At this point in development, it is impossible to determine whether RO - 7.2.2 requires further decomposition or a trace established from it. In this study, this problem is remedied by the third assumption; all element decomposition is assumed to be complete and correct, so RO - 7.2.2 *must* be a trace entry that has not been completed yet. In practice, without this assumption, there is still a problem distinguishing the two types of entries if column (3) is blank. This problem must be addressed by the implementors of the approach, who may rely on an assumption also, i.e., RO - 7.2.2 in Figure 16 is assumed to be decomposed to its lowest level, and needs trace information entered in column (3).

An example section of the database is shown in Figure 17 and displays the database entry format information discussed so far. This figure contains the

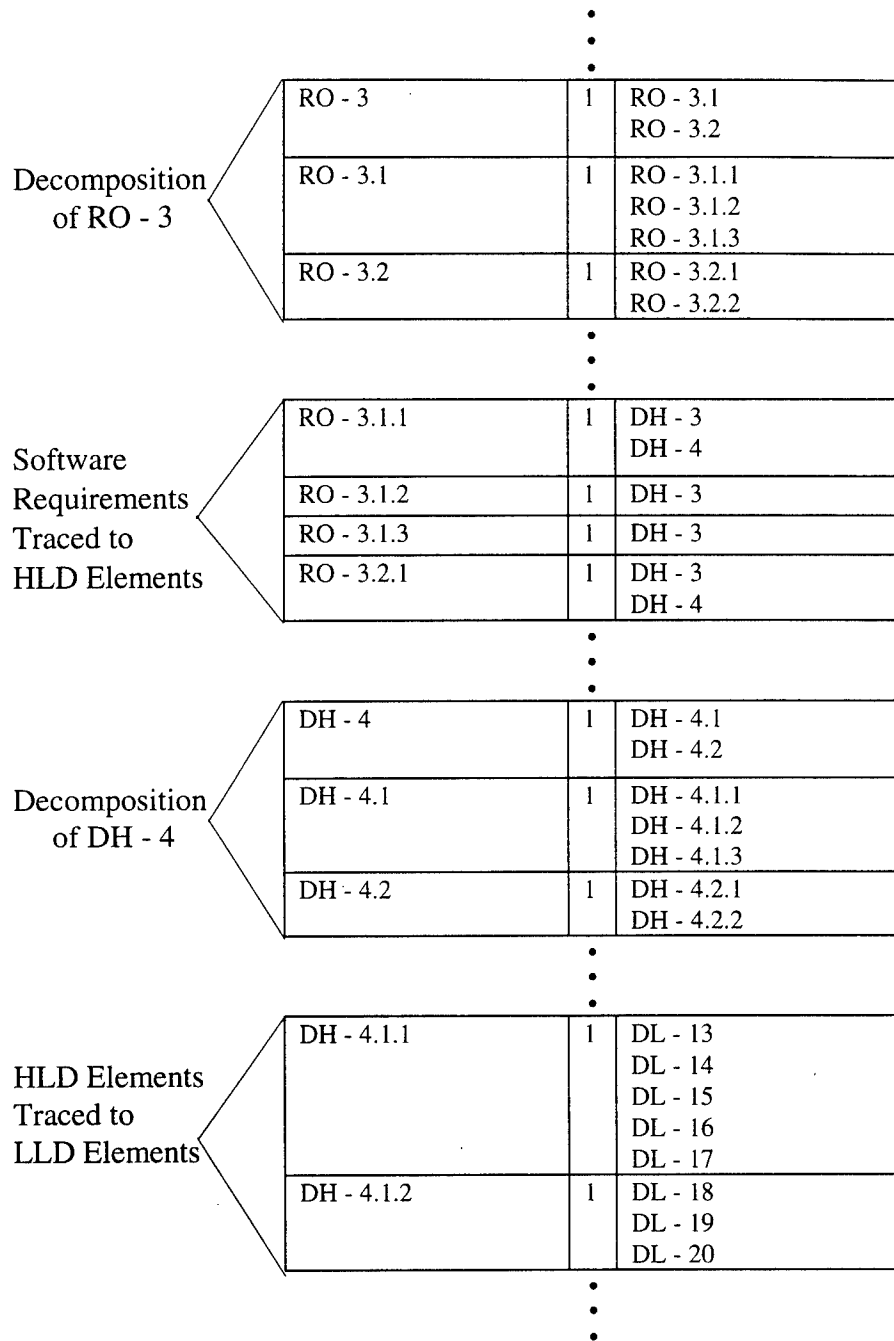


Figure 17. Example Database Section

decomposition of RO - 3 into its lowest-level elements. Then, these lowest-level software requirements are traced to HLD elements. Next, the HLD elements are decomposed into their lowest-level elements. Finally, the lowest-level HLD elements are traced to LLD elements. The database section in Figure 17 shows one portion of the entire database, which contains decomposition and trace information from the ORD, eventually to the software requirements, and from the LLD elements to the unit tests.

4.8 Database Operations

4.8.1 Introduction. The entire database consists of six logical “sections”, with each section containing the element decomposition information for one development artifact, as well as the traces *from* those elements. The sections are logically arranged in the database according to Figure 18, with the sections numbered 1) through 6). This arrangement serves three purposes:

- 1) It divides the database into logical sections based on the development artifact.
- 2) From the top down, each section “flows” (tracewise) into the next section.
- 3) The ordered arrangement allows for faster database searches.

Now that the format of the database entries and the logical arrangement of the database sections have been explained, the actual operations on the database are described in the sections that follow.

1)	Decomposition of ORD
	Traces from ORD to System Requirements
2)	Decomposition of System Requirements
	Traces from System Requirements to Software Requirements
3)	Decomposition of Software Requirements
	Traces from Software Requirements to Validation Tests
	Decomposition of Validation Tests
	Traces from Software Requirements to HLD
4)	Decomposition of HLD
	Traces from HLD to LLD
5)	Decomposition of LLD
	Traces from LLD to Code
6)	Decomposition of Code
	Traces from Code to Unit Tests
	Decomposition of Unit Tests

Figure 18. Logical Arrangement of Database Sections

4.8.2 Database Initialization. In this implementation, database initialization results in the database containing the decomposed ORD. The intent is to “grow” the database from this point; trace the lowest-level ORD elements to system requirements, decompose the system requirements, trace the lowest-level system requirements to software requirements, etc. Figure 19 shows a sample section of the initialized database, with the completely decomposed ORD elements and the lowest-level ORD elements (ORD - 1.1.1.1.1 through ORD - 1.1.1.1.4) not yet traced to system requirements.

4.8.3 Data Entry. Element decomposition data and trace data are entered into the database in an ongoing process, beginning with the onset of system development and ending with the system’s retirement. Therefore, the database should be considered a “living” thing, as important to the final software product as the development process

itself. The database in this implementation provides a connection from capabilities specified in the ORD to their appearance in the final product. This data must be maintained as requirements change (and requirements *will* change), or the database will quickly become outdated and useless. As stated earlier, the database “grows” from the ORD; this is accomplished by adding the traces from the ORD to the system requirements.

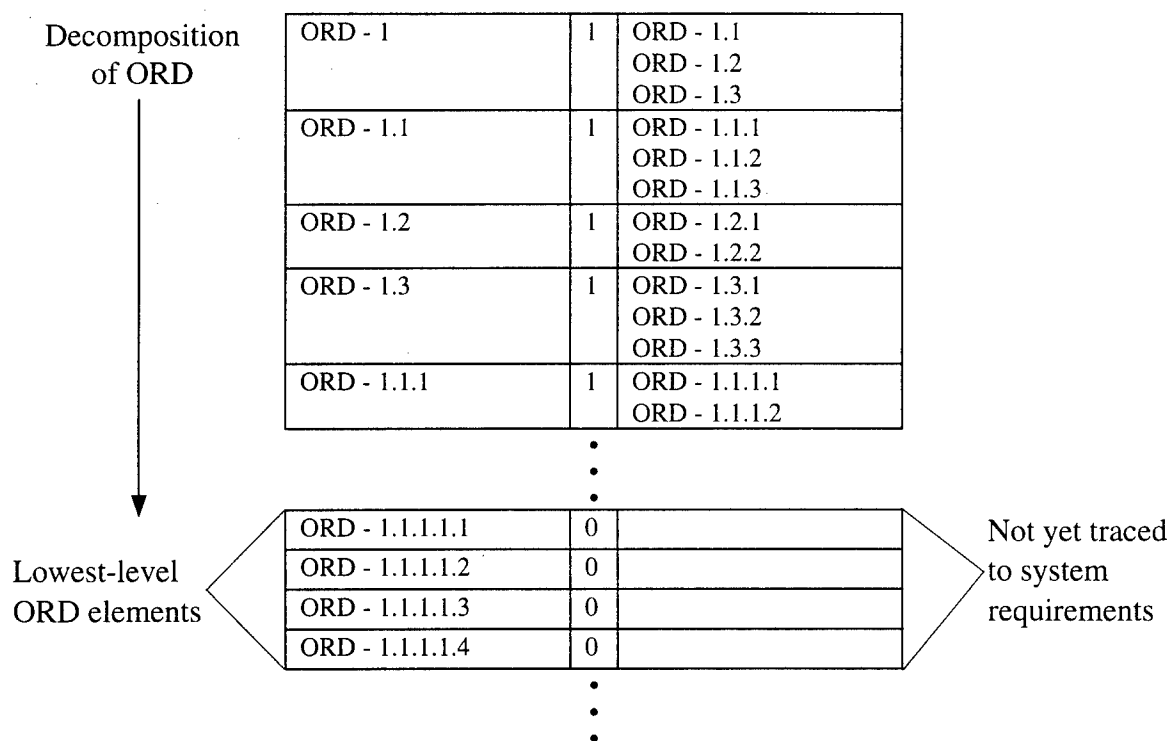


Figure 19. Example Section of Database after Initialization

Data entry is further explained by continuing from the initialization example shown in Figure 19. The lowest-level ORD elements have been duplicated in Figure 20, where they are traced to system requirements. Also, Figure 20 shows the decomposition of the system requirements traced for the ORD elements. Finally, Figure 20 shows the

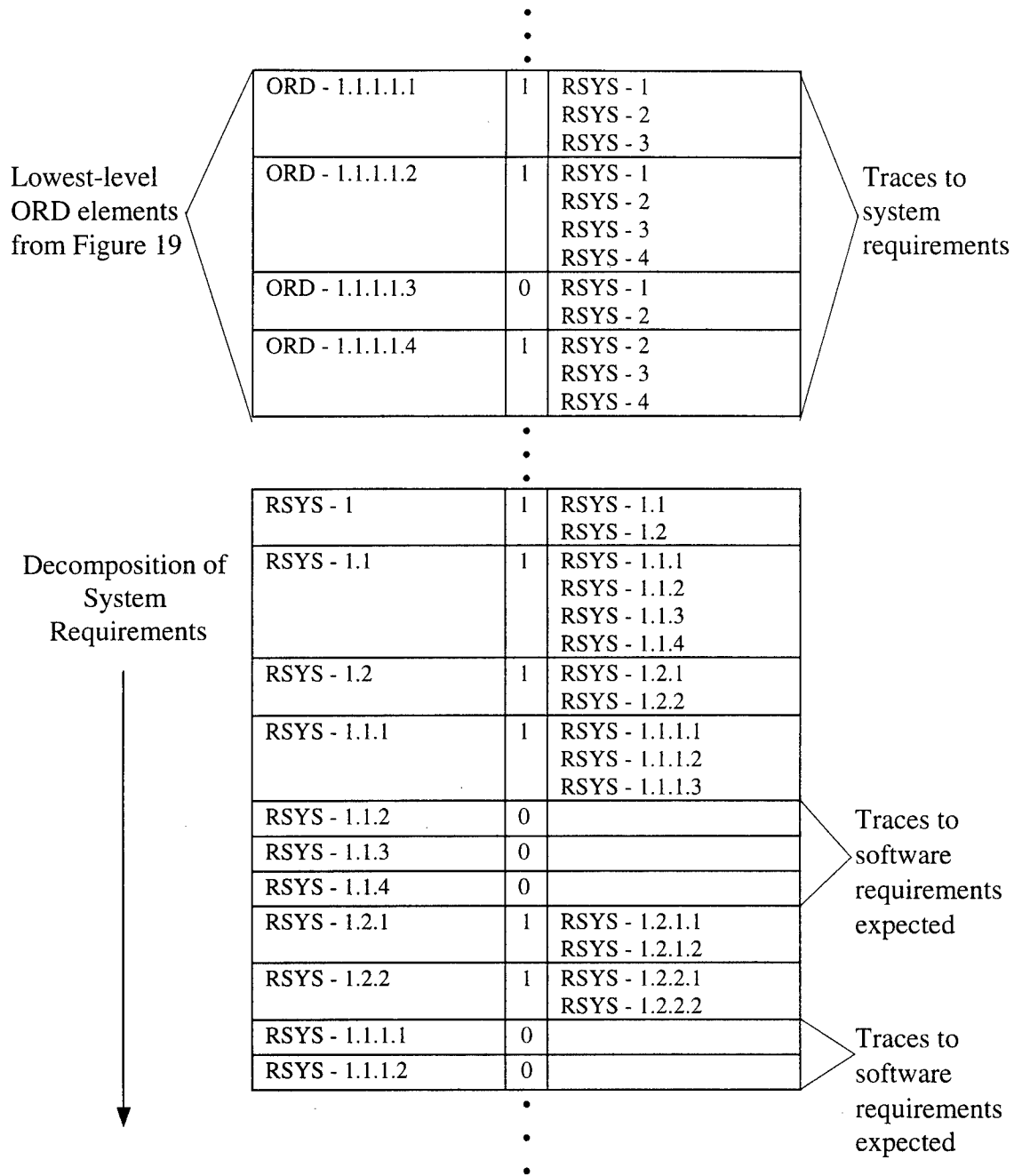


Figure 20. Data Entry Example

lowest-level system requirements expecting trace information (software requirements) to be entered in column (3). The example database section in Figure 20 also contains several database entries that have previously been described only in text.

Figure 20 shows two examples of incomplete traceability. ORD capability ORD - 1.1.1.1.3 is traced to system requirements RSYS - 1 and RSYS - 2, yet the complete trace flag in column (2) of the database entry is still set to 0.

During development, some traces may be more easily determined than others. The developer can leave the complete trace flag set to 0 until he or she has determined the elements in column (3) represent a complete trace from the element in column (1). The example in Figure 20 should be interpreted as follows: the developer knows that ORD - 1.1.1.1.3 traces to RSYS - 1 and RSYS - 2, but realizes there are more system requirements needed to address this particular ORD capability. Therefore, the developer enters the information of which he or she is certain, but leaves the complete trace indicator at 0. Then, the trace can be revisited and completed at a future date, at which time the complete trace indicator will be set to 1. The other type of incomplete traceability is displayed in Figure 20 by the following system requirements: RSYS - 1.1.2 through RSYS - 1.1.4, RSYS - 1.1.1.1, and RSYS - 1.1.1.2. These system requirements have not been addressed yet, and need software requirements entered in column (3) to establish traces for these entries.

The database continues to expand in this manner. The software requirements are entered in column (3) of the system requirements to software requirements traces. Then

the software requirements are decomposed as describe previously. The software requirements are then traced to HLD elements, which are decomposed, then traced to LLD elements, etc., until all the decomposition and trace data is entered in the database. This cycle of decomposition and tracing occurs over the course of system development.

4.8.4 Error Checking. A minimum amount of error checking is required in a database of this size, complexity, and importance. Error checking can either be accomplished upon data entry, as a separate utility that is executed on the database, or some combination of the two. Only the most basic, necessary types of error checking are described in the paragraphs that follow.

Both decomposition and trace entries in the database must be checked for accuracy. Of course, all input should be checked for valid element character codes, complete flag indicators (0 or 1), and element identification formats. In addition, decomposition must be checked for accuracy. For example, an error should result if the database reflects the decomposition of RO - 5.3.1 into RO - 5, RO - 6, and RO - 7, because a lower-level element *cannot* be decomposed into upper-level element. Also, an error should result if the elements are decomposed using valid character codes *outside* the artifact of the element being decomposed. For example, it is incorrect to decompose software requirements (RO) into code elements (CSUs).

Traces must be checked for accuracy also. Only legitimate traces are permitted; in this implementation, there are no traces from the HLD to the ORD. Therefore, an attempt to enter such a trace in the database should result in an error. In addition, traces must be checked to ensure they are traced at the correct level, i.e., from the lowest-level of

decomposition to the highest-level of the *element traced to*. For example, it is incorrect to attempt to trace a lowest-level requirement such as RO - 2.3.1.4 to a HLD element such as DH - 2.4.1.3.

Once decomposition and trace data is entered in the database, information can be retrieved immediately. As development progresses and more data is added to the database, more information about the system under development becomes available. The retrieval of information from the database is discussed in the next section.

4.8.5 Information Retrieval. Since the focus of this research is the definition and evaluation of software effectiveness, the discussion of information that is retrieved from the database will only concern information that can assist in this effort. The database sections from Figure 18 are displayed again in Figure 21, with emphasis on the sections that will provide the trace information used in the evaluation of software effectiveness. These emphasized database sections in Figure 21 involve five of the six *software* artifacts discussed Section 4.2.5. The sixth software artifact, unit tests, is not used in the evaluation of software effectiveness.

The four database sections emphasized in Figure 21 are the core of the software effectiveness evaluation:

- 1) Traces from Software Requirements to HLD
- 2) Traces from HLD to LLD
- 3) Traces from LLD to Code
- 4) Traces from Software Requirements to Validation Tests

1)	Decomposition of ORD
	Traces from ORD to System Requirements
2)	Decomposition of System Requirements
	Traces from System Requirements to Software Requirements
3)	Decomposition of Software Requirements
	Traces from Software Requirements to Validation Tests
	Decomposition of Validation Tests
	Traces from Software Requirements to HLD
4)	Decomposition of HLD
	Traces from HLD to LLD
5)	Decomposition of LLD
	Traces from LLD to Code
6)	Decomposition of Code
	Traces from Code to Unit Tests
	Decomposition of Unit Tests

- ☐ Data used in software effectiveness evaluation
☒ Data NOT used in software effectiveness evaluation

Figure 21. Database Information Used in Software Effectiveness Evaluation

Each type of trace listed above is described in Section 4.6, *Trace Descriptions*, and will be referred to individually as *a set* of traces and collectively as *the sets* of traces. These sets of traces, stored as software development information, will provide all the information necessary to determine the software's *effectiveness*, as defined in Chapter 3. Namely, software effectiveness is defined as the degree to which the software requirements are satisfactorily met (Section 3.10), and requirements satisfaction is implied by complete requirements traceability throughout the software development process (Section 3.11.3).

Each of the four sets of traces above provides a *degree of traceability*, stated as a percentage, between two software artifacts of the development process. For each set of traces, the degree of traceability is the number of traces that *actually* exist divided by the number of traces that *should* exist. Refer to Figure 22, which contains a sample portion of the database; this figure will be used to explain the calculation of the degree of traceability.

Put simply, the traces are identified by the character codes in columns (1) and (3) of each database entry, and the number of complete traces is counted by adding up the complete trace flags that are set to 1 in column (2) of each identified trace. The number of complete traces is divided by the number of traces of that type; this yields the degree of traceability between the two software artifacts participating in the trace. For the small section of the software requirements to HLD traces in Figure 22, there are 13 traces, 10 of which are complete, yielding 10/13 or 77% as the degree of traceability. This calculation is repeated for *all* the requirements to HLD traces, yielding the degree of traceability between the software requirements and the HLD.

Each set of traces yields a degree of traceability. This degree of traceability pertains to the “completeness” of the connection between the two software artifacts. The connections between the artifacts are documented by each set of traces. By following the example of the traceability calculation on the sample database in Figure 22, traceability calculations are made for each set of traces. These traceability calculations yield four degrees of traceability, which correspond to the four sets of traces, and quantify the

10 of 13 traces
are complete

Degree
of Traceability
= $10/13 = 77\%$

RO - 1.1.1	1	DH - 1 DH - 2 DH - 3
RO - 1.1.2	1	DH - 1 DH - 2 DH - 3 DH - 4
RO - 1.1.3	0	
RO - 1.1.4	1	DH - 2 DH - 3 DH - 4
RO - 1.2.1	1	DH - 4 DH - 5 DH - 6
RO - 1.2.2	1	DH - 3 DH - 4 DH - 5 DH - 6
RO - 1.2.3	0	DH - 5 DH - 6
RO - 1.2.4	1	DH - 2 DH - 3 DH - 4
RO - 1.2.5	1	DH - 3 DH - 4 DH - 5 DH - 6
RO - 1.2.6	1	DH - 5 DH - 6
RO - 1.2.7	1	DH - 5 DH - 6 DH - 7
RO - 1.3.1	1	DH - 6 DH - 7 DH - 8
RO - 1.3.2	0	DH - 7

Figure 22. Example Database for Degree of Traceability Calculation

following aspects about the software being developed:

- 1) How well the software requirements were incorporated into the HLD.
- 2) How much of the HLD was retained during the detailing to the LLD.
- 3) How much of the LLD was implemented in the code.
- 4) How many software requirements were traced to validation tests.

These four pieces of information, quantified by the four degrees of traceability, will be referred to as the *components* of software effectiveness, since they are used in the software effectiveness calculation. The software effectiveness calculation is explained in Section 4.9.

Although these components of software effectiveness are the most important information retrieved from the database, other information indirectly related to software effectiveness can be obtained from the database. By searching on the complete trace flags, elements that are *not* traced can be located. Elements that are not traced show incomplete traceability, which reveals where the software is *ineffective*, according to the definition of software effectiveness. These incomplete traces expose requirements that have not been incorporated in the HLD, requirements that have no validation test, and LLD elements that have not yet been coded. Incomplete traces show the remaining work in the software development process and also prevent requirements, design elements, and tests from being “lost” during development.

There are two types of incomplete traces and they can be distinguished by the elements (or lack of elements) in column (3). One type of incomplete trace is a software element that has not yet been considered by the developers at all and, consequently, has

no software elements in column (3) of the database entry. From Figure 22, there is one requirement that is not addressed at all: RO - 1.1.3.

The other type of incomplete trace is a “partially” complete trace. A partially complete trace contains software elements in column (3) of the database entry, but these elements do not completely address the element being traced from column (1). From Figure 22, the requirements that have partially complete traces are RO - 1.2.3 and RO - 1.3.2.

4.8.6 Creation of Traceability Matrices. The traceability matrix is a graphical presentation of the data contained in the database. The data From Figure 22, showing a portion of the requirements to HLD database, is presented in matrix form in Figure 23. Traces appear as shaded squares on a horizontal line connecting the requirement with HLD element(s). Complete traceability and partial traceability between elements are shown in two different shades at the intersection of the requirement and the HLD element. A lack of traceability from a software requirement to an HLD element is indicated by a horizontal blank line in the matrix.

By inspection of the matrix in Figure 23, it is simple to see that RO - 1.1.3 has not been addressed in the HLD since it has no trace to any HLD elements and RO - 1.2.3 and RO - 1.3.2 are partially traced to the HLD. At a glance, the traceability matrix shows a “picture” of the software’s effectiveness; as more darkly shaded squares are added to the matrix, the software is “seen” as more effective.

4.8.7 Summary. Section 4.8 describes the operations that can be performed on the database, as well as the various types of information that can be retrieved from the

Traceability of Requirements to High-Level Design	DH - 1	DH - 2	DH - 3	DH - 4	DH - 5	DH - 6	DH - 7	DH - 8
RO - 1.1.1								
RO - 1.1.2								
RO - 1.1.3								
RO - 1.1.4								
RO - 1.2.1								
RO - 1.2.2								
RO - 1.2.3								
RO - 1.2.4								
RO - 1.2.5								
RO - 1.2.6								
RO - 1.2.7								
RO - 1.3.1								
RO - 1.3.2								




 = Complete Traceability
 = Partial Traceability
 = No Traceability

Figure 23. Example Portion of the Requirements to HLD Traceability Matrix

database. Although many types of information can be obtained from the database, the focus in this section was the information that can aid in the evaluation of software effectiveness. This information includes:

- 1) Calculations for the degrees of traceability, which are components of the overall software effectiveness calculation.
- 2) Identification of incomplete traceability, which reveals where the software is, by definition, *ineffective*.
- 3) A graphical presentation of software effectiveness in a traceability matrix.

It is important to note that partial traceability only indicates a lack of complete traceability. Partial traceability is not predictive, i.e., there is no way to determine how “close” the trace is to complete traceability. Therefore, partial traceability offers little more information than no traceability at all. However, partial traceability shows *some* progress towards complete traceability and is easily distinguished in the database, so partial traceability is identified and documented in this approach.

4.9 Calculation of Software Effectiveness

4.9.1 Five Components of Software Effectiveness Evaluation. Four components of software effectiveness are described in Section 4.8.5 above. These components quantify the degree of traceability between the artifacts of the software development process. The fifth component of the software effectiveness evaluation is the percentage of validation tests passed satisfactorily. Although it has nothing to do with traceability, this fifth component is necessary in the calculation of software effectiveness. If the software requirements are completely traced through the design to the code, yet the software fails all of its validation tests, the software cannot be considered effective. In summary, the five components of software effectiveness consist of five percentages: four describe the degree of tracability between the software artifacts, and the fifth indicates the percentage of validation tests passed.

4.9.2 Combining Effectiveness Components for Overall Evaluation. A composite value for software effectiveness can be obtained by using the five effectiveness components. To simply average the separate components would be misleading, because an average does not take into consideration the impact one effectiveness component has

on another. For example, consider the software system in Figure 24. The five components of software effectiveness are:

- 1) 50% traceability from software requirements to HLD
- 2) 100% traceability from HLD to LLD
- 3) 100% traceability from LLD to Code
- 4) 100% traceability from software requirements to validation tests
- 5) 50% of the validation tests passed

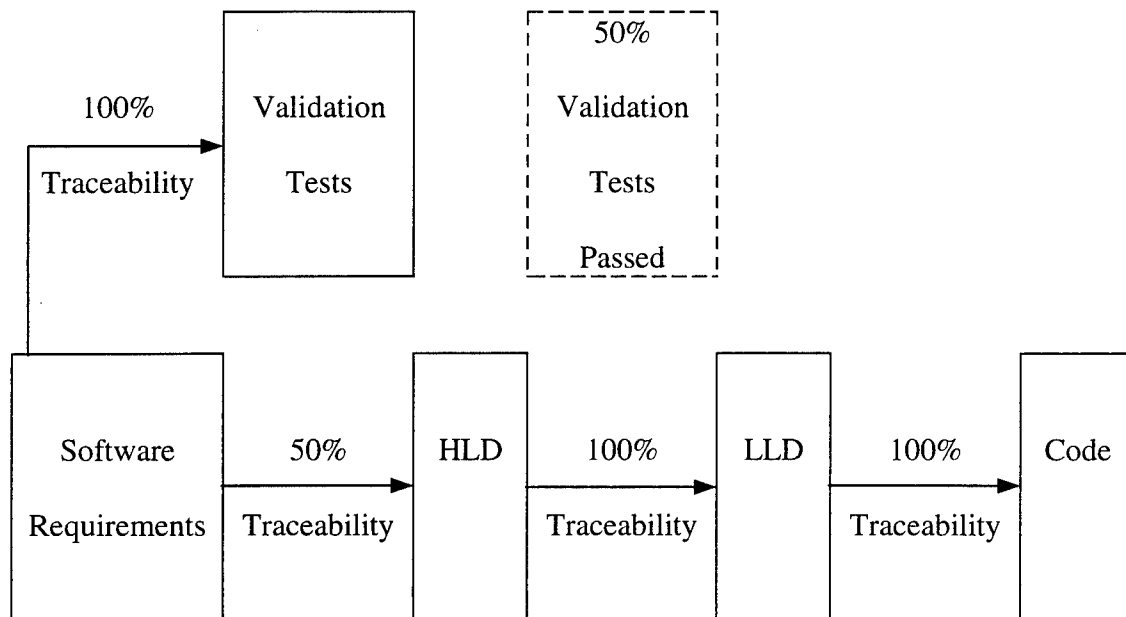


Figure 24. Effectiveness Components for an Example Software System

A simple average of all five effectiveness components yields:

$$\frac{50\% + 100\% + 100\% + 100\% + 50\%}{5} = 80\%$$

This overall effectiveness rating is misleading, since 80% effective implies the software is 20% from being completely (100%) effective, but does not describe the fact that *half of the software requirements did not make it into the design*.

To obtain a more representative overall assessment of the software's effectiveness, multiply the effectiveness components together. Multiplying the components for an overall assessment is similar to calculating the reliability of a system by multiplying the reliabilities of each of the system's individual components. The multiplication of the individual components to achieve an overall quantification is also similar to Seiler's calculation of *system* effectiveness, as described in Section 3.8.1. In the example from Figure 24, the overall assessment of software effectiveness yields:

$$(50\%) (100\%) (100\%) (100\%) (50\%) = 25\%$$

The calculation above, by multiplying, conserves the impact of inadequate traceability between artifacts. The calculation does not "spread out" the lack of traceability between components over all the other components, as it does with a simple average. By multiplying, this overall effectiveness rating provides better insight into a system that has failed to account for half of its requirements.

4.10 Summary of Software Effectiveness Approach

This chapter outlined an approach to evaluate software effectiveness through requirements traceability, which will subsequently be referred to as the Software Effectiveness Traceability Approach (SETA). Software effectiveness is defined as the

satisfaction of software requirements which is measured by the traceability from the software requirements throughout the various software development artifacts. The degree of traceability is measured in the development process by the number of elements that are traced from one artifact to another, such as the software requirements to the HLD. There is a degree of traceability, expressed as a percentage, for each of the following sets of traces in this approach:

- 1) Software Requirements to HLD
- 2) HLD to LLD
- 3) LLD to Code
- 4) Software Requirements to Validation Tests

Each degree of traceability represents a component of the software effectiveness evaluation. The fifth component of software effectiveness is the percentage of validation tests passed satisfactorily. The components of software effectiveness can be viewed separately to consider the progress made in various stages of the software development process, or they can be multiplied together for a composite software effectiveness value.

An example database implementation of SETA was provided. The information concerning the software artifacts and the traces between them are stored in the database, and the traceability information can be entered, edited, and retrieved to yield the software effectiveness components described above.

5. Demonstration of the Software Effectiveness Traceability Approach

5.1 Introduction

The Software Effectiveness Traceability Approach (SETA), outlined in Chapter 4, is demonstrated below. The demonstration utilizes actual data from a real-world software development program in the database form previously described. The traceability data used in the demonstration is from a small portion of the avionics system software aboard the C-17. The C-17 is the latest aircraft in the Air Force inventory that provides airlift of outsize cargo to austere airfields, as well as airdrop and full aeromedical evacuation capabilities [Des95].

This chapter begins with a discussion of the process which led to the decision to demonstrate the traceability portion of SETA. Section 5.3 documents the search for traceability data. In the next section, 5.4, some background information is provided on some of the components of the C-17 avionics system. Section 5.5 gives more detailed information on MIL-STD-1553B, the standard for one particular avionics component on the C-17 known as the 1553 Data Bus. The demonstration traces the 1553 Data Bus software from the ORD to the code and from the software requirements to their validation tests. However, before the demonstration, the software development artifacts from the C-17 SPO are described in Section 5.6. The next section, 5.7, describes the correlation between the artifacts from the C-17 SPO and the artifacts from SETA. Section 5.8 contains the demonstration of SETA, using the data from the C-17 SPO and the terminology from SETA. The effectiveness calculation for the 1553 Data Bus software is given in Section 5.9. The final section offers a summary and some concluding remarks.

5.2 *The Decision Process to Demonstrate SETA*

The decision to demonstrate the traceability portion of SETA was made after considering other alternatives. To lend credibility to SETA, it was necessary to go beyond an explanation and offer some sort of indication that SETA worked as envisioned.

The best validation method would be to apply SETA to a software development project from its inception, and evaluate its utility after the software product is delivered. This validation method is impractical, since the development cycle of the typical software development project evaluated by AFOTEC may take several *years* to deliver a final product; more practical alternatives that were considered are discussed below.

5.2.1 Validation of SETA with Actual Data. Validating SETA with actual traceability data would have been the next best way to legitimize the approach, but this presented a few insurmountable problems. Validating SETA using traceability data involves three basic steps:

- 1) Locate historical data on a software development project, either stand-alone or as part of a weapon system, that has been evaluated by AFOTEC/SAS.
- 2) Take the traceability data from the software development project located in step one, and apply it to SETA, i.e., enter the data in the database and calculate the software's effectiveness.
- 3) Compare the effectiveness rating with the results of AFOTEC's evaluation.

There are a few problems with this process, which make this type of validation impossible. First of all, the sheer size of some of the software projects evaluated by AFOTEC make it difficult to use them in SETA, since the data entry alone would take

time and effort beyond the focus of this research. However, this difficulty could have possibly been overcome by using data from a small portion of a software development project, then generalizing the results to the entire project. Secondly, locating historical data on a software development project that has been evaluated by AFOTEC was not difficult, but locating the traceability data for a given project was extremely difficult. Complete traceability data that would be applicable to SETA simply *does not exist*. Granted, there is some traceability data for each software development project, but none of the data was complete to the point where could be directly used in SETA. Finally, since AFOTEC has no effectiveness evaluation methodology currently in use, there are no other effectiveness evaluation results with which to compare the results of SETA. Considering these problems, validation with actual data was eliminated as a way to legitimize SETA.

5.2.2 Validation of SETA with Test Data. Another alternative to lend credibility to the software effectiveness evaluation approach would be to develop traceability data for a simulated software development project, apply SETA to the traceability data of this test-case project, and analyze the results.

This alternative had many difficulties as well. Several examples of simulated traceability data were provided in the explanation of SETA in Chapter 4. An additional example would not prove beneficial, since the traceability data could be manipulated to display any effectiveness result that would show SETA in the most favorable light. Test cases or examples aid in the explanation of SETA, but are too subjective for validation if

they are introduced by the developer of the approach. Therefore, this variation on the validation of SETA was rejected as well.

5.2.3 Demonstration of SETA with Actual Data. The final alternative was to demonstrate SETA with actual data from an existing software development project. In this way, SETA gains more credibility than from the examples and explanations provided during the development of the approach. In addition, limitations to SETA may be discovered that would lead to changes before it was actually implemented into a full methodology. The demonstration of SETA to evaluate software effectiveness begins with a search for traceability data from an actual software development project.

5.3 The Search for Traceability Data

Finding traceability data on software for large weapon systems, particularly aircraft, proved extremely difficult. None of the seven SPOs contacted at Wright-Patterson AFB had the complete traceability necessary to apply to SETA. This was surprising, since requirements traceability is required in software development projects, according to the Data Item Descriptions (DIDs) referred to in MIL-STD-498 [MIL94]. Until recently, MIL-STD-498 was the governing document for software development in the military; it defined "a set of activities and documentation suitable for the development of both weapon systems and Automated Information Systems" [MIL94].

Of all the SPOs contacted, the C-17 SPO was the most promising prospect, since they were in the process of documenting complete software traceability. Their traceability included: system-level requirements to software requirements, software

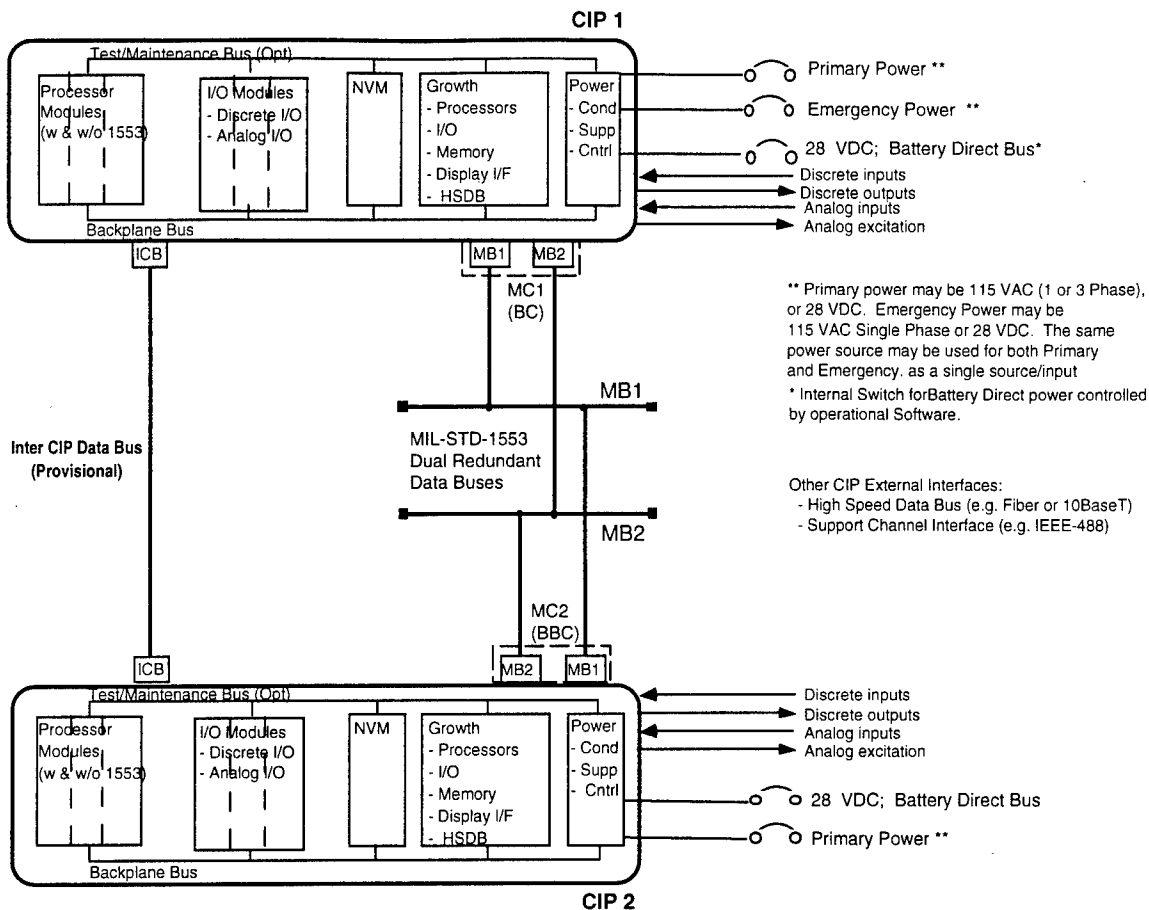
requirements to software design, software design to code, and software requirements to tests. This was *precisely* the traceability data needed to demonstrate the software effectiveness approach.

Unfortunately, the C-17 traceability data was not complete; therefore, hard copies of the software development documentation were provided to manually collect the traceability data. Before the demonstration of SETA, it is necessary to explain the terminology involved with the particular section of software that will be traced, as well as the software development documentation at the C-17 SPO.

5.4 Background Information on C-17 Avionics

5.4.1 Core Integrated Processor. The Core Integrated Processor (CIP) is a general purpose computer specifically designed to meet the real-time requirements of the C-17 aircraft [McD96]. There are two CIPs aboard each aircraft, with one operating as a backup. Figure 25 shows the architecture of the CIPs, connected by the MIL-STD-1553B Dual Redundant Data Busses.

5.4.2 Operating System Utilities. The OSU serve as the primary software interface between the CIP hardware and the C-17 Application Software (AS) of the various avionics subsystems. The OSU design is decomposed into 13 CSCs, one of which is the driver for the 1553 Data Bus. The 1553 Driver provides the control software for the message traffic on the data bus. Other CSCs in the OSU include the Built-In Tests (BITs) executed during power-up and maintenance, which perform diagnostic checks on



K. Rhine
12/3/94

Figure 25. System Diagram for C-17 Core Integrated Processor Architecture [McD96]

the 1553 Data Bus. The driver software, as well as the BIT software, for the 1553 Data Bus are the focus of the traceability demonstration.

5.4.3 1553 Data Bus. A major component of the CIP architecture is the 1553 Data Bus that is used to communicate between the avionics subsystems aboard the aircraft. The dual-redundant 1553 Data Busses are located in the center of Figure 25, connecting the two CIPs. Since the 1553 Data Bus software is the focus of the traceability demonstration, some background information of this particular component of the C-17 avionics system is provided in the next section.

5.5 Overview of MIL-STD-1553

5.5.1 Background. MIL-STD-1553 standardized the communication connections between avionics subsystems on aircraft. Up until the late 1960s, subsystem communication was handled by an increasing number of discrete physical connections between avionics units, resulting in increased weight and complexity [Tyl91]. To address this problem, MIL-STD-1553 was established in 1968; this standard was updated in 1978 and released as MIL-STD-1553B, but is still generally referred to as MIL-STD-1553. Although the terms MIL-STD-1553 and 1553 Data Bus are often used interchangeably, in this chapter MIL-STD-1553 refers to the document that describes the capabilities and characteristics of the actual component, which is referred to as the 1553 Data Bus.

The 1553 Data Bus saves space and weight by replacing the interconnecting wires between avionics units with a single twisted pair. All the communication between avionics subsystems is multiplexed over this wire. The 1553 Data Bus also offers cost savings from the decreased space and weight aboard the aircraft. Additional cost savings are realized through standardization; the 1553 Data Bus can be used on many types of aircraft.

5.5.2 1553 Data Bus Modes. Up to 32 units or terminals can be connected to the data bus. One terminal, known as the bus controller, acts as a “traffic cop” as it coordinates the flow of information along the data bus [Tyl91]. Another type of terminal is the monitor, a passive device, that records data for later analysis and can act as a backup to the bus controller [Tyl91]. The third type of terminal is the remote terminal and is defined as “all terminals not operating as the bus controller or bus monitor” [Tyl91]. Remote terminals gather data from aircraft sensors and convert the data to a

format that can be placed on the 1553 Data Bus. The remote terminals also receive information from the data bus, and convert it to a format that can be used by the aircraft.

5.5.3 Software for the 1553 Data Bus. The 1553 Data Bus requires software to function properly. The 1553 Driver is responsible for interfacing between the AS from the various avionics subsystems and the 1553 Data Bus. The 1553 Driver allows the AS to specify the operating modes of the interface. Corresponding to the modes described in Section 5.5.2 above, the interface can function as Bus Controller (BC), Remote Terminal (RT), Bus Monitor (MON) or a combination of Remote Terminal/Bus Monitor (RT/MON) [Tyl91]. The 1553 Data Bus provides communication between avionics components by using messages. A message entry is an individual bus transaction, which contains various pieces of data, such as commands, data to transmit (or data to receive), and message status information [Tyl91].

In addition to the driver, the 1553 Data Bus requires diagnostic programs that are used to check the functions of the data bus during power-up or maintenance. These diagnostic programs are tests that are built in to the 1553 software architecture, and are referred to as BITs (for Built-In Tests). The BITs for the 1553 Data Bus are portions of the power-on BITs (POBITs) and the maintenance initiated BITs (MIBITs), which test all the components of the CIP. For the demonstration, only the portions of the POBITs and MIBITs that contain the tests for the 1553 Data Bus are considered. However, before the demonstration, an explanation of the software development terminology used at the C-17 SPO is provided in the next section.

5.6 *Software Development Terminology Used at the C-17 SPO*

5.6.1 Prime Item Development Specification. The Prime Item Development Specification (PIDS) contains the system requirements. Specifically, the PIDS “establishes the performance, design, development and verification requirements” for a particular system [McD96]. In this case, the *system* is actually the CIP. The PIDS for the CIP corresponds to the system requirements artifact in SETA.

5.6.2 Computer Program Development Specification. The Computer Program Development Specification (CPDS) “establishes the performance, design, test and qualification of a computer program identified as Operating System Utilities” [Loc96a]. The software specified in this document is a collection of drivers that provide interfaces to the CIP. The CPDS for the Operating System Utilities (OSU) corresponds to the software requirements in SETA.

5.6.3 Computer Program Product Specification. The Computer Program Product Specification (CPPS) “establishes the requirements for complete identification of the C17 Core Integrated Processor (CIP) Operating System Utilities (OSU)” [Loc96b]. The CPPS contains the preliminary design, the detailed design, and the code for the OSU. The preliminary design corresponds to the HLD in SETA. The detailed design corresponds to the LLD in SETA. Of course, the code in CPPS refers to the same artifact (the code) in SETA.

5.6.4 Computer Program Test Plan. The Computer Program Test Plan (CPTPI) documents the test procedures that test against the software requirements in the CPDS. These test procedures are not to be confused with the BITs, which are part of the 1553 software architecture. These test procedures validate the requirements of the 1553

software. Since the BITs are part of the 1553 software, there are actually test procedures in the CPTPI that test the BITs. Incidentally, the acronym “CPTPI” is used to distinguish the test plan from another software artifact, the Computer Program Test Procedures (CPTPr). The tests in the CPTPI correspond to the validation tests in SETA.

5.6.5 Unit Tests. The unit tests were not available for the traceability demonstration of SETA. The unit tests are created and maintained at the individual contractor that develops a particular piece of software [Est96]. Therefore, traceability from the CSUs to the unit tests is not included in the demonstration of SETA.

5.6.6 Summary. The artifacts for the software development process at the C-17 SPO correspond to the artifacts in SETA. The correspondence between these artifacts is summarized in Table 1.

Software Development Artifacts for the OSU of the CIP aboard the C-17	Corresponding Software Development Artifacts from SETA
ORD	ORD
PIDS	System Requirements
CPDS	Software Requirements
CPPS	HLD
CPPS	LLD
CPPS	Code
CPTPr	Validation Tests

Table 1. Correspondence Between Software Development Artifacts

From this point forward, for the purposes of the demonstration, the terminology from SETA will be used.

5.7 Translation of Element Identification Methods

5.7.1 Introduction. The element identification method in the artifacts from the C-17 SPO that contain the OSU software are very similar to the code and numbering scheme of SETA. Since the *artifacts* from SETA will be used in the demonstration, the *identification scheme* from SETA will also be used in the demonstration, for consistency. The differences in the two identification methods, and the translation of the C-17 SPO identification method to the SETA identification method, are explained in the next two sections.

5.7.2 Element Identification Method for C-17 SPO Artifacts. Within the C-17 SPO artifacts (PIDS, CPDS, etc.), there are no “codes” as in SETA. A similar nested numbering scheme is used for decomposition, and it is understood from the context what elements are being discussed. In the case of the C-17 SPO, the context is the artifact being considered; if there is a reference to paragraph 3.1.2.2 in the CPDS, it is understood that the element being discussed is a software requirement. The numbering scheme is nested, as with SETA, except elements within a paragraph of a software artifact are offset by brackets. For example, within paragraph 3.1.2.2 of the CPDS, the software requirement in the paragraph may list five subrequirements. These five subrequirements are identified within the traceability sections of the artifact as 3.1.2.2 [1], 3.1.2.2 [2], 3.1.2.2 [3], etc. In further decomposition, lower-case letters are used to identify subelements. For example, 3.1.2.2 [1] is decomposed into 3.1.2.2 [1] a), 3.1.2.2 [1] b), etc. Although the lower-case letters are used in the text of the artifact, they are not used in the traceability section. The decomposition recorded in the traceability section stops at the level indicated by the brackets, such as 3.1.2.2 [1].

5.7.3 Translation of Identification Method to SETA. The translation of the identification method in the C-17 SPO artifacts to SETA is quite simple, and similar to the translation of the ORD element identification method discussed in Section 4.4.1. The translation of the identification method used in the C-17 SPO software artifacts is accomplished as follows: the brackets are removed and the familiar dotted notation is substituted; for the lower-case letters, the number indicating the letter's position in the alphabet is substituted for the letter. For example, from the CPDS, 3.1.2.2 [1] a) is translated into RO - 3.1.2.2.1.1 in SETA. The CPPS contains the preliminary design, detailed design, and code in separate sections; the elements in these sections are translated with the appropriate number and the codes corresponding to HLD, LLD and CSU respectively.

Now that the terminology has been defined and a translation description has been given, a demonstration of the traceability for the 1553 Data Bus software is provided in the next section.

5.8 Demonstration of Traceability of 1553 Data Bus Software

5.8.1 Introduction. Traceability of the 1553 Data Bus software is accomplished in the following sections, using the terminology from SETA. Each section describes where the 1553 Data Bus (or its associated software) appears in the software development artifact and its connection to the next artifact in the software development process. As stated previously, traceability to the units tests is not included since these tests are kept by the contractor and were not available for this research. Traceability is described in two ways: first, the traces are described in text; then, the traces are described graphically, by

providing examples of database sections containing the traces. Throughout the demonstration, only the elements that have a direct relationship to the 1553 Data Bus are considered; all other elements are ignored.

5.8.2 ORD to System Requirements. The 1553 Data Bus is referenced in the ORD in paragraph 5a(2)(c) [AMC93]. This is translated into the terminology from SETA and is subsequently referred to as ORD - 5.1.2.3. The 1553 Data Bus requirement from the ORD is stated as follows:

USAF designated standard avionics systems will be used where appropriate. The aircraft will have reliable and maintainable avionics equipment with adequate growth potential in computer memory and processing. An integrated avionics architecture is imperative. MIL-STD-1553B multiplex data busses will be used to integrate existing avionics systems and increase growth capability. Particular attention must be paid to partitioning the avionics components. The aircraft design must facilitate the cooling of the avionics equipment such that routine ground maintenance operations can be performed at internal compartment temperatures up to 90 degrees Fahrenheit [AMC93].

The first requirement in the paragraph, "MIL-STD-1553B multiplex data busses will be used . . .", will be designated ORD - 5.1.2.3.1. The rest of the subrequirements are ignored, since they have nothing to do with the 1553 Data Bus. ORD 5.1.2.3.1 is traced to RSYS - 3.7.1 and RSYS - 3.7.6.6, which are paragraphs in the system requirements document; decomposition and traceability from ORD - 5.1.2.3, as entered in the database, are shown in Figure 26.

ORD - 5.1.2.3	1	ORD - 5.1.2.3.1 • • •
ORD - 5.1.2.3.1	1	RSYS - 3.7.1 RSYS - 3.7.6.6

Figure 26. ORD to System Requirements Database Entries

5.8.3 *System Requirements to Software Requirements.* The text of the two system requirements paragraphs, RSYS - 3.7.1 and 3.7.6.6, which were traced from ORD - 5.1.2.3.1, are shown below. RSYS - 3.7.1 is stated in the systems requirements artifact as follows:

MIL-STD-1553B Bus Interface. Dual-redundant, multiplex data buses in accordance with MIL-STD-1553B notice 1 interface. The MIL-STD-1553 buses shall be able to be configurable as Bus Controller (BC) or Remote Terminal (RT) or Bus Monitor (BM) or both RT and BM simultaneously via software commands [McD96].

RSYS - 3.7.1 is decomposed into the following subsystem requirements:

RSYS - 3.7.1.1

A hardware requirement that specifies the use of a 1553 Bus interface.

RSYS - 3.7.1.2

The MIL-STD-1553 buses shall be able to be configurable as Bus Controller (BC) or Remote Terminal (RT) or Bus Monitor (BM) or both RT and BM simultaneously via software commands.

Traceability of RSYS - 3.7.1.1 ends here, since it is a hardware requirement, but traceability will continue for RSYS - 3.7.1.2.

RSYS - 3.7.6.6 is stated in the systems requirements artifact as follows:

MIL-STD-1553 multiplex bus data transfer characteristics. The CIP shall have I/O modules that shall be implemented in accordance with MIL-STD-1553B notice 1. The CIP design shall be interrupt driven so that the processor can meet the MIL-STD-1553 timing requirements. The MIL-STD-1553 multiplex bus shall have Bus Controller, Remote Terminal, Bus Monitor, and Remote Terminal/Bus Monitor capability that shall be selectable under program control. The MIL-STD-1553B Multiplexed Bus design shall provide an interrupt mechanism to the I/O modules. Self-test shall be available via software commands from the OFP [Operational Flight Profile]. Self-test shall be comprehensive enough to provide fault detection capability defined in Built-In Test (BIT) paragraph 3.5.1.1.1 *Organizational Level BIT Provisions* [McD96].

RSYS - 3.7.6.6 is decomposed into six subsystem requirements:

RSYS - 3.7.6.6.1

The CIP shall have I/O modules that shall be implemented in accordance with MIL-STD-1553B notice 1.

RSYS - 3.7.6.6.2

The CIP design shall be interrupt driven so that the processor can meet the MIL-STD-1553 timing requirements.

RSYS - 3.7.6.6.3

The MIL-STD-1553 multiplex bus shall have Bus Controller, Remote Terminal, Bus Monitor, and Remote Terminal/Bus Monitor capability that shall be selectable under program control.

RSYS - 3.7.6.6.4

The MIL-STD-1553B Multiplexed Bus design shall provide an interrupt mechanism to the I/O modules.

RSYS - 3.7.6.6.5

Self-test shall be available via software commands from the OFP [Operational Flight Profile].

RSYS - 3.7.6.6.6

Self-test shall be comprehensive enough to provide fault detection capability defined in Built-In Test (BIT) paragraph 3.5.1.1.1 *Organizational Level BIT Provisions*.

Decomposition of RSYS - 3.7.1 and RSYS - 3.7.6.6, along with the traceability of their lowest-level subrequirements is shown in Figure 27. Refer to Figure 27 as the decomposition and traceability of the system requirements to the software requirements are explained in further detail.

Traceability for the purposes of software effectiveness stops here for RSYS - 3.7.1.1, RSYS - 3.7.6.6.1 and RSYS - 3.7.6.6.2, since they are hardware requirements for the CIP. In a full-blown traceability approach, these system requirements would be traced to hardware requirements, then traced to hardware design, etc. Since they have nothing to do with the software, these system requirement to hardware requirement traces are ignored in the database and play no part in the software effectiveness evaluation. RSYS - 3.7.6.6.3 is a duplicate of RSYS - 3.7.1.2 above, so it can be traced to the same software requirement that governs software control of the 1553 Data Bus mode. RSYS - 3.7.6.6.4 and RSYS - 3.7.6.6.5 are traced to their respective software requirements. RSYS - 3.7.6.6.6 requires that self-tests must follow another system requirement, RSYS - 3.5.1.1.1. This situation is accounted for by “decomposing” subrequirement RSYS - 3.7.6.6.6 into RSYS - 3.5.1.1.1. Then, RSYS - 3.5.1.1.1 is appropriately traced to the software requirement concerning self-tests, which is shown in Figure 27. In summary, the subrequirements that will continue to be traced are RSYS - 3.5.1.1.1 and RSYS - 3.7.6.6.3 through RSYS - 3.7.6.6.5. These lowest-level system requirements are traced to software requirements in Figure 27; the traceability demonstration continues in the next section with these software requirements.

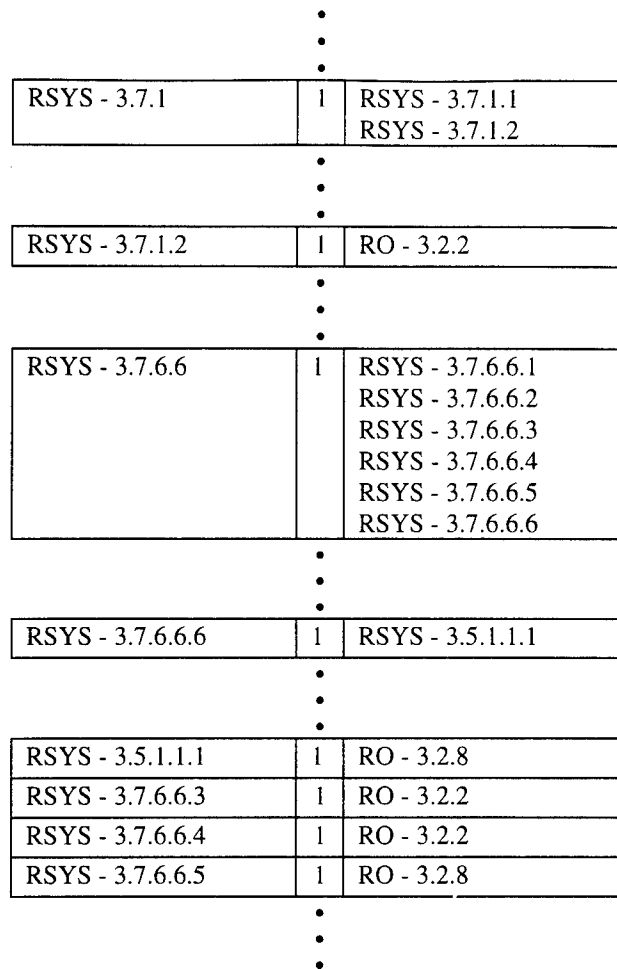


Figure 27. System Requirements to Software Requirements Database Entries

5.8.4 *Software Requirements to High-Level Design.* Software requirements RO - 3.2.2 and RO - 3.2.8, are partially decomposed and described in Figures 28 and 29, respectively. These software requirements RO - 3.2.2 and RO - 3.2.8, were traced from the system requirements in Figure 27.

RO - 3.2.2	: 1553 Driver
RO - 3.2.2.1	: 1553 Operational Mode Selection
RO - 3.2.2.1.1	
RO - 3.2.2.1.2	
RO - 3.2.2.2	: 1553 BC Operations
RO - 3.2.2.2.1	: 1553 BC Write
RO - 3.2.2.1.1	
•	
•	
•	
RO - 3.2.2.1.9	
RO - 3.2.2.2.2	: 1553 BC Start
RO - 3.2.2.2.1	
•	
•	
•	
RO - 3.2.2.2.9	
RO - 3.2.2.2.3	: 1553 BC Read
•	
•	
•	
RO - 3.2.2.3	: 1553 MON Operations
•	
•	
•	
RO - 3.2.2.4	: 1553 RT Operations
•	
•	
•	
RO - 3.2.2.5	: 1553 RT/MON Operations
•	
•	
•	

Figure 28. Partial Decomposition and Description of RO - 3.2.2

RO - 3.2.8	: Built-In Test (BIT)
RO - 3.2.8.1	: BIT Operational Modes
•	
•	
RO - 3.2.8.1.4	: Maintenance Initiated BIT (MIBIT)
•	
•	
RO - 3.2.8.1.4.2.17	: 1553 Receiver RAM Pattern Test (RRPT)
•	
•	
RO - 3.2.8.2	: BIT Test Organization and Execution
•	
•	
RO - 3.2.8.2.3	: Test Requirements
RO - 3.2.8.2.3.1	: Power-On BIT (POBIT)
RO - 3.2.8.2.3.1.1	: 1553 Receiver RAM Pattern Test (RRPT)
•	
•	
RO - 3.2.8.2.3.3	: 1553 Test Group
RO - 3.2.8.2.3.3.1	: 1553 Mode Code Processing Test (MCPT)
•	
•	
RO - 3.2.8.2.3.3.2	: 1553 Multiple Receive Test (MRXT)
•	
•	
RO - 3.2.8.2.3.3.3	: 1553 Off-Line Loopback Test (OLBT)
•	
•	
RO - 3.2.8.2.3.3.4	: 1553 Register Test (MRGT)
•	
•	
RO - 3.2.8.2.3.3.5	: 1553 RT Address Parity Test (RAPT)
•	
•	
RO - 3.2.8.2.3.3.6	: 1553 RT Address Test (RTAT)
•	
•	

Figure 29. Partial Decomposition and Description of RO - 3.2.8

The software requirement for the 1553 Data Bus driver, RO - 3.2.2, is decomposed into RO - 3.2.2.1 through RO - 3.2.2.5. Each of the five subrequirements is decomposed further in the C-17 documentation, eventually yielding 62 lowest-level software requirements.

For the purposes of this demonstration, only RO - 3.2.2.1 and RO - 3.2.2.2 will continue to be traced. As an example of the type of software requirement at the lowest level of decomposition, the text of RO - 3.2.2.1 and a portion of the text of RO - 3.2.2.2 are provided below, from the software requirements document [Loc96a]:

RO - 3.2.2.1 Operational Mode Selection

RO - 3.2.2.1.1

The 1553 driver shall accept a command from the AS to select the operating mode: either BC or RT, or MON or RT/MON.

RO - 3.2.2.1.2

The 1553 bus shall be reconfigured in less than 2 milliseconds when commanded to switch operating modes.

RO - 3.2.2.2 1553 BC Operations

RO - 3.2.2.2.1 1553 BC Write

RO - 3.2.2.2.1.1

The driver shall accept a list that specifies the number of message entries (maximum 512 entries) and an individual message entry for each message.

RO - 3.2.2.2.1.2

Bus transactions shall occur based on the order specified by the message entries. That is, the first entry is the first bus transaction to be commanded.

RO - 3.2.2.2.1.3

A message entry shall specify the bus (A or B) in which to attempt the message transaction.

Software requirement RO - 3.2.8, which encompasses the BITs, is also decomposed into several subrequirements that concern the 1553 Data Bus. These BITs include RO - 3.2.8.1.4 (MIBIT), RO - 3.2.8.2.3.1 (POBIT) and RO - 3.2.8.2.3.3 (1553 Test Group). The decomposition of RO - 3.2.8.1.4 and RO - 3.2.8.2.3.1 include the same test, RO - 3.2.8.3.1.1, the "1553 Receiver RAM Pattern Test Module (RRPT)" [Loc96a]. Put simply, both maintenance tests and power-on tests (MIBIT and POBIT) occasionally use the same test procedures during self-test.

RO - 3.2.8 is eventually decomposed into hundreds of lowest-level requirements. As before, for the purposes of this demonstration, most of the software requirements will be "trimmed" and traceability will continue with the following software requirements:

- RO - 3.2.8.1.4.2.17 (RRPT)
- RO - 3.2.8.2.3.1.1.1 (RRPT)
- RO - 3.2.8.2.3.3.1 (MCPT)
- RO - 3.2.8.2.3.3.3 (OLBT)
- RO - 3.2.8.2.3.3.5 (RAPT)

Figures 30 through 33 show the database entries for the selected software requirements that will continue to be traced for the demonstration. Figure 30 begins with the partial decomposition of the software requirement for the 1553 Data Bus Driver, RO - 3.2.2, into its lowest-level elements.

Next, Figures 31 and 32 show the partial decomposition of RO - 3.2.8 into its lowest-level subrequirements.

RO - 3.2.2	1	RO - 3.2.2.1
		RO - 3.2.2.2
		RO - 3.2.2.3
		•
		•
•		
RO - 3.2.2.1	1	RO - 3.2.2.1.1
		RO - 3.2.2.1.2
•		
RO - 3.2.2.2	1	RO - 3.2.2.2.1
		•
		•
•		
RO - 3.2.2.2.1	1	RO - 3.2.2.2.1.1
		RO - 3.2.2.2.1.2
		RO - 3.2.2.2.1.3
		RO - 3.2.2.2.1.4
		RO - 3.2.2.2.1.5
		RO - 3.2.2.2.1.6
		RO - 3.2.2.2.1.7
		RO - 3.2.2.2.1.8
		RO - 3.2.2.2.1.9
•		
•		

Figure 30. Database Entries for Partial Decomposition of RO - 3.2.2

Finally, Figure 33 shows the traces from the selected lowest-level subrequirements of RO - 3.2.2 and RO - 3.2.8 to their corresponding HLD elements. These traces were taken from the tracability matrices in the C-17 documentation.

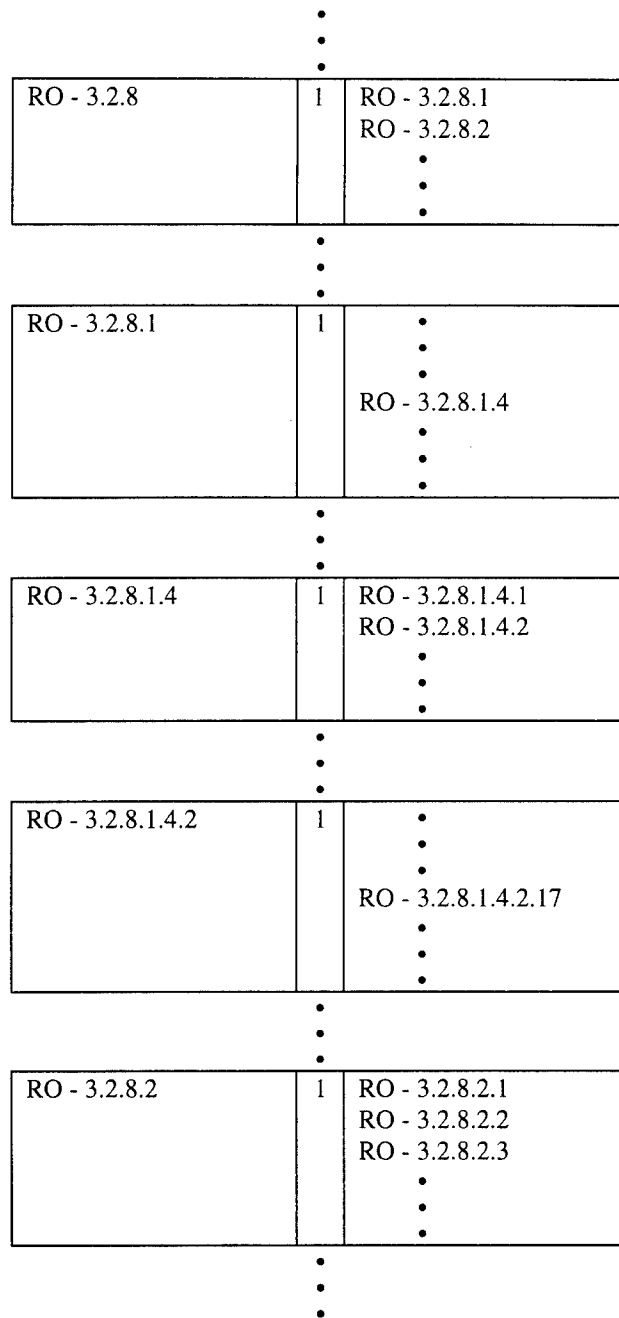


Figure 31. Database Entries of Partial Decomposition of RO - 3.2.8

•		
•		
•		
RO - 3.2.8.2.3	1	RO - 3.2.8.2.3.1 RO - 3.2.8.2.3.2 RO - 3.2.8.2.3.3
•		
•		
•		
RO - 3.2.8.2.3.1	1	RO - 3.2.8.2.3.1.1 • • •
•		
•		
•		
RO - 3.2.8.2.3.3	1	RO - 3.2.8.2.3.3.1 RO - 3.2.8.2.3.3.2 RO - 3.2.8.2.3.3.3 RO - 3.2.8.2.3.3.4 RO - 3.2.8.2.3.3.5 RO - 3.2.8.2.3.3.6
RO - 3.2.8.2.3.3.1	1	RO - 3.2.8.2.3.3.1.1 RO - 3.2.8.2.3.3.1.2 RO - 3.2.8.2.3.3.1.3 RO - 3.2.8.2.3.3.1.4 RO - 3.2.8.2.3.3.1.5 RO - 3.2.8.2.3.3.1.6 RO - 3.2.8.2.3.3.1.7
•		
•		
•		
RO - 3.2.8.2.3.3.3	1	RO - 3.2.8.2.3.3.3.1 RO - 3.2.8.2.3.3.3.2 RO - 3.2.8.2.3.3.3.3 RO - 3.2.8.2.3.3.3.4 RO - 3.2.8.2.3.3.3.5 RO - 3.2.8.2.3.3.3.6
•		
•		
•		
RO - 3.2.8.2.3.3.5		RO - 3.2.8.2.3.3.5.1 RO - 3.2.8.2.3.3.5.2 RO - 3.2.8.2.3.3.5.3 RO - 3.2.8.2.3.3.5.4 RO - 3.2.8.2.3.3.5.5
•		
•		
•		

Figure 32. (Continued)

•		
•		
•		
RO - 3.2.2.2.1.1	1	DH - 3.2.1
RO - 3.2.2.2.1.2	1	DH - 3.2.1
•	•	•
•	•	•
•	•	•
RO - 3.2.2.2.1.9	1	DH - 3.2.1
•		
•		
•		
RO - 3.2.8.1.4.2.17	1	DH - 3.2.9
•		
•		
•		
RO - 3.2.8.2.3.1.1	1	DH - 3.2.6
•		
•		
•		
RO - 3.2.8.2.3.3.1.1	1	DH - 3.2.10
RO - 3.2.8.2.3.3.1.2	1	DH - 3.2.10
RO - 3.2.8.2.3.3.1.3	1	DH - 3.2.10
RO - 3.2.8.2.3.3.1.4	1	DH - 3.2.10
RO - 3.2.8.2.3.3.1.5	1	DH - 3.2.10
RO - 3.2.8.2.3.3.1.6	1	DH - 3.2.10
RO - 3.2.8.2.3.3.1.7	1	DH - 3.2.10
•		
•		
•		
RO - 3.2.8.2.3.3.3.1	1	DH - 3.2.10
RO - 3.2.8.2.3.3.3.2	1	DH - 3.2.10
RO - 3.2.8.2.3.3.3.3	1	DH - 3.2.10
RO - 3.2.8.2.3.3.3.4	1	DH - 3.2.10
RO - 3.2.8.2.3.3.3.5	1	DH - 3.2.10
RO - 3.2.8.2.3.3.3.6	1	DH - 3.2.10
•		
•		
•		
RO - 3.2.8.2.3.3.5.1	1	DH - 3.2.10
RO - 3.2.8.2.3.3.5.2	1	DH - 3.2.10
RO - 3.2.8.2.3.3.5.3	1	DH - 3.2.10
RO - 3.2.8.2.3.3.5.4	1	DH - 3.2.10
RO - 3.2.8.2.3.3.5.5	1	DH - 3.2.10
•		
•		
•		

Figure 33. Software Requirements to HLD Database Entries

5.8.5 *High-Level Design to Low-Level Design.* All of the software requirements that will continue to be traced in this demonstration are traced to one of four HLD elements, which are listed and described below:

DH - 3.2.1 : 1553 Driver

DH - 3.2.6 : POBIT Utility

DH - 3.2.9 : MIBIT Utility

DH - 3.2.10 : Common_BIT Utility

Portions of the text for these four HLD elements, which were traced from the selected software requirements, are presented below. The text of the HLD elements is provided to show the depth and breadth of the design at this point in the software's development. It should be noted that any references to sections or appendices within the portions of text below are in regards to the HLD artifact from the C-17 SPO (in this case, the CPPS).

DH - 3.2.1 1553 Driver

The requirements allocated to this CSC from the CPDS are listed in Section 7. No derived design requirements have been imposed on this CSC at this time. The preliminary design of this CSC is an Ada package specification listed in Appendix A.

The 1553 driver is responsible for interfacing between the AS and one of the C-17 1553B buses. Each CIP is connected to both dual redundant 1553B buses. Separate interface hardware is provided to interface with each mission bus. Each 1553B bus will have a separate driver that is created to attach the driver to the appropriate bus.

The 1553B driver, allows the AS to specify the operations of the interface. The interface can function as a Bus Controller (BC), Remote Terminal (RT), Bus Monitor (MON) or combination of Remote Terminal/Bus Monitor (RT/MON).

The main interface between the AS and the driver is through the use of VxWorks READ, WRITE, and IOCTL operations. A READ operation returns data and status information. The data and status information returned is a function of the operational mode selected. A WRITE operation provides data and command information. The data and command structures are a function of the

operational mode. Since a common parameter passing structure is used in READ and WRITE operations, some information contained in the parameters is not used in some modes of operations, or may have a different meaning based on the operational mode.

The intent is to have one main parameter passed to the WRITE and returned from the READ. This parameter will consist of two parts: a header and a variable number of individual message entries. The header will contain general purpose data, such as the number of message entries, overall bus status information. A message entry is a definition of an individual bus transaction (i.e. 1553B message). The message entry will contain various pieces of data, such as commands, data to transmit (or data received), message status information, etc.

In addition to the basic commands, the 1553B driver also sets the VxWorks Error Number (errno) for specific types of errors. The 1553B Driver also allows for AS Hook routines via the VxWorks Message Queue structure [Loc96b].

DH - 3.2.6 POBIT Utility

The requirements allocated to this CSC from the CPDS are listed in Section 7. No derived design requirements have been imposed by Sanders on this CSC at this time. The preliminary design of this CSC consists of a C language specification that is listed in Appendix A.

After the application of power, each processor within the CIP automatically executes Power-On BIT (POBIT). The purpose of POBIT is to execute a set of tests required to assure minimum equipment performance within the allotted [sic] maximum execution time. As part of the Bootstrap function, POBIT is physically located within the Bootstrap PROM. Once POBIT execution has started, it will run to completion without interruption, i.e. it will run until all tests have been executed or the first failure has been detected.

POBIT is run separately in each processor. POBIT Tests associated with specific hardware functions are grouped together into POBIT Test Modules. As each Test is completed within a POBIT Test Module, a "Test Progress Indicator" will be incremented and written to a dedicated location within the Non-Volatile Memory, to the RS232 Serial Port A, and to a spare register that is available to other processors via the VME backplane. A failure of any POBIT test in either processor is considered a fatal CIP error and will be handled by the Bootstrap Utility [Loc96b].

DH - 3.2.9 MIBIT Utility

The requirements allocated to this CSC from the CPDS are listed in Section 7. No derived design requirements have been imposed on this CSC at this

time. The preliminary design of this CSC consists of an Ada package specification listed in Appendix A.

MIBIT is the only Initiated BIT utility. Its purpose is to re-execute and augment the tests supplied by the other modes of BIT in order to fully satisfy failure detection and isolation requirements. Normal MIBIT operation consists [sic] of an automatic sequencing of tests.

Upon request from the AS, MIBIT activates and replaces all other functions within the CIP. MIBIT is an on-ground only test mode and the AS must ensure that on-ground interlocks are satisfied prior to issuing the request. The CIP Fail Out discrete will be asserted for the duration of MIBIT. This serves to notify the other CIP that the CIP in MIBIT cannot support normal operation while MIBIT testing is ongoing.

BIT Test Modules that are shared by more than one BIT mode (EPOBIT, PBIT or MIBIT) are called from the Common_BIT CSC.

If a failure is detected, MIBIT creates a full fault record for storage in NVM. MIBIT will attempt to complete a full pass of all tests based on the continuation criteria for the failed test. Upon completion of all tests, if a failure was detected on either processor, it is considered to be, in effect, a fatal CIP error and the processor (and ultimately the CIP) will hang and appear dead to the outside world. If MIBIT completes successfully in one processor, that processor will wait for MIBIT to complete successfully in the other processor until time out. When both processors have successfully completed MIBIT, they will both initiate a soft reset in order to restart the CIP and bring it back to normal operation [Loc96b].

DH - 3.2.10 Common BIT Utility

The requirements allocated to this CSC from the CPDS are listed in Section 7. No derived design requirements have been imposed on this CSC at this time. The preliminary design of this CSC consists of an Ada package specification listed in Appendix A.

BIT Test Modules that are shared by more than one BIT mode (EPOBIT, PBIT or MIBIT) are called from the Common_BIT CSC. If a failure is detected, Common_BIT creates a full fault record for storage in NVM. The Common_BIT CSC has no direct interface to the AS [Loc96b].

The HLD elements are not decomposed any further and are traced to the LLD elements as shown in Figure 34.

5.8.6 Low-Level Design to Code. The LLD elements that were traced from the HLD elements in Figure 34 will continue to be traced in the demonstration, and are listed

and described below:

DL - 4.1 : 1553 Driver
DL - 4.6 : POBIT Utility
DL - 4.9 : MIBIT Utility
DL - 4.10 : Common_BIT Utility

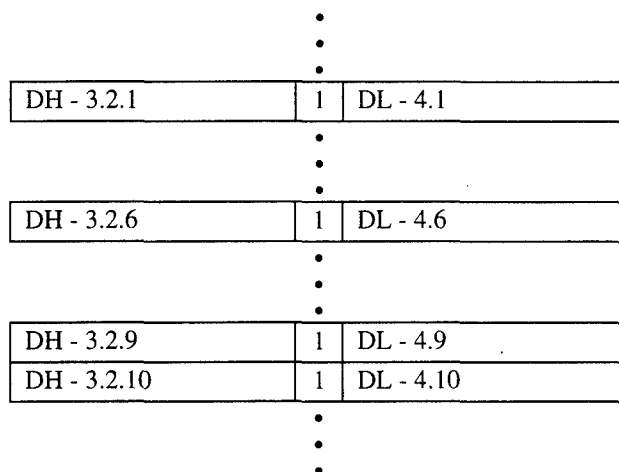


Figure 34. HLD to LLD Database Entries

Portions of the text for these four LLD elements are presented below [Loc96b]. The text of the LLD elements is provided to show the depth and breadth of the design at this point in the software's development.

DL - 4.1 1553 Driver

This CSC is the interface between the AS and the 1553B Devices on the [Input/Output Processor] IOP. This CSC is designed to be a VxWorks IO compatible driver that is accessed via the VxWorks IO System.

DL - 4.6 POBIT Utility

This CSC, when activated by the Bootstrap CSC, performs a Power-On Built-In-Test (POBIT) of the processor modules and the [Input/Output Module] IOM.

DL - 4.9 MIBIT Utility

This CSC, when activated by the Application Software (AS), performs a Maintenance Initiated Built-In-Test (MIBIT) of the processor modules and the [Input/Output Module] IOM.

DL - 4.10 Common_BIT Utility

This CSC encapsulates BIT Test Modules that are shared between the EPOBIT, PBIT and MIBIT CSCs.

In the C-17 software development artifacts, the code modules are not uniquely identified by a nested number, so they could not be translated into the identification method from SETA. However, the *names* of the code modules and their corresponding LLD elements were identical. For the purposes of this demonstration, the code modules are given the same numeric identifiers as their corresponding elements in the LLD from where they are traced.

Each element in the LLD is decomposed in a manner similar to the previously described decompositions. For example, the driver for the 1553 Data Bus, DL - 4.1, is decomposed into DL - 4.1.1 through DL - 4.1.13. Each subelement is further decomposed into two more subelements. The first subelement lists the specifications and constraints of that particular component of the driver; the second subelement contains the actual design. As an example, the text from the decomposition of DL - 4.1.1, the first subelement of the 1553 driver, is provided below.

DL - 4.1.1 CDI_Open_Wrapper

This CSU initializes the VxWorks IO system using the CDI_Open CSU. Its main purpose is to convert data formats for the VxWorks IO system.

DL - 4.1.1.1 CDI_Open_Wrapper Design Specification/Constraints

None.

DL - 4.1.1.2 CDI_Open_Wrapper Design

1. Input/output data elements - This CSU has the following I/O elements:
 - a. pDev - A pointer to a VxWorks I/O Device Header.
 - b. Name - Address of the name of the device.
 - c. Mode - VxWorks I/O Mode.
 - d. Integer returned by this function.
2. Local data elements - This CSU has the following local data elements:
 - a. Fd - A local VxWorks file descriptor (fd).
 - b. Result - A CDI_Status_Type which indicates the status of the CDI_Open.
3. Interrupts and signals - None
4. Algorithms - Calls the CDI_Open CSU and gets Result and Fd back.
5. Error handling - If an Ada exception occurs in this CSU, a failure status is returned.
6. Data conversion - Converts VxWorks I/O Open syntax to Ada-specific CDI_Open syntax.
7. Use of other elements - CDI_Open
8. Logic flow - Returns success status when CDI_Open returns success and returns failure if CDI_Open fails or an Ada exception occurs.
9. Data structures - None.
10. Local data files or database - None.
11. Limitations - None [Loc96b].

Decomposition and trace database entries are displayed in Figures 35 through 38, respectively, for the following LLD elements: 1553 Driver, POBIT Utility, MIBIT Utility, and Common_BIT Utility. In these figures, only the portions of software selected in Section 5.8.4, to be continued in the traceability demonstration, are shown. This software includes portions of the 1553 Driver and the BITs.

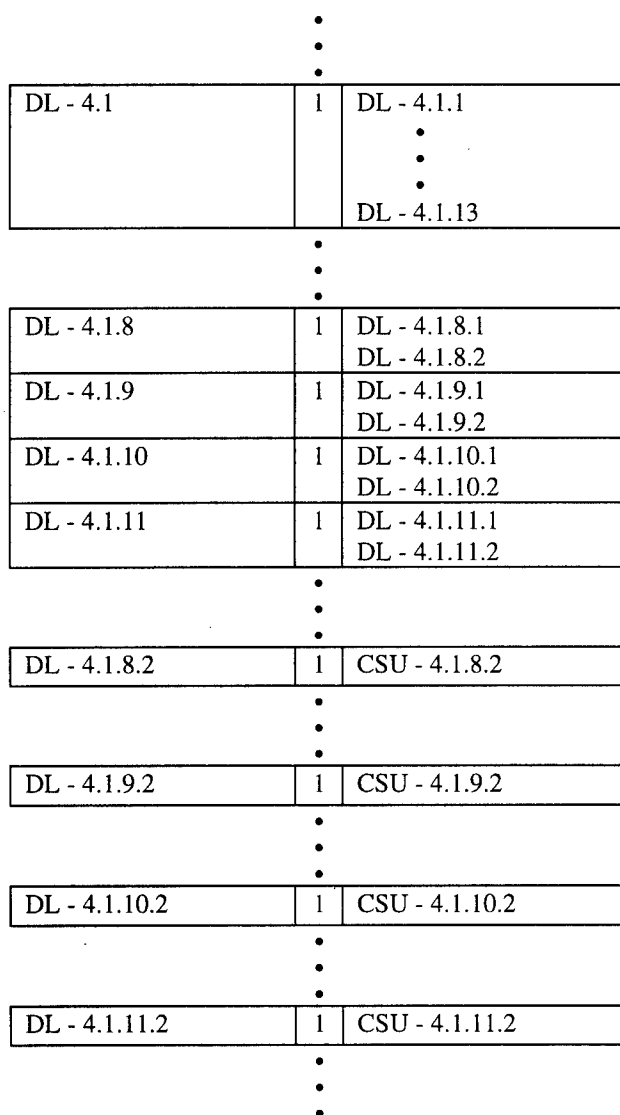


Figure 35. LLD to Code Database Entries for 1553 Driver

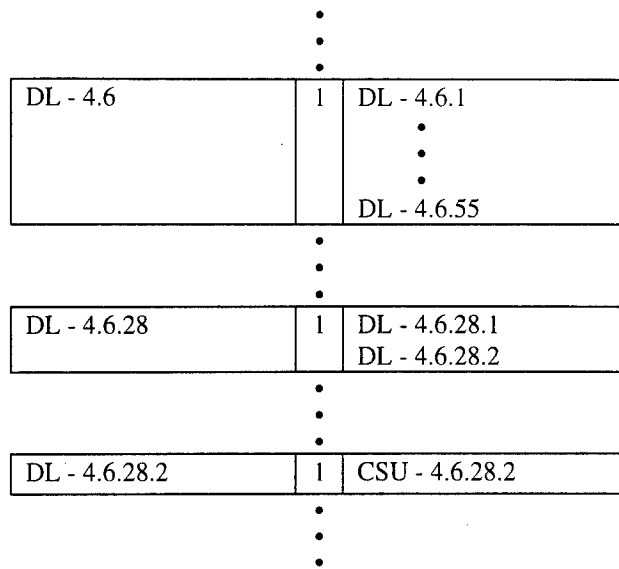


Figure 36. LLD to Code Database Entries for POBIT Utility

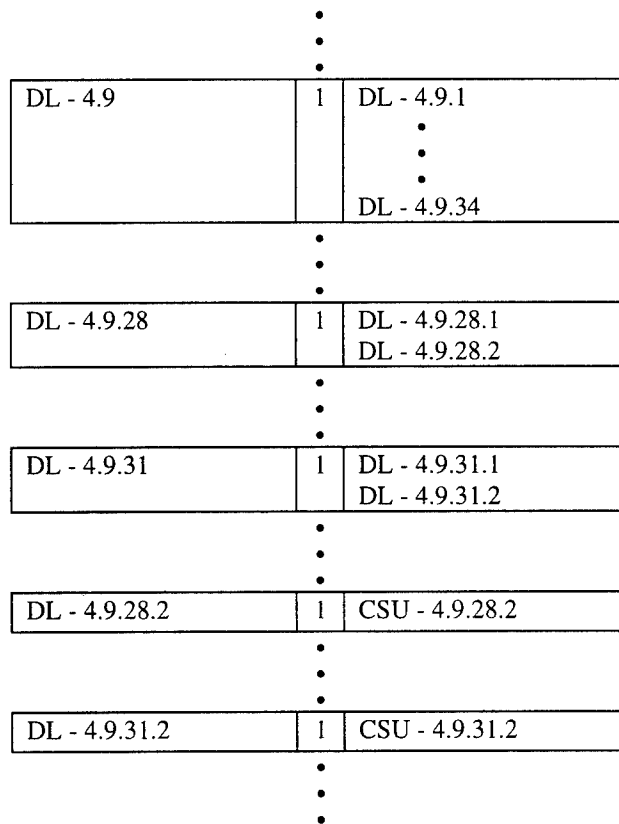


Figure 37. LLD to Code Database Entries for MIBIT Utility

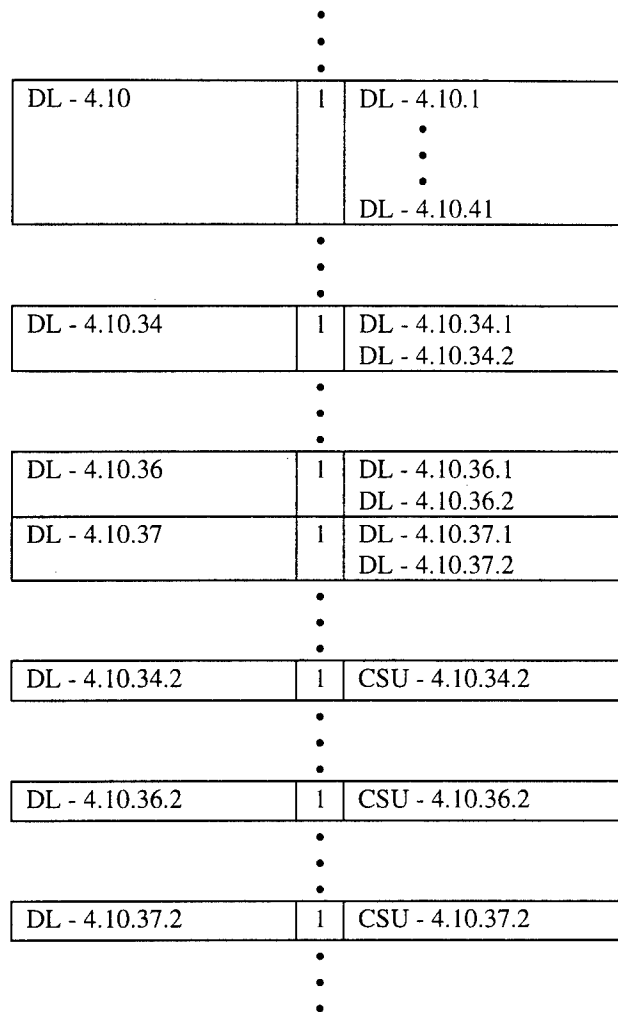


Figure 38. LLD to Code Database Entries for Common_BIT Utility

The CSUs traced thus far are listed below. The CSUs are described by the CSC in which they are contained and the name of the actual procedure or function in code. The code for the CSUs that are traced in the demonstration is contained in Appendix B. The code is included in this research for completeness, to show the entire path from the statement in the ORD to the actual code.

CSU - 4.1.8.2 : 1553 Driver, CDI_Open

CSU - 4.1.9.2	: 1553 Driver, CDI_Close
CSU - 4.1.10.2	: 1553 Driver, CDI_Read
CSU - 4.1.11.2	: 1553 Driver, CDI_Write
CSU - 4.6.28.2	: POBIT Utility, RRPT
CSU - 4.9.28.2	: MIBIT Utility, RRPT
CSU - 4.9.31.2	: MIBIT Utility, Run_1553_Test_Group
CSU - 4.10.34.2	: Common_BIT Utility, Common_BIT_OLBT
CSU - 4.10.36.2	: Common_BIT Utility, Common_BIT_MCPT
CSU - 4.10.37.2	: Common_BIT Utility, Common_BIT_RAPT

5.8.7 Software Requirements to Validation Tests. Software requirements RO - 3.2.2 and RO - 3.2.8 are decomposed into several subrequirements. The software requirement for the 1553 Data Bus driver, RO - 3.2.2, is decomposed into RO - 3.2.2.1 through RO - 3.2.2.5. Each of the five subrequirements is decomposed further, and eventually yields 62 lowest-level requirements. Once again, refer to Figure 28 for a partial decomposition and description of RO - 3.2.2.

As before, for this demonstration, only RO - 3.2.2.1 and RO - 3.2.2.2 will be traced to the validation tests in the demonstration. Figure 39 shows the database entries for the traces of the software requirements to the validation tests. The decomposition of software requirements RO - 3.2.2.1 and RO - 3.2.2.2 were shown in Figures 28 and 30.

•		
•		
•		
RO - 3.2.2.2.1.1	1	TR - 3.3.3.5.3.2 TR - 3.3.3.5.5.1
RO - 3.2.2.2.1.2	1	TR - 3.3.3.5.3.3
RO - 3.2.2.2.1.3	1	TR - 3.3.3.5.3.1
RO - 3.2.2.2.1.4	1	TR - 3.3.3.5.3.1
RO - 3.2.2.2.1.5	1	TR - 3.3.3.5.3.2 TR - 3.3.3.5.3.3
RO - 3.2.2.2.1.6	1	TR - 3.3.3.5.3.3
RO - 3.2.2.2.1.7	1	TR - 3.3.3.5.4.1
RO - 3.2.2.2.1.8	1	TR - 3.3.3.5.4.1
RO - 3.2.2.2.1.9	1	TR - 3.3.3.5.7.1
•		
•		
•		
RO - 3.2.8.1.4.2.17	1	TR - 3.3.3.5.11.1
•		
•		
•		
RO - 3.2.8.2.3.1.1	1	TR - 3.3.3.5.13.16
•		
•		
•		
RO - 3.2.8.2.3.3.1.1	1	TR - 3.3.3.5.15.5
•	•	•
•	•	•
•	•	•
RO - 3.2.8.2.3.3.1.7	1	TR - 3.3.3.5.15.5
•		
•		
•		
RO - 3.2.8.2.3.3.3.1	1	TR - 3.3.3.5.15.3
•	•	•
•	•	•
•	•	•
RO - 3.2.8.2.3.3.3.6	1	TR - 3.3.3.5.15.3
•		
•		
•		
RO - 3.2.8.2.3.3.5.1	1	TR - 3.3.3.5.15.7
RO - 3.2.8.2.3.3.5.2	1	TR - 3.3.3.5.15.7
RO - 3.2.8.2.3.3.5.3	1	TR - 3.3.3.5.15.7
RO - 3.2.8.2.3.3.5.4	1	TR - 3.3.3.5.15.7
RO - 3.2.8.2.3.3.5.5	1	TR - 3.3.3.5.15.7
•		
•		
•		

Figure 39. Database Entries for Software Requirements to Validation Tests

The text for two of the validation tests are presented below. The text of the validation tests are provided to show the depth and breadth of the test process that ensures the requirements are satisfactorily met.

TR - 3.3.3.5.13.16 : 1553 Receiver RAM Test

This test verifies correct operation of the 1553 Receiver RAM TEST module (RRPT). The RRPT tests the entire 1553 Receiver RM address space for address, data, coupling, or stuck-at bit faults. This test will perform the following:

- a) Verify the RRPT runs in the IOP,
- b) Verify RRPT runs during POBIT and MIBIT,
- c) Confirm the RRPT tests 1553 Receiver RAM for both chipsets as required,
- d) Confirm RRPT ability to verify required failure recording in POBIT and MIBIT mode, and
- e) Confirm RRPT verifies continuation requirements after failure [Loc96c].

TR - 3.3.3.5.15.5 : 1553 Off-line Loopback Test

This test verifies the correct operation of the 1553 Off-line Loopback Test module (OLBT). The OLBT tests that the ACE BC Off-Line Self-Test passes and that the 1553 chipsets are capable of transmitting and receiving messages on both mission buses while in Self-Test mode. This test will:

- a) Verify the OLBT runs in the IOP only,
- b) Verify OLBT runs during EPOBIT and MIBIT,
- c) Verify that OLBT declares a fault if the Off-Line Self-Test fails,
- d) Verify that OLBT declares a fault if transmission or receipt of messages is unsuccessful on either bus in Self-Test Mode,
- e) Verify that OLBT performs required failure recording, and
- f) Verify that OLBT meets continuation requirement after failure [Loc96c].

This concludes the demonstration of complete traceability of a single requirement from the ORD to the code, and from the software requirements to the validation tests.

The next step is to perform the software effectiveness calculation, which is accomplished in the next section.

5.9 Effectiveness of 1553 Data Bus Software

5.9.1 Introduction. The software effectiveness calculation below is only for the portion of the 1553 Data Bus software used in the demonstration of SETA. For the purposes of the demonstration, it was necessary to focus on a select portion of the software requirements, since tracing all the requirements would have become unwieldy.

5.9.2 Effectiveness Components. The following effectiveness components were taken from the database entries in the demonstration. Listed below are the component name, the number of complete traces divided by the number of elements that required a trace, and the result given as a percentage:

Requirements to HLD:	21/21	= 100%
HLD to LLD:	4/4	= 100%
LLD to Code:	10/10	= 100%
Requirements to Tests:	29/29	= 100%

The results of validation tests are not available at this time, since delivery of the CIP from Lockheed Martin will not officially occur until a year from now [Est96].

Therefore, the component for the effectiveness value for the percentage of validation tests satisfactorily passed is 0%. Of course, multiplying these components together yields an overall effectiveness value of 0%. This overall effectiveness rating is misleading, since

the software has some degree of effectiveness (based on the other components), and the data on the percentage of validation tests passed exists, but is just *not available*.

5.9.3 Significance of Overall Effectiveness Value. This overall effectiveness value of 0% is insignificant in terms of describing the state of the software. First of all, the traceability information *must be available* to make any worthy effectiveness evaluation using SETA. Secondly, it is understood that SETA is being applied *after-the-fact*, so it should not be surprising to see the traceability components at 100%. The demonstration emphasizes the fact that the traceability data must be maintained from the onset of development, and all the data must be available to make a credible overall effectiveness evaluation.

5.10 Summary

5.10.1 Introduction. The purpose of this chapter was to demonstrate SETA, the approach to evaluate software effectiveness by using traceability of requirements, that was developed in Chapter 4. Traceability was successfully demonstrated using data from the development process of the software for the 1553 Data Bus, which connects various components of the avionics system aboard the C-17 aircraft.

After describing the process that led to the decision to demonstrate SETA, and the search for traceability data, some background information was provided on the avionics system components of the C-17. More detailed background information was provided for the 1553 Data Bus, a major component of the C-17 avionics system. Then, the correlation between the software development terminology at the C-17 SPO and SETA

was outlined. Traceability was demonstrated by completely tracing selected requirements for the 1553 Data Bus from the ORD to the code, and from selected software requirements for the 1553 Data Bus to the validation tests. Finally, the software effectiveness calculation was determined, although it led to a misleading overall effectiveness value of 0%. The process of applying actual data to SETA revealed some interesting information about requirements traceability and SETA itself.

5.10.2 Traceability Must Start Early. The demonstration showed the importance of one of the assumptions in SETA; traceability must be maintained as the software is developed, used, and maintained. The job of maintaining complete requirements traceability is made much easier if it is done incrementally, as the software is developed. This small demonstration also showed how difficult it is to establish requirements traceability on a program that has been in the development stages for many years. The C-17 SPO has dedicated one person, full-time, to establish complete requirements traceability for the operating system utilities of the CIP; after almost two years, the effort is far from complete [Est96].

5.10.3 Additional Benefits of Traceability. In addition to evaluating software effectiveness as it is defined in this research, traceability offers additional benefits to the software developer. Traceability allows the developer to see the impact that change to one artifact of the software has on other artifacts. Also, traceability may serve as an indicator of progress in the software development process; for example, progress can be measured by the percentage of software requirements that are traced to the design. Additional benefits of traceability are discussed in further detail in Chapter 6.

5.10.4 Limitations Revealed by the Demonstration. The demonstration revealed some limitations to SETA. This is beneficial to the research, since these “bugs” would have to be addressed before SETA is developed into a full methodology. A demonstration with actual software development data from a real-world project is an appropriate way to “exercise” SETA and uncover its limitations. The limitations of SETA are discussed in further detail in Chapter 6.

6. Conclusions and Recommendations

6.1 Introduction

This chapter begins with a summary of the research, then revisits the objectives and questions from previous chapters to determine whether or not they were adequately addressed by the research. Section 6.4 outlines some additional benefits to establishing and maintaining requirements traceability throughout the software development process. The next section, 6.5, addresses the issue of requirements traceability in military documentation, and suggestions for future research are presented in Section 6.6. Final comments are offered in Section 6.7.

6.2 Research Summary

6.2.1 Overview. The focus of the research changed when it was determined that virtually no information existed on software effectiveness. Initially, the plan was to review the available information on software effectiveness and the methodologies to evaluate software effectiveness. Then, from this information, a working definition of software effectiveness and an evaluation approach would be outlined. Unfortunately, there was very little information on software effectiveness, so the focus changed to the *creation* of a definition for software effectiveness. Based on the proposed definition, an approach to evaluate software effectiveness was outlined. Since this is a new approach to software effectiveness, it is admittedly a “straw man” that is subject to criticism. It was AFOTEC’s desire to obtain a developed approach to evaluate software effectiveness.

As created in this research, the definition of software effectiveness follows a simple, logical pattern. Software effectiveness refers to how well the software performs. The desired software performance is specified by the software requirements; therefore, software effectiveness specifies to what extent the software requirements are met.

SETA, the approach to evaluate software effectiveness, also follows a simple, logical pattern. If software effectiveness is defined by the degree to which the software requirements are met, one way to ensure the requirements are met is to trace the requirements through all the phases of the software development process. Assuming complete and correct requirements, by completely tracing the software requirements to the design and then to the code, and also tracing from the requirements to validation tests, the requirements will be satisfied. The degree to which the requirements are satisfied is the “degree of traceability” between selected software artifacts, i.e., the percentage calculated from the number of the traces that actually *do* exist and the number of traces that *should* exist. The degrees of traceability between selected software artifacts, coupled with the percentage of validation tests passed satisfactorily, determine the effectiveness of the software.

It was shown that the traces could be implemented in a database, and the degree of traceability is calculated by searching such a database for the number of complete traces, and dividing by the total number of software elements (requirements, design components, etc.) that should be traced.

SETA was demonstrated with actual data from a small portion of the C-17 avionics software. The demonstration revealed some minor limitations in SETA as well as the implementation. These are outlined below.

6.2.2 Limitations of SETA. The demonstration in Chapter 5 revealed some minor limitations that must be addressed prior to developing SETA into a full methodology. The largest limitation is the inability to validate SETA as part of this research. Although SETA makes sense according to the definition of software effectiveness and follows a logical progression, the lack of a validation step may call the credibility of SETA into question. SETA may be validated incrementally by applying the approach to a small software subsystem, assessing the utility of SETA, then applying the approach to larger and larger portions of a system. During this incremental validation of SETA, problems with the approach would be resolved and SETA could then be validated on a complete system.

There are two other limitations in the finer details of SETA. In tracing the lowest-level elements from one software artifact to a highest-level element of another artifact, the details of the lowest-level elements are obscured by the generality of the highest-level element. Consider an example from the demonstration; some of the BITs, such as the 1553 RAM Test, were decomposed into extremely detailed requirements. These minute requirements included testing individual bits of registers and comparing registers for equality. When these details are traced to the HLD element "1553 RAM Test", they are accounted for, but it may not be clear where they are specifically addressed until the "1553 RAM Test" HLD element is decomposed into smaller subelements. Also, what the

“highest-level” element in the next artifact may be open to interpretation. For example, should the software requirements from the 1553 RAM Test be traced to the HLD element for the 1553 RAM test? Should the requirements be traced to the “1553 Tests” HLD element, then decomposed into the 1553 RAM Test and other tests? Perhaps the software requirements should be traced to the HLD element “Tests”, and decomposed from that point. This obfuscation can be remedied by tracking the detailed subelements that are traced to a highest-level element during the decomposition. In short, these detailed subelements could be tracked with an *intra*-trace to account for the details during decomposition *within* an artifact, as opposed to the *inter*-traces that establish the relationship *between* artifacts that have been discussed throughout this research.

Another limitation involved the partial traces in the database. A partial trace in the database is indicated by the appearance of elements in the “element(s) traced to” column and a 0 for the complete trace flag for that entry. Defined in this way, “partial traceability” does not differentiate between the two following extreme cases: 1) an element that needs to be traced to *one* additional element for complete traceability and 2) an element that needs to be traced to *ten* (or *one hundred*) additional elements for complete traceability. This minor limitation could possibly be addressed by inserting a fractional value in the complete trace flag field in the database entry. This fractional value would have to be determined by an experienced software developer that was familiar with the project, and the value would be just an estimate. This value would indicate how complete the trace is in its present state; therefore it would determine how many additional elements would need to be added to the “element(s) traced to” field to

complete the trace from the element in the “element traced from” field. Unfortunately, this introduces more subjectivity to SETA.

A final limitation to SETA concerns integration testing. The only traces in SETA that involved testing were from the software requirements to the validation tests and from the lowest-level LLD elements to the unit tests. A more thorough traceability approach would have included a combination of higher-level LLD elements traced to integration tests, to ensure the individual CSUs functioned together properly. This limitation, as well as the other limitations mentioned above, do not discredit SETA; they merely point out issues that must be addressed before developing SETA into a full methodology.

6.2.3 Limitations of Database Implementation. There are also limitations to the database implementation that must be addressed before it can be used to monitor traceability and evaluate software effectiveness. As stated before, this database implementation is not to be taken as a model of efficiency or optimization; it would have to be analyzed by someone with specialized skills in database design before it could be developed into a usable product.

The biggest drawback of the database implementation is the duplication of data during decomposition. Each decomposed element has its subelements “copied” back to the “element to be decomposed” column of another database entry. Refer to Figure 40 for an example.

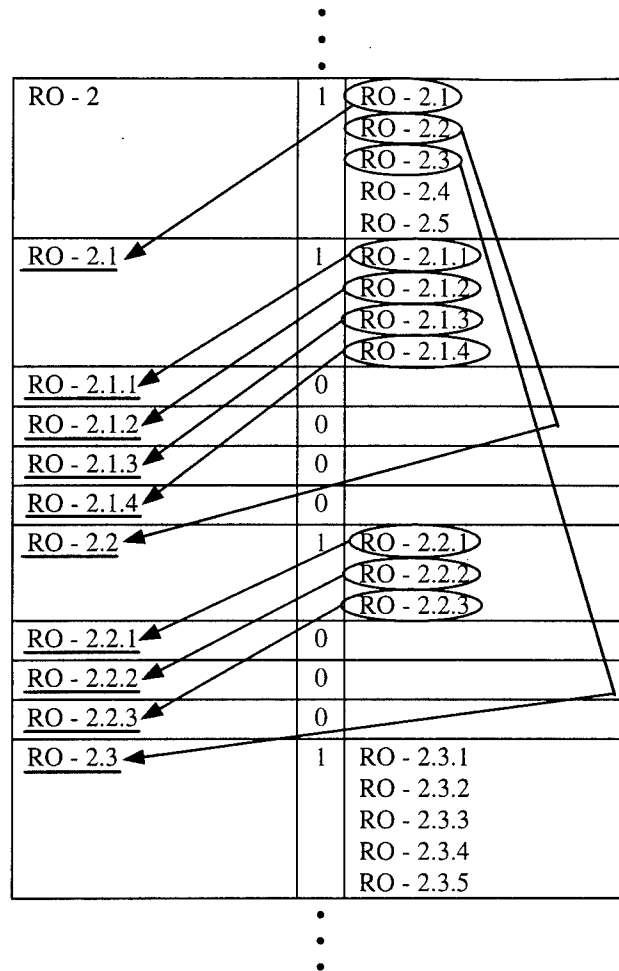


Figure 40. Example Decomposition Database Entries

Figure 40 shows the duplication of data quite clearly as each decomposition in the right column is transferred to the “element to be decomposed” column on the left. Perhaps a more efficient method of database implementation would be to conceptually “grow” the database horizontally in addition to vertically. Refer to Figure 41 for a possible alternate database design.

• • •		• • •		• • •		• • •	
RO - 2	1	RO - 2.1	1	RO - 2.1.1	1	RO - 2.1.1.1 RO - 2.1.1.2 RO - 2.1.1.3 RO - 2.1.1.4	• • •
				RO - 2.1.2	1	RO - 2.1.2.1 RO - 2.1.2.2 RO - 2.1.2.3	• • •
				RO - 2.1.3	1	RO - 2.1.3.1 RO - 2.1.3.2	
				RO - 2.1.4	1	RO - 2.1.4.1 RO - 2.1.4.2 RO - 2.1.4.3	• • •
		RO - 2.2		RO - 2.2.1	1	RO - 2.2.1.1 RO - 2.2.1.2 RO - 2.2.1.3 RO - 2.2.1.4	• • •
• • •		• • •		• • •		• • •	

Figure 41. Possible Alternate Database Design

Another limitation with the implementation of SETA is the overhead required to determine the existence of a trace in the database. To distinguish between a trace and decomposition information, SETA requires a comparison between the element codes in columns (1) and (3) of the database. If the codes are the same, the entry is decomposition information; if they are different, the entry is trace information. This could be remedied by a flag for each entry indicating whether the entry is a trace or decomposition information.

Despite the minor limitations in SETA and the implementation, the research adequately addressed both the goals and questions outlined in the previous chapters.

6.3 Addressing Objectives and Questions from Previous Chapters

6.3.1 Meeting Research Objectives. The primary objectives of this thesis were to develop a working definition of software effectiveness and outline an approach to evaluate software effectiveness. These goals came directly from the problem statement in Section 1.2, which is repeated below.

AFOTEC does not have a methodology to directly address the evaluation of software effectiveness of the software portion of new systems acquired by the Air Force and currently depends on the evaluation of system effectiveness to determine software effectiveness.

The primary objectives were detailed into four objectives, which were initially presented in Chapter 1. A lack of information on software effectiveness in literature and in practice led to a modification of these objectives in Section 3.5. The four objectives of the modified research plan are addressed below.

1) Develop a working definition of software effectiveness.

This objective was met in the subsections leading up to Section 3.10, where software effectiveness was defined as the degree to which the software requirements are satisfactorily met.

2) Research effectiveness in other areas of study, including AFOTEC's definition of system effectiveness. Also research other performance-based software attributes and activities such as software quality, software reliability, software testing, and software verification and validation.

This objective was thoroughly addressed in Chapters 2 and 3.

3) Since there are no software effectiveness methodologies to review, develop one approach to evaluate software effectiveness to recommend to AFOTEC/SAS to develop into a full methodology.

This objective was addressed in Chapter 4, with the development of SETA to evaluate software effectiveness through requirements traceability. A database implementation for SETA was also outlined in this chapter. In addition, database operations were described that allow the retrieval of information, such as an effectiveness value, for the software product being evaluated.

4) Demonstrate the operation of the recommended approach to evaluate software effectiveness.

This objective was addressed in Chapter 5. The traceability portion of SETA was addressed in Section 5.8 and the software effectiveness calculation was completed in Section 5.9.

6.3.2 Answering Research Questions. While focusing on the research objectives, some questions were raised that concerned AFOTEC, the sponsor of this research. The four research questions from Section 1.4 are addressed below.

1) Does this software effectiveness evaluation support AFOTEC's system effectiveness evaluation?

Yes, this evaluation approach supports AFOTEC's *system* effectiveness evaluation by ensuring the *software* requirements (which are a large part of the system requirements) are met. After all, AFOTEC's definition of system effectiveness, as specified in Section 3.7, is defined as meeting *system* operational requirements.

2) How can current software development practices facilitate the evaluation of software effectiveness?

Assuming SETA is used to evaluate software effectiveness, the following software development practices would facilitate the evaluation:

- Unique identification of elements within software artifacts.
- Requirements traceability
- Requirements management
- Peer reviews

These practices can aid in the evaluation of software effectiveness since they all ensure the proper application of SETA. SETA requires unique identification of the software elements (requirements, design components, etc.). Any attempt at requirements traceability would aid in establishing the database implementation. SETA assumes correct and complete requirements; in the real world, this assumption can be addressed by aggressive requirements management. Finally, peer reviews aid in any approach to improve software; decomposition and trace information can be “peer reviewed” in the same manner as requirements, design, or code.

3) Can the effectiveness evaluation be used during the software development process as an early indicator of the software’s effectiveness, i.e., before it reaches AFOTEC for OT&E?

Yes, the effectiveness evaluation can serve as an early indicator of software effectiveness. From the onset of development, the components of effectiveness described in Section 4.9.1 can be used as early indicators. As software development continues, progress can be monitored by reviewing the degrees of traceability between the requirements and the design, the design and the code, etc. Although a final effectiveness

value may not be useful before the project has entered the coding and testing phases of development, a substantial amount of information can be gathered during the initial stages of software development.

4) Can the effectiveness evaluation be used to determine the product's readiness to *enter* OT&E?

Yes, indirectly. The effectiveness evaluation can be used to determine if the software is *not* ready to enter OT&E. As stated previously, SETA provides a substantial amount of information about the software during its development. Consider a software project that is supposedly ready to enter OT&E. By following SETA, it is determined that only half the software requirements are addressed by the design. In addition, only half the requirements are traced to validation tests, and all these tests have failed. This information implies that something is obviously wrong with the software, and it is certainly not ready to enter OT&E.

6.4 Additional Benefits of SETA

6.4.1 Introduction. While developing SETA (and the implementation), some additional benefits of requirements traceability were uncovered. One such benefit is that SETA can serve as an early indicator and can provide insight to Air Staff (one of AFOTEC's concerns) as well as the SPO during software development. Other benefits were also discovered with the database implementation.

6.4.2 Additional Benefits of Database Implementation. Searching on selected element codes within the database can reveal a considerable amount of information about the software being developed. For example, consider the software requirements. The

software requirements can be identified by function and the database can be searched to determine what functions have been addressed in the design. If the customer is concerned over the user-interface, the database can be checked to see how many of the user interface requirements have been designed and tested. In addition, by selecting element codes unique to a certain software development process, software requirements can be sorted by design team, and employee progress can be monitored as requirements are addressed by the design. Similar database searches can focus on uniquely identified requirements that involve safety or performance issues.

6.4.3 Additional Benefits of Tracing Requirements. Besides ensuring requirements satisfaction, there are many benefits in tracing requirements. In fact, it is considered by some to be a necessary part of large-scale software development [Dav95, Lin93, Wat94]. Requirements traceability, unlike testing and validation, improves confidence in meeting requirements *before* the product is complete [Cro96]. Also, traces highlight the effect that a change to one artifact can have on the other artifact in the software development process. For example, if a requirement changes, the traces indicate the impact on the design, the code, and the tests used to validate that requirement. In addition, traceability helps prevent “bells and whistles” from being added into the design or the code if the design elements or code modules are not traced from a specific requirement. Furthermore, the absence of traces from requirements to validation tests exposes holes in test coverage. In summary, Lindstrom emphasizes that requirements traceability “can be one of the best mechanisms for ensuring completeness and monitoring progress” during software development [Lin93]. These additional benefits can be realized in SETA as a result of a natural occurrence within the database

implementation, which will be referred to as a *backtrace*. In order to explain what backtraces are, and what benefits can be derived from them, it is necessary to use the logical structures and arrows initially presented in Chapter 4.

Put simply, backtraces refer to traces that are opposite in direction to the “forward” traces discussed in this research. Backtraces are a consequence of the creation of the traces used in the determination of software effectiveness. While not addressed at all in SETA, since they make no contribution to the evaluation of software effectiveness, backtraces are of significant importance to warrant a brief discussion. An example of backtraces for HLD to software requirements is shown in Figure 42.

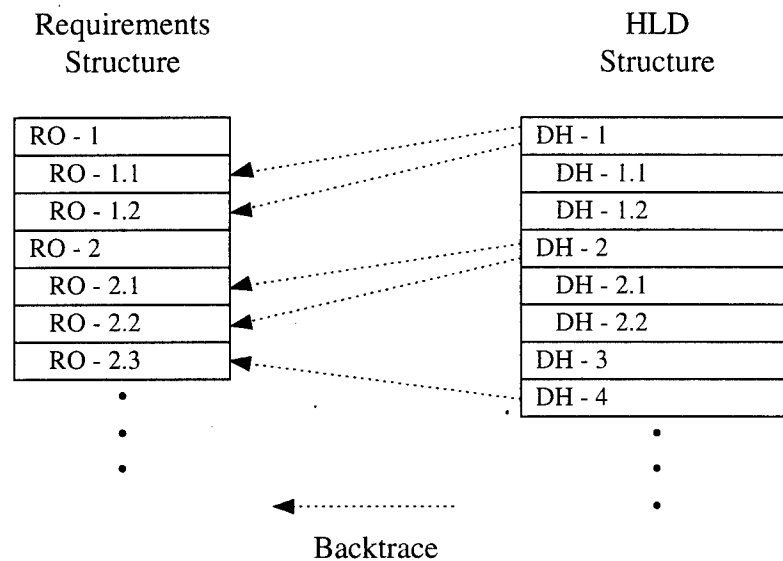


Figure 42. Example of HLD to Requirements Backtraces

Backtraces can be analyzed to reveal more information about the software being developed. First, backtraces can be used to identify errors in traces. Looking at how backtraces connect design elements to requirements provides a way to double-check the

established traces. In Figure 42, if RO - 2.3 is *actually* supposed to be traced to DH - 3 instead of DH - 4, analyzing all the backtraces from DH - 4 and DH - 3 will reveal this incorrect trace. Granted, this can also be found by analyzing the requirement to HLD traces, but the backtraces provide this “error-checking” from another perspective.

More importantly, backtraces reveal excess elements in the software artifacts.

Consider Figure 43, which shows the *traces* from Figure 42.

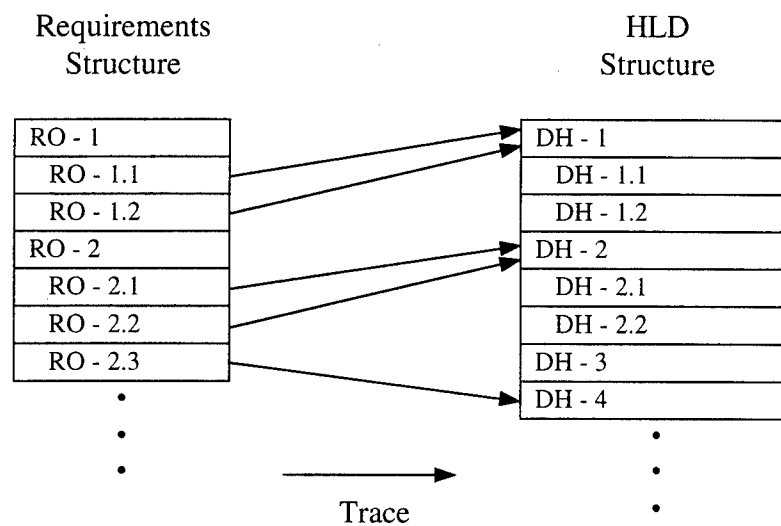


Figure 43. Traces from Example of HLD to Requirements Backtraces

Assume the HLD is complete, and the developers are ready to begin a more detailed design. By analyzing the backtraces, and eliminating each design element that is traced back to a software requirement, what remains are excess design elements. From Figure 42, DH - 3 is an excess design element. This same procedure can be used to find excess code (sometimes known to as “dead” code) and excess tests. These excess elements are sometimes referred to as “gold-plating” that waste valuable development resources. It

should be noted that these excess elements cannot be arbitrarily deleted, since they may be traced to other elements in the development process.

There are other reasons why "excess" design elements may appear in the database. These design elements may be a result of requirements that were changed or eliminated, and it may be more difficult to remove the design elements than to leave them in the design. Reuse is another reason why excess design elements may exist in the database; if a design component is reused that can adequately address certain software requirements, yet has "extra" features that are not necessary in the design, these features will appear as "excess" elements and will be discovered by backtraces.

Finally, backtraces can aid in regression testing. Consider Figure 44, which shows the traces from Figure 43 with a slight modification. Software requirement RO - 2.2 has been changed. It has been determined that a slight modification of DH - 4 will satisfactorily address this changed requirement. Therefore, to reflect this change in traceability, the trace from RO - 2.2 to DH - 2 has been changed to now trace to DH - 4. Figure 44 reflects the updated trace for RO - 2.2.

By using backtraces, all the requirements that trace to the modified design element DH - 4 can be identified. In this case there is one requirement that traces to the modified design element: RO - 2.3. As a part of regression testing, all the validation tests which this requirement traces to must be reaccomplished, since the change in the design element it traces to may no longer address the requirements satisfactorily.

In summary, backtraces are a byproduct of establishing requirements traceability to measure software effectiveness. Although backtraces are not part of the software

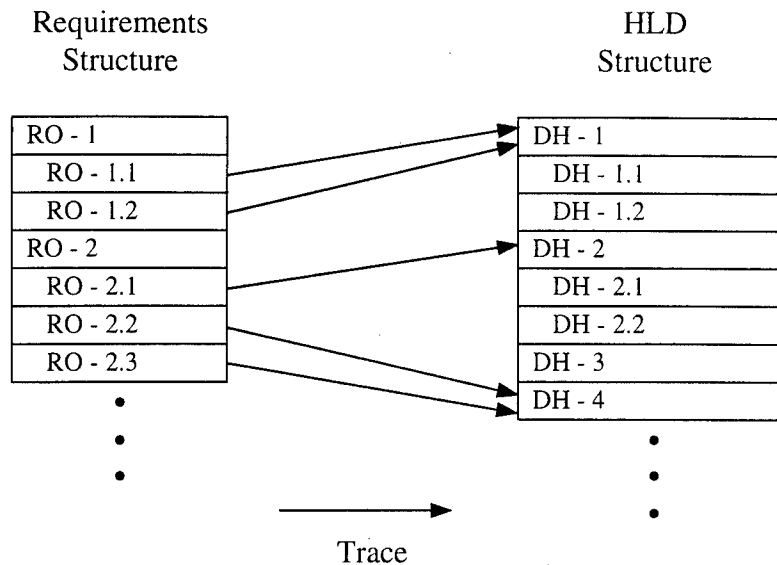


Figure 44. Example of HLD to Requirements Traces with Requirement Change

effectiveness evaluation, they warrant a brief discussion since they can be useful in detecting “gold-plating” in the design, “dead” code, or unused tests. Backtraces are also used to validate existing traces between software artifacts. Lastly, backtraces are used after requirements and design changes to aid in regression testing. Incidentally, backtraces can be viewed in the Traceability Matrix in Figure 23 from Section 4.8.6 by looking at the shaded squares along a vertical line connecting a specific HLD element with a software requirement.

6.5 Practicality of Implementing SETA

As stated previously, there are virtually no references to software effectiveness in the available documentation. Since SETA is based on requirements traceability, the practicality of implementing the approach may be determined by the availability of information on requirements traceability in current documentation and in practice.

There are references to requirements traceability in AFOTEC's software evaluation documentation. There are also references to requirements traceability in MIL-STD-498, and the Data Item Descriptions (DIDs) mentioned throughout the standard [DoD94]. The references in MIL-STD-498 are significant, since up until recently, all software development efforts were required to adhere to this standard. A search for documentation on requirements traceability uncovered two articles stating that the DoD *mandated* requirements traceability [Cro96, Wat94]. Subsequently, references to requirements traceability were found in the one DoD regulation that was reviewed [DoD96b].

6.5.1 Requirements Traceability in AFOTEC Documents. Requirements traceability is addressed in two AFOTEC pamphlets. In the *Software Operational Assessment Guide*, AFOTEC PAM 99-102, Volume 8, requirements traceability from the ORD to the system specification is considered "desirable" [AFO94b]. In addition, complete requirements traceability by the development contractor is considered an "absolute necessity" for complex systems [AFO94b].

The *Software Maintainability Evaluation Guide*, AFOTEC PAM 99-102, Volume 3, describes traceability as "connecting programming information between all levels of lesser and greater detail" and "a clearly defined trail between top-level requirements and detailed implementation" [AFO94a]. Traceability is one of three documentation characteristics that are evaluated to determine maintainability; the other two characteristics are organization and descriptiveness [AFO94a]. In summary, AFOTEC

not only addresses requirements traceability as part of their own maintainability evaluation, they also demand requirements traceability from the development contractor.

6.5.2 Requirements Traceability in MIL-STD-498. The former standard for software development and documentation for the military, MIL-STD-498, has many references to requirements traceability. For each document in the software development process, requirements specification, design specification, test plans, etc., requirements traceability plays a major role. MIL-STD-498 utilizes the DIDs to describe the format of the previously mentioned documents. Requirements traceability is an established part of the following DIDs:

- 1) System/Subsystem Specification (SSS)
- 2) System/Subsystem Design Description (SSDD)
- 3) Software Requirements Specification (SRS)
- 4) Software Design Description (SDD)
- 5) Software Product Specification (SPS)
- 6) Software Test Plan (STP)
- 7) Software Test Description (STD)

This is precisely the requirements traceability data needed for the software effectiveness approach developed in Chapter 4. Unfortunately, MIL-STD-498 allows the documentation to be tailored, allowing the software developer to eliminate the sections concerning traceability from each of the documents mentioned above.

6.5.3 Requirements Traceability in DoDR 5000.2-R. The most profound reference to requirements traceability was located in DoDR 5000.2-R [DoD96b], the subject of which is provided below.

5000.2-R SUBJECT: Mandatory Procedures for Major Defense Acquisition Programs (MDAPs) and Major Automated Information System (MAIS) Acquisition Programs

The most shocking revelation in this document was that requirements traceability is not only mandatory, *it is used to certify readiness for OT&E*. The text of the particular paragraph is provided below.

3.4.3 Certification of Readiness for Operational Test and Evaluation.

The developing agency shall prepare a DT&E Report, and formally certify that the system is ready for the next dedicated phase of operational test and evaluation to be conducted by the DoD Component operational test activity. The developing agency shall also provide software maturity criteria and performance exit criteria necessary for certification for operational test. Risk management metrics, measures, indicators, and associated thresholds shall include cost, schedule, **requirements traceability**, and fault profile. A mission impact analysis of unmet metrics shall be completed before certification for operational tests [DoD96b].

All the references to requirements traceability in the documents above begs the question:

“Why is there no complete traceability data for large software development projects, if requirements traceability is: 1) required by DoD regulation, 2) stressed in the former standard for software development and documentation (MIL-STD-498), and 3) considered an ‘absolute necessity’ by AFOTEC?”

Obviously, this level of traceability would have assisted the author in developing the Chapter 5 demonstration. However, the more disconcerting thought involves all the software development projects that have gone over-budget or beyond schedules because of inadequate software development practices, *including requirements traceability*.

6.6 Recommendations for Future Research

There are many directions in which to continue this research effort; several are listed below with some explanatory comments.

1) Survey software development organizations in the military as well as industry, and document the efforts at complete requirements traceability.

Since requirements traceability is an essential factor in successful software development, it would be useful to know the degree to which it is being accomplished in the military as well as industry. This is especially true in the software development efforts for the military, since requirements traceability is mandatory in all major software projects.

2) Expand the software effectiveness approach developed in this research to determine *system* effectiveness.

This expansion would require addressing the limitations to SETA and the database implementation described above, and adding more components of traceability. The additional *system* effectiveness components would include traces from ORD to validation tests as well as traces from system requirements to validation tests.

3) Address the largest assumption in SETA concerning complete and correct requirements.

The assumption concerning complete and correct requirements can be eliminated from SETA by adding another step to the software development process. This additional step is an aggressive requirements engineering program. By focusing on requirements

elicitation, requirements management, and requirements validation, complete and correct requirements verge upon a reality, eliminating the assumption in SETA.

4) Address the assumption in SETA concerning complete and correct traceability between software artifacts.

This assumption can be eliminated from SETA by adding yet another step to the software development process. Formal methods may be used to ensure a complete and correct “transition” from one software artifact to another. For example, formal methods can be used to make the transition from software requirements to preliminary design, preliminary design to detailed design, and from detailed design to code.

5) Perform a survey of requirements management tools.

Since there are dozens of commercial software products that aid in managing requirements, perform a survey of the available tools. This survey would include a thorough comparison and contrast of all the products to document strengths and weaknesses, especially in the area of requirements traceability.

6) Implement SETA and apply it to a software development project.

6.7 Final Comments

6.7.1 Importance of the Research. The importance of this research cannot be overemphasized. In addition to addressing AFOTEC’s concerns about a working definition and evaluation method for software effectiveness, as well as their request to add traceability from the ORD and to the unit tests, the research uncovered some insight into the importance of requirements traceability. Granted, the definition and evaluation

method for software effectiveness may not be the *best* answer, but it is *an* answer, where apparently *no* answer existed previously. Perhaps this research can serve as a launching point for additional research on the topic of software effectiveness; then again, perhaps the term “effectiveness” should be dismissed as a descriptive term for software.

6.7.2 *Software Effectiveness; What’s in a Name?* In closing, something must be said about “software effectiveness” and the nagging question from Section 3.3.4: “why is it that no one else in the software development community seems to be concerned about software effectiveness?”

As it turns out, software effectiveness may be just two words put together in an attempt to describe the “goodness” of software. This in no way minimizes the importance of software effectiveness (or this research), since AFOTEC’s *mission* is to evaluate the operational effectiveness of systems (and therefore software) for the Air Force. It may be the case that AFOTEC has inherited the term effectiveness from their previous evaluations of system effectiveness. It may also be time to examine the origin and necessity of the word “effectiveness”.

In his 1981 paper entitled, *Operational Test and Evaluation of Software*, Murch quotes a report from 1970 that uses the term “effectiveness” as one of the five broad categories of OT&E [Mur81]. Murch then describes effectiveness as an attribute that should be assessed to determine the software’s “readiness for operations” [Mur81]. It certainly appears that “software effectiveness” is a direct descendant of “system effectiveness”.

Perhaps the term “software effectiveness” can now be put to rest as a vague term that once had a purpose in software development but, in the light of the maturing discipline of software engineering, no longer has a place. Then, more time and effort can be spent on the policies and practices that improve the “goodness” of software and have been studied extensively, such as software quality assurance, V&V, software process improvement, and requirements traceability.

Bibliography

- [AFO94a] Department of the Air Force. *Software Maintainability Evaluation Guide*. AFOTTECP 99-102, Volume 3. Kirtland AFB, NM: HQ AFOTEC, 15 June 1994.
- [AFO94b] Department of the Air Force. *Software Operational Assessment Guide*. AFOTTECP 99-102, Volume 8. Kirtland AFB, NM: HQ AFOTEC, 15 June 1994.
- [AFO95] Department of the Air Force. *Management of Operational Test and Evaluation*. AFOTTECI 99-101. Kirtland AFB, NM: HQ AFOTEC, 2 October 1995.
- [AFI94a] Department of the Air Force. *Developmental Test And Evaluation*. AFI 99-101. Washington: DoD, 25 July 1994.
- [AFI94b] Department of the Air Force. *Operational Test And Evaluation*. AFI 99-102. Washington: DoD, 22 July 1994.
- [AFI94c] Department of the Air Force. *Test And Evaluation Process*. AFI 99-103. Washington: DoD, 25 July 1994.
- [AMC93] Air Mobility Command. *Operational Requirements Document for C-17 Acquisition*. AMC 002-091. HQ AMC/XPQC, Scott AFB, IL. 15 September 1993.
- [Ash94] Ashqar, Abdelhaleem, and others. "Use of a Group Support System to Evaluate Management Information System Effectiveness," *Journal of Systems and Software*, 24: 267-275 (March 1994).
- [Bow85] Bowen, Thomas P., Gary B. Wigle, and Jay T. Tsai. *Specification of Software Quality Attributes*. Technical Report RADC-TR-85-37 Vol 1. Griffiss Air Force Base, New York: Rome Air Development Center, February, 1985.
- [Cla92] Clapp, Judith A. and Saul F. Stanton. *A Guide to Total Software Quality Control*. Technical report RL-TR-92-316. Griffiss Air Force Base, New York: Rome Laboratory, December, 1992.
- [Cro96] Cross, Gary. "Tracking the Changing Face of System Development," *Real-Time Magazine*, 1: 10-12 (January-March 1996).

- [DAF94] Department of the Air Force. "Guidelines for Successful Acquisition and Management of Software Intensive Systems: Weapons Systems, Command and Control Systems, Management Information Systems", Software Technology Support Center, 1994.
- [DAF95] Department of the Air Force. "Requirements Engineering and Design Technology Report", Software Technology Support Center, 1995.
- [Dav95] Davis, Alan M. "Tracing: A Simple Necessity Neglected," *IEEE Software*, 12: 6-7 (September 1995).
- [Des95] Dessert, Donald M., Colonel, USAF, Test Director and others. "C-17 Initial Operational Test and Evaluation (IOT&E) Final Report." HQ AFOTEC, Kirtland AFB, NM. 27 October 1995.
- [DoD94] Department of Defense. *Software Development and Documentation*. MIL-STD-498. Washington: DoD, 5 December 1994.
- [DoD96a] Department of Defense. *Defense Acquisition Program Procedures*. DoDI 5000.2. Washington: DoD, 15 March 1996.
- [DoD96b] Department of Defense. *Regulation, Subject: Mandatory Procedures for Major Defense Acquisition Programs (MDAPs) and Major Automated Information System (MAIS) Acquisition Programs*. DoDR 5000.2-R. Washington: DoD, 15 March 1996.
- [Ens96] Enslow, Greg. Technical Sales Representative, Microsoft Developer Tool Sales Team, Microsoft, Bellevue WA. Telephone interview. 31 July, 1996.
- [Est96] Estes, Nelson. Avionics Integrated Product Team Member, C-17 System Program Office, Wright-Patterson AFB, OH. Personal interview. 30 September, 1996.
- [Eva88] Evans, Patricia A. and others. "An Instrument for Measuring Effectiveness of Information Systems," *Computers & Industrial Engineering*, 14: 227-236 (1988).
- [Gil85] Gillis, P. D. "Refining Computer-Based Invention Through Computer-Aided Evaluation and 'State-of-the-Art' Tutorial Design," *Journal of Educational Technology Systems*, 13: 315-323 (1984-1985).
- [Gla92] Glass, Robert L. *Building Quality Software*. Englewood Cliffs, New Jersey: Prentice-Hall, 1992.

- [Hed96] Hedstrom, Margaret. Software Developer (former Software Quality Engineering Group member), IBM Software Solutions at Research Triangle Park, IBM, Raleigh NC. Telephone interview. 31 July, 1996.
- [Het88] Hetzel, William C. *The Complete Guide to Software Testing*. Wellesley, Massachusetts: QED Information Sciences, Inc., 1988.
- [Hua95] Huarng, Adam S. "A Comparative Study of Systems Development Effectiveness," *Journal of Computer Information Systems*, 35: 42-49 (Summer 1995).
- [Hum89] Humphrey, Watts S. *Managing the Software Process*. Reading, Massachusetts: Addison-Wesley, 1989.
- [IEE86] Institute of Electrical and Electronics Engineers. *IEEE Standard for Software Verification and Validation Plans*. ANSI/IEEE Std 1012-1986. New York: IEEE, 1986.
- [IEE90] Institute of Electrical and Electronics Engineers. *IEEE Standard Glossary of Software Engineering Terminology*. ANSI/IEEE Std 610.12-1990. New York: IEEE, 1991.
- [ISO91] International Organization for Standardization, International Electrotechnical Commission. *Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for their Use*. ISO/IEC 9126. Switzerland: ISO/IEC, December 1991.
- [Kit89] Kitfield, James. "Is Software DoD's Achilles' Heel?", *Military Forum*, 28-35 (July 1989).
- [Lew92] Lewis, Robert O. *Independent Verification and Validation: A Life Cycle Engineering Process for Quality Software*. New York: John Wiley & Sons, Inc., 1992.
- [Lin93] Lindstrom, David R. "Five Ways to Destroy a Development Project," *IEEE Software*, 10: 55-58 (September 1993).
- [Lo94] Lo, Hong K., and others. "Evaluation Framework for IVHS," *Journal of Transportation Engineering*, 120: 447-460 (May-Jun 1994).
- [Loc96a] Lockheed Corporation. "Computer Program Development Specification for the Operating System Utilities of the C-17A Core Integrated Processor." Prepared by Lockheed Martin Control Systems. Contract No. F33657-95-D-2026. Document No. DEV355A5326. 10 April 1996.

- [Loc96b] Lockheed Corporation. "Computer Program Product Specification for the C-17A Core Integrated Processor Operating System Utilities." Prepared by Sanders, a Lockheed Company. Contract No. F33657-95-D-2026. Document No. 6387531. 24 July 1996.
- [Loc96c] Lockheed Corporation. "Computer Program Test Plan for the Operating System Utilities of the Core Integrated Processor." Prepared by Lockheed Martin Control Systems. Contract No. F33657-95-D-2026. Document No. DEV355A5330. 16 February 1996.
- [McD96] McDonnell Douglas Corporation. "Prime Item Development Specification For the C-17A Computer, Digital CP-2343/AYQ-18." Contract No. F33657-95-R-2026. 19 July 1996.
- [MIL94] Department of Defense. *Software Development and Documentation*. MIL-STD-498. Washington: DoD, 5 December 1994.
- [Mur81] Murch, W.G. "Operational Test and Evaluation of Software," *Proceedings of the IEEE 1981 National Aerospace and Electronics Conference*, 3: 1390-1398 (1981).
- [Nie96] Nieto, Gerald. Manager of Project Management Support, Applications Division, Lockheed Martin, Sunnyvale CA. Telephone interview. 13 August, 1996.
- [Oiv93] Oivo, Markuu. "Incremental Resource Estimation with Real-Time Feedback from Measurement," *Microprocessing and Microprogramming*, 38: 281-289 (1993).
- [PRC94] PRC, Inc. *Software Effectiveness Methodology Study Task Report of Concept Options (Final)*. Contract F29601-89-C-0071. Albuquerque NM: PRC, 21 January 1994.
- [Pre92] Pressman, Roger S. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill, 1992.
- [Pre93] Pressman, Roger S. *A Manager's Guide to Software Engineering*. New York: McGraw-Hill, 1993.
- [Pro95] Proposed Thesis Topic Outline. Topic: Software Effectiveness. Sponsor: Air Force Operational Evaluation Center (AFOTEC), Kirtland AFB, NM. POC: Mr. Jeff Wiltse, 1 November 1995.

- [Rad94] Radford, Donald W. "Volume Fraction Effects in Ultra-Lightweight Composite Materials for EMI Shielding," *Journal of Advanced Materials*, 26: 45-53 (Oct 1994).
- [Ran92] *Random House Webster's College Dictionary*. New York: Random House, Inc., 1992.
- [Ros94] Ross, Jeffrey, Vic Basili, and Mike Berry. "Establishing Measurement for Software Quality Improvement," *IFIP Transactions A: Computer Science and Technology*. A-54: 319-329 (1994).
- [Sca94] Scavo, Frank. "Software Validation for Pharmaceutical and Medical Device Manufacturers," *Proceedings of the 37th International Conference - American Production and Inventory Control Society*. 676-681 (1994).
- [Sch95] Schwab, A. J., B. W. Johnson, and J. B. Dugan. "Analysis Techniques for Real-Time, Fault-Tolerant, VLSI Processing Arrays," *Annual Reliability and Maintainability Symposium 1995 Proceedings*. 137-143 (1995).
- [Sco95] Scott, Judy E. "The Measurement of Information Systems Effectiveness: Evaluating a Measurement Instrument," *Database for Advances in Information Systems*, 26: 43-59 (February 1995).
- [Sei69] Seiler, Karl III. *Introduction to Systems Cost-Effectiveness*. New York: John Wiley & Sons, Inc., 1969.
- [She90] Shepperd, M. "Early Life-cycle Metrics and Software Quality Models," *Information and Software Technology*, 32: 311-316 (1990).
- [Sta91] Stanko, Joseph J. *A Standardized Software Reliability Measurement Methodology*. MS thesis, AFIT/GCE/ENG/91-09. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
- [Tyl91] Tyler, William. "An Overview of MIL-STD-1553B Part I," *Avionics*, 3: 38-43 (March 1991).
- [Vu96] Vu, John. Senior Principal Scientist, Software Engineering Research and Technology Division, Boeing, Seattle WA. Telephone interview. 5 August, 1996.
- [Wal89] Wallace, Dolores R. and Roger U. Fujii. "Software Verification and Validation: An Overview," *IEEE Software*, 6: 10-17 (May 1989).

- [Wal91] Wallmueller, Ernest. "Software Quality Management," *Microprocessing and Microprogramming*, 32: 609-616 (1991).
- [War87] Warthman, James L. *Software Quality Measurement Demonstration Project (I)*. Technical report RADC-TR-87-247. Rome Air Development Center, New York: Rome Laboratory, December, 1987.
- [Wat94] Watkins, Robert and Mark Neal. "Why and How of Requirements Tracing," *IEEE Software*, 11: 104-106 (July 1994).
- [Wel93] Welzel, Dieter and Hans-Ludwig Hausen. "A Metric-based Software Evaluation Method," *Software Testing, Verification and Reliability*, 3: 181-194 (September - December 1993).
- [Zan92] Zane, Thomas and Connell G. Frazer. "The Extent to Which Software Developers Validate Their Claims," *Journal of Research on Computing in Education*, 24: 410-419 (Spring 1992).

Vita

Captain Timothy J. Schalick [REDACTED]

[REDACTED] right next to McGuire AFB, where his father was stationed in the Air Force). Tim graduated from Vineland High School in June of 1980, Lincoln Technical Institute in September of 1981, and enlisted in the United States Air Force in July of 1983. After completing basic training at Lackland AFB, Texas, and computer operator training at Keesler AFB, Mississippi, Tim was stationed for his first assignment at McGuire AFB, New Jersey. Tim married Mary Gore in June of 1985 and the following month they left for Tim's new assignment to the 2114th Communications Squadron at Misawa AB, Japan. In September of 1988, Tim was accepted to the Airman Education and Commissioning Program and attended Arizona State University to complete his Bachelor of Science Degree in Computer Science.

Tim graduated magna cum laude in December of 1991, attended Officer Training School at Lackland AFB, and was commissioned a Second Lieutenant on June 3, 1992. After returning to Keesler AFB to complete Basic Communication-Computer Officer Training, Lt Schalick was assigned to the Air Force Operational Test and Evaluation Center (AFOTEC) at Kirtland AFB, New Mexico. In May of 1995, Lt Schalick entered the Air Force Institute of Technology (AFIT) at Wright-Patterson AFB, Ohio, to pursue a Master of Science Degree in Computer Systems (emphasis on Software Engineering). Upon graduation from AFIT in December of 1996, Captain Schalick was assigned to Air Combat Command's Computer Support Squadron at Langley AFB, Virginia.

[REDACTED]
[REDACTED]

Appendix A: Acronyms and Abbreviations

AFI	Air Force Instruction
AFOTEC	Air Force Operational Test and Evaluation Center
AFOTECI	AFOTEC Instruction
AFOTECP	AFOTEC Pamphlet
AFOTEC/SAS	AFOTEC's Software Analysis Team
AF/TE	Air Force Test and Evaluation
AS	Application Software
BC	Bus Controller
BIT	Built-In Test
CIP	Core Integrated Processor
COI	Critical Operational Issue
CPDS	Computer Program Development Specification
CPPS	Computer Program Product Specification
CPTPI	Computer Program Test Plan
CPTPr	Computer Program Test Procedures
CSC	Computer Software Component
CSCI	Computer Software Configuration Item
CSU	Computer Software Unit
DID	Data Item Description
DoD	Department of Defense
DoDR	DoD Regulation

DT&E	Developmental Test and Evaluation
EMI	Electromagnetic Interference
HLD	High-Level Design
IVHS	Intelligent-Vehicle Highway Systems
LLD	Low-Level Design
OT&E	Operational Test and Evaluation
MIBIT	Maintenance Initiated BIT
MCPT	1553 Mode Code Processing Test
MOE	Measure of Effectiveness
MON	Bus Monitor
MOP	Measure of Performance
MRGT	1553 Register Test
MRXT	1553 Multiple Receive Test
N/A	Not Applicable
OLBT	1553 Off-Line Loopback Test
ORD	Operational Requirements Document
OSU	Operating System Utilities
PIDS	Prime Item Development Specification
POBIT	Power-On BIT
RAPT	1553 RT Address Parity Test
RRPT	1553 Receiver RAM Pattern Test
RT	Remote Terminal

RTAT	1553 RT Address Test
RT/MON	Remote Terminal/Bus Monitor
SDD	Software Design Description
SETA	Software Effectiveness Traceability Approach
SPO	System Program Office
SPS	Software Product Specification
SQA	Software Quality Assurance
SRS	Software Requirements Specification
SSDD	System/Subsystem Design Description
SSS	System/Subsystem Specification
STD	Software Test Description
STP	Software Test Plan
V&V	Verification and Validation

Appendix B: Selected Code from 1553 Data Bus Software

1553 Data Bus Driver, CDI_Open

```
-----
--*** (U) SUBPROGRAM NAME: CDI_Open
--***
--*** (U) EFFECTS: 1553 driver open routine.
--***
--***      The Application Software will call this procedure at
--***      start-up to receive a file descriptor which will then be
--***      used in subsequent read, write and ioctl calls to
--***      reference the appropriate 1553 mission bus.
--***
--*** (U) EXCEPTIONS PROPAGATED: No propagated exceptions are explicitly
--***      raised in this unit.
--***
--*** (U) USAGE CONSTRAINTS: This routine is not to be called directly by
--***      the AS. It is accessed via the VxWorks IO
--***      System.
--***
--*** (U) UNDESIREED EVENTS: None
-----

procedure CDI_Open (Bus      : in Character := 'x';
                   Fd       : out CDI_File_Descriptor_Type;
                   Result    : out CDI_Status_Type) is
    Status      : CDI_Status_Type      := Success; -- Local status
    Local_Fd    : CDI_File_Descriptor_Type := 0;    -- Local file descriptor

    -- Local constants
    Dev_One_Char : constant Character := '1';      -- Dev #1 character value
    Dev_Two_Char : constant Character := '2';      -- Dev #2 character value
    Dev_One_Fd   : constant := 1;                 -- Dev #1 Fd value
    Dev_Two_Fd   : constant := 2;                 -- Dev #2 Fd value
    Invalid_Fd   : constant := 3;                 -- Invalid Fd value
begin
    case Bus is
        when Dev_One_Char => Local_Fd := Dev_One_Fd;
        when Dev_Two_Char => Local_Fd := Dev_Two_Fd;
        when others       => Local_Fd := Invalid_Fd;
    end case;

    -- Error check (Invalid Fd or already opened)
    if (Local_Fd < 1) OR (Local_Fd > Max_Num_Devs) OR
        (Driver.Devs(Local_Fd).Created = False) OR
        (Driver.Devs(Local_Fd).Opened  = True) then
        Local_Fd := Dev_One_Fd;
        Status := Failure;
        -- Set VxWorks error number
    else
        Driver.Devs(Local_Fd).Opened := True;
    end if;
end CDI_Open;
```

```

        Driver.Devs(Local_Fd).Mode           := Idle;
        Driver.Devs(Local_Fd).BC.Frame_Started := False;
        Driver.Devs(Local_Fd).BC.Frame_Done   := False;
        Driver.Devs(Local_Fd).RT.Recv_Index   := RT_Recv_Index_Type'First;
    end if;
    if (Status = Success) then
        Driver.Driver_Status.CDI_OK_Flag := True;
    else
        Driver.Driver_Status.CDI_OK_Flag := False;
    end if;
    Fd := Local_Fd;
    Result := Status;
exception
    when others =>
        -- Set VxWorks error number
        Result := Failure;
end CDI_Open;

```

1553 Data Bus Driver, CDI_Close

```
-----
--*** (U) SUBPROGRAM NAME: CDI_Close
--***
--*** (U) EFFECTS: 1553 driver close routine.
--***           The Application Software will call this procedure to
--***           delete or remove a file descriptor received from a call
--***           to CDI_OPEN.
--***
--*** (U) EXCEPTIONS PROPAGATED: No propagated exceptions are explicitly
--***           raised in this unit.
--***
--*** (U) USAGE CONSTRAINTS: This routine is not to be called directly by
--***           the AS. It is accessed via the VxWorks IO
--***           System.
--***           Doesn't affect current message frame, frame
--***           is allowed to complete.
--***
--*** (U) UNDESIREED EVENTS: None
--***-----

function CDI_Close (Fd : in CDI_File_Descriptor_Type)
    return CDI_Status_Type is
        Status : CDI_Status_Type := Success; -- Local status
begin
    -- Error check (Invalid Fd or not opened)
    if (Fd < 1) OR (Fd > Max_Num_Devs) OR
        (Driver.Devs(Fd).Opened = False) then
        Status := Failure;
        -- Set VxWorks error number
    else
        Driver.Devs(Fd).Opened      := False; -- Close device
        Driver.Devs(Fd).Mode        := Idle;  -- Change to idle mode
        Driver.Devs(Fd).BC.Active_Stack := Alpha; -- Stack to known condition
    end if;
    if (Status = Success) then
        Driver.Driver_Status.CDI_OK_Flag := True;
    else
        Driver.Driver_Status.CDI_OK_Flag := False;
    end if;
    return Status;
exception
    when others =>
        -- Set VxWorks error number
        return Failure;
end CDI_Close;
```

1553 Data Bus Driver, CDI_Read

```
-----
--*** (U) SUBPROGRAM NAME: CDI_Read
--***
--*** (U) EFFECTS: 1553 driver read routine.
--***           The Application Software will call this procedure to
--***           receive a Message Data Block from the Driver.
--***
--*** (U) EXCEPTIONS PROPAGATED: No propagated exceptions are explicitly
--***           raised in this unit.
--***
--*** (U) USAGE CONSTRAINTS: This routine is not to be called directly by
--***           the AS. It is accessed via the VxWorks IO
--***           System.
--***
--*** (U) UNDESIREED EVENTS: None
--***-----

procedure CDI_Read (Fd                : in  CDI_File_Descriptor_Type;
                   Message_Data_Block : in  CDI_Message_Block_Access_Type;
                   Max_Bytes           : in  Integer;
                   Num_Bytes           : out Integer;
                   Result               : out CDI_Status_Type) is
    Status : CDI_Status_Type := Success; -- Local status
    Done   : Boolean         := False;   -- Loop completed indicator
    Index  : Integer         := 1;      -- Loop counter
begin
    -- Error check (Invalid Fd or not opened)
    if (Fd < 1) OR (Fd > Max_Num_Devs) OR (Max_Bytes < 1) OR
       (Driver.Devs(Fd).Opened = False) then
        Status := Failure;
        -- Set VxWorks error number
    else
        case Driver.Devs(Fd).Mode is
            when Idle =>
                -- Set VxWorks error number (CDI_Read_Invalid_Mode_Error)
                Status := Failure;
            when BC =>
                if (Driver.Devs(Fd).BC.Frame_Done = False) then
                    Status := Failure;
                    -- Set VxWorks error number
                else
                    -- Copy saved data block to 'Message_Data_Block'
                    -- Select previous stack for read (Not active or inactive)
                    -- For all messages in block
                    -- Check size of message against space remaining in 'Max_Bytes'
                    -- If space and stack entries still remain then read msg data
                end if;
            end case;
        end if;
    end if;
end CDI_Read;
```

```

        -- Copy time tag and error information for message
        -- Check for last message in block
        -- Compute 'Num_Bytes'
        null;
    end if;

when RT =>
    -- Copy all data since last read up to Max_Bytes
    Message_Data_Block.CDI_Data_Header :=
        Driver.Devs(Fd).RT.Recv_Data.CDI_Data_Header;
    while (NOT Done) loop
        if ((CDI_Data_Header_Type'Size +
            CDI_Message_Data_Type'Size * Index) > Max_Bytes) OR
            (Index >= Driver.Devs(Fd).RT.Recv_Index) then
            Done := True;
        end if;
        if (NOT Done) then
            Message_Data_Block.CDI_Message_Data(Index) :=
                Driver.Devs(Fd).RT.Recv_Data.CDI_Message_Data(Index);
        end if;
        Index := Index + 1;
    end loop;

    -- Compute current number of messages
    Message_Data_Block.CDI_Data_Header.CDI_Message_Count :=
        Driver.Devs(Fd).RT.Recv_Index - 1;

    -- Reset number of messages
    Driver.Devs(Fd).RT.Recv_Index := RT_Recv_Index_Type'First;

    -- Reset device time tag register

    -- Reset device stack pointer to start of stack

    -- Reset current command/data pointers
    Driver.Devs(Fd).RT.Stack_Ptr := RT_Stack_Index_Type'First;

    -- Compute 'Num_Bytes'

when MON =>
    -- Swap to inactive from active stack (comm and data ptrs)
    -- Copy data from inactive stack to Message_Data_Block
    -- Copy time tag and status info to Message_Data_Block
    -- Set Message_Data_Block header info
    -- Reset device time tag register
    -- Compute 'Num_Bytes'
    null;

when RT_MT =>
    -- RT Portion
    -- Copy all data since last read up to Max_Bytes
    Message_Data_Block.CDI_Data_Header :=
        Driver.Devs(Fd).RT.Recv_Data.CDI_Data_Header;
    while (NOT Done) loop
        if ((CDI_Data_Header_Type'Size +

```

```

        CDI_Message_Data_Type'Size * Index) > Max_Bytes) OR
        (Index >= Driver.Devs(Fd).RT.Recv_Index) then
        Done := True;
    end if;
    if (NOT Done) then
        Message_Data_Block.CDI_Message_Data(Index) :=
            Driver.Devs(Fd).RT.Recv_Data.CDI_Message_Data(Index);
    end if;
    Index := Index + 1;
end loop;

-- Compute current number of messages
Message_Data_Block.CDI_Data_Header.CDI_Message_Count :=
    Driver.Devs(Fd).RT.Recv_Index - 1;

-- Reset number of messages
Driver.Devs(Fd).RT.Recv_Index := RT_Recv_Index_Type'First;

-- Reset device time tag register
-- Reset device stack pointer to start of stack
-- Reset current command/data pointers

Driver.Devs(Fd).RT.Stack_Ptr := RT_Stack_Index_Type'First;

-- MON portion
-- Swap to inactive from active MON stack (comm and data ptrs)
-- Copy data from inactive MON stack to Message_Data_Block
-- Copy time tag and status info to Message_Data_Block
-- Set Message_Data_Block header info
-- Reset device time tag register
-- Compute 'Num_Bytes'

    end case;
end if;

if (Status = Success) then
    Driver.Driver_Status.CDI_OK_Flag := True;
else
    Driver.Driver_Status.CDI_OK_Flag := False;
end if;
Result := Status;
exception
    when others =>
        -- Set VxWorks error number
        Result := Failure;
end CDI_Read;

```


1553 Data Bus Driver, CDI_Write

```
-----
--** (U) SUBPROGRAM NAME: CDI_Write
--**
--** (U) EFFECTS: 1553 driver write routine.
--**           The Application Software will call this procedure to
--**           give a Message Data Block to the Driver.
--**
--** (U) EXCEPTIONS PROPAGATED: No propagated exceptions are explicitly
--**           raised in this unit.
--**
--** (U) USAGE CONSTRAINTS: This routine is not to be called directly by
--**           the AS. It is accessed via the VxWorks IO
--**           System.
--**
--** (U) UNDESIREED EVENTS: None
--**-----
procedure CDI_Write (Fd           : in CDI_File_Descriptor_Type;
                    Message_Data_Block : in CDI_Message_Block_Access_Type;
                    Num_Bytes       : out Integer;
                    Result           : out CDI_Status_Type) is
    Status : CDI_Status_Type := Success; -- Local status
begin
    -- Error check (Invalid Fd or not opened)
    if (Fd < 1) OR (Fd > Max_Num_Devs) OR
        (Driver.Devs(Fd).Opened = False) then
        Status := Failure;
        -- Set VxWorks error number
    else
        case Driver.Devs(Fd).Mode is
            when Idle => -- Idle
                -- Set VxWorks error number (CDI_Write_Invalid_Mode)
                null;
            when BC => -- Bus controller
                -- Save a copy of 'Message_Data_Block'
                -- Clear flags for new block
                Driver.Devs(Fd).BC.Data_Ready := True;
                -- Select the inactive stack (Not active or previous stack)
                -- For all messages
                -- Check message size against remaining space
                -- If space and stack entries still remain then write msg data
                -- If a hook routine desired enable EOM interrupts
                -- Check for last message
                -- Compute 'Num_Bytes'
```

```

when RT => -- Remote terminal
    -- Save write data
    Driver.Devs(Fd).RT.Trans_Data := Message_Data_Block.all;
    -- Set all Tx lookup table pointers to zero
    -- Set all Driver.Devs(Fd).RT.Trans_Count() to zero
    -- For all messages
        -- If a Tx message
            -- if Driver.Devs(Fd).RT.Trans_Count(SA) = 0
                -- Copy data to buffer and set Tx lookup table ptr
                -- Increment Trans_Count
            -- if Driver.Devs(Fd).RT.Trans_Count(SA) = 1
                -- Copy data to 2nd buffer
                -- Increment Trans_Count
            -- if hook routine requested
                -- Set Driver.Devs(Fd).RT.Hook_Req(SA+RT_Buf_Size)
        -- else if a Rx message and a hook routine requested
            -- Set Driver.Devs(Fd).RT.Hook_Req(SA) to true
    -- Set all Driver.Devs(Fd).RT.Trans_Count() at one to zero
    -- Set all Trans_Count() greater than one to one

when MON => -- Monitor
    -- Copy any hook related messages to Hook_Data in Driver
    -- Clear selective monitor lookup table
    -- For all messages in 'Message_Data_Block'
        -- Enable monitoring for this message
    null;

when RT_MT => -- Remote terminal/Monitor
    -- Save write data
    Driver.Devs(Fd).RT.Trans_Data := Message_Data_Block.all;
    -- Clear MON selective monitor lookup table
    -- Set all Tx lookup table pointers to zero
    -- Set all Driver.Devs(Fd).RT.Trans_Count() to zero
    -- For all messages
        -- If an RT Tx message
            -- Copy message into Driver.Devs(Fd).RT.Trans_Data
            -- if Driver.Devs(Fd).RT.Trans_Count(SA) = 0
                -- Copy data to buffer and set Tx lookup table ptr
                -- Increment Trans_Count
            -- if Driver.Devs(Fd).RT.Trans_Count(SA) = 1
                -- Copy data to 2nd buffer
                -- Increment Trans_Count
            -- if hook routine requested
                -- Set Driver.Devs(Fd).RT.Hook_Req(SA+RT_Buf_Size)
        -- else if an RT Rx message and a hook routine requested
            -- Set Driver.Devs(Fd).RT.Hook_Req(SA) to true
        -- else it is a MON message
            -- Copy any hook related messages to Hook_Data in Driver
            -- Enable monitoring for this message

```

```

        -- Set all Driver.Devs(Fd).RT.Trans_Count() at one to zero
        -- Set all Trans_Count() greater than one to one
    end case;
end if;

if (Status = Success) then
    Driver.Driver_Status.CDI_OK_Flag := True;
else
    Driver.Driver_Status.CDI_OK_Flag := False;
end if;
Result := Status;
exception
    when others =>
        -- Set VxWorks error numbers
        Result := Failure;
end CDI_Write;

```

POBIT Utility, RRPT

```

/* *****
--  NAME:          RRPT
--  TITLE:         1553 Receiver RAM Pattern Test Module
--  DESCRIPTION:
--                The purpose of the 1553 Receiver RAM Pattern Test Module
--                (RRPT) is to thoroughly test the entire 1553 Receiver RAM
--                address space (for both chipsets) for address, data,
--                coupling or bit stuck-at faults.
--
--  INPUTS:
--    PARAMETERS:
--      None
--
--    GLOBAL:
--      None
--
--    LOCAL:
--      None
--
--  OUTPUTS:
--    PARAMETERS:
--      failure - PASS or FAIL status of test
--
--    GLOBAL:
--      None
--
--    LOCAL:
--      None
--
--  REVISION HISTORY
--
--  Ver  Date      Prgrm          PCR #      Title
--  ---  ---      ---          ---          ---
--  1.1  3-6-96    J. Daly          Created
--  CM CONTROL
--  -    3-29-96  J.Daly          STN-035   Initial Transfer to PSL
--
--  *****
--  ***-----
--  *** | RESTRICTED RIGHTS LEGEND
--  *** | Contract No. F33657-95-D-2026
--  *** | Contractor Name: Sanders A Lockheed Martin Company
--  *** | Contractor Address: 65 Spitbrook Road, Nashua, NH 03061-0868
--  *** |
--  *** | The Governments rights to use, modify, reproduce, release,
--  *** | perform, display, or disclose this software are restricted by
--  *** | paragraph (b) (3) of the Rights in Noncommercial Computer Software
--  *** | and Noncommercial Computer Software Documentation clause
--  *** | contained in the above identified contract. Any reproduction of
--  *** | computer software or portions thereof marked with this legend
--  *** | must also reproduce the markings. Any person, other than the
--  *** | Government, who has been provided access to such software must
--  *** | promptly notify the above named Contractor.
--  *** |-----
--  ***
--  *** CONFIGURATION CONTROL #: 6387668
--  ***
--  *** FILE NAME:          RRPT.c
--  ***
--  *** ABBREVIATION:       RRPT
--  ***
--  *** ASSUMPTIONS:       None
--  ***
--  *** USAGE CONSTRAINTS:  Executed by IOP only.

```

```

--***
--***----- */
#include "RRPT.h"

/*
 * external function declarations
 */
extern void bzero(); /* write zeros to memory */

int RRPT
(
    void
)
{
    int failure = 0; /* no failures to start out */
    volatile unsigned short int *rcvr_one_start; /* rcvr 1 start address */
    volatile unsigned short int *rcvr_two_start; /* rcvr 2 start address */
    volatile unsigned short int *rcvr_one_end; /* rcvr 1 end address */
    volatile unsigned short int *rcvr_two_end; /* rcvr 2 end address */
    volatile unsigned short int *ptr_one; /* ptr to current address 1 */
    volatile unsigned short int *ptr_two; /* ptr to current address 2 */

    /*
     * set address pointers to receiver SRAM addresses
     */
    rcvr_one_start = (volatile unsigned short int *)RECEIVER_1_SRAM_ADDR;
    rcvr_two_start = (volatile unsigned short int *)RECEIVER_2_SRAM_ADDR;
    rcvr_one_end = (volatile unsigned short int *) (rcvr_one_start +
        RECEIVER_SRAM_SIZE - 1);
    rcvr_two_end = (volatile unsigned short int *) (rcvr_two_start +
        RECEIVER_SRAM_SIZE - 1);

    /*
     * The RRPT module shall test the 1553 receiver RAM for both chipsets
     * according to the following:
     * a) Test the entire 1553 message space external RAM.
     * b) Test that each bit of each RAM location can be set and reset.
     * c) Verify that adjacent bits in a RAM location are not coupled during
     * write activities.
     * d) Verify that a write/read activity of a given RAM location does not
     * affect data in other RAM locations.
     * e) Verify that each RAM location is uniquely addressable.
     */

    /*
     * Use a well-known memory array test algorithm (M-ATS, K-H algorithm)
     * to test 1553 SRAM for address and data stuck-at faults and bit
     * couplings. e.g:
     * if read value != written value then
     * increment failure
     * NOTE: test both areas at the same time
     */

    /*
     * pass 1
     */
    for (ptr_one = rcvr_one_start + 1, ptr_two = rcvr_two_start + 1;
        ptr_one <= rcvr_one_end; ptr_one++, ptr_two++)
    {
        *ptr_one = PATTERN_ONE;
        *ptr_two = PATTERN_ONE;
    }
    *rcvr_one_start = PATTERN_TWO;
    *rcvr_two_start = PATTERN_TWO;

    /*
     * pass 2
     */
    for (ptr_one = rcvr_one_start + 1, ptr_two = rcvr_two_start + 1;
        ptr_one <= rcvr_one_end; ptr_one++, ptr_two++)
    {

```

```

        if ((*ptr_one != PATTERN_ONE) || (*ptr_two != PATTERN_ONE))
        {
            failure++; /* increment failure count */
            break;     /* discontinue testing */
        }
        *ptr_one = PATTERN_TWO;
        *ptr_two = PATTERN_TWO;
    }
    /*
    * pass 3
    */
    if (!failure)
    {
        for (ptr_one = rcvr_one_start, ptr_two = rcvr_two_start;
             ptr_one <= rcvr_one_end - 1; ptr_one++, ptr_two++)
        {
            if ((*ptr_one != PATTERN_TWO) || (*ptr_two != PATTERN_TWO))
            {
                failure++; /* increment failure count */
                break;     /* discontinue testing */
            }
            *ptr_one = PATTERN_ONE;
            *ptr_two = PATTERN_ONE;
        }
    }
    /*
    * verify end is still 0xaaaa
    */
    if (!failure)
    {
        if ((*rcvr_one_end != PATTERN_TWO || (*rcvr_two_end != PATTERN_TWO))
        {
            failure++; /* increment failure count */
            break;     /* discontinue testing */
        }
    }
    /*
    * verify start is still 0x5555
    */
    if (!failure)
    {
        if ((*rcvr_two_start != PATTERN_ONE || (*rcvr_two_start != PATTERN ONE))
        {
            failure++; /* increment failure count */
        }
    }
    bzero(rcvr_one_start, RECEIVER_SRAM_SIZE - 1); /* restore to original state */
    bzero(rcvr_two_start, RECEIVER_SRAM_SIZE - 1); /* restore to original state */
    return failure;
}

```

MIBIT Utility, Run_1553_Test_Group

```

-----
--*** (U) SUBPROGRAM NAME: Run_1553_Test_Group
--***
--*** (U) EFFECTS: Performs Initiated and Maintenance Built-In-Test of the
--***               1553 Test Group.
--***
--*** (U) EXCEPTIONS PROPAGATED: Exceptions are propagated to the calling
--***                           unit.
--***
--*** (U) USAGE CONSTRAINTS: Internal procedure. Executed by IOP only.
--***
--*** (U) UNDESIREED EVENTS: None
--***-----
procedure Run_1553_Test_Group (Fatal_Error : out Boolean;
                             Result       : out MIBIT_Status_Type) is
    Status      : MIBIT_Status_Type := Fail;      -- Test status indicator
    Done        : Boolean            := False;     -- Loop complete boolean
    Fatal       : Boolean            := False;     -- Flag for fatal error
    Test_Count  : Test_Count_1553_Type := Test_Count_1553_Type'First;
    Pass_Count  : Test_Count_1553_Type := Test_Count_1553_Type'First;
begin
    -- Cycle through all bit test modules in 1553 Test Group:
    -- RTAT, MRGT, OLBT, MRXT, MCPT, RAPT
    while (not Done) loop
        -- Check discrete input for abort signal
        -- execute soft reset on abort
        Done := True;

        -- Initialize status so tests can fall through if statement on failure
        Status := Fail;

        -- Do appropriate test
        case Test_Group_1553_Type'Val (Test_Count) is
            when RTAT => -- 1553 RT Address Test
                if (Common_BIT_RTAT = Pass) then
                    Status := Pass;
                end if;
            when MRGT => -- 1553 Register Test
                if (Common_BIT_MRGT = Pass) then
                    Status := Pass;
                end if;
            when OLBT => -- 1553 Offline Loopback Test
                if (Common_BIT_OLBT = Pass) then
                    Status := Pass;
                end if;
            when MRXT => -- 1553 Multiple Buffer Receive Mode Test
                if (Common_BIT_MRXT = Pass) then
                    Status := Pass;
                end if;
            when MCPT => -- 1553 Mode Code Processing Test
                if (Common_BIT_MCPT = Pass) then
                    Status := Pass;
                end if;
            when RAPT => -- RT Address Parity Test
                if (Common_BIT_RAPT = Pass) then
                    Status := Pass;
                end if;
        end case;
        Test_Count := Test_Count + 1; -- Count test just performed
    end loop;
end Run_1553_Test_Group;

```



```
    if (Status = Pass) then
        Pass_Count := Pass_Count + 1;
    end if;

    if (Test_Count = MIBIT_Computer_Count) then
        Done := True;
    end if;
end loop;

if (Pass_Count = MIBIT_1553_Count) then
    Status := Pass;
else
    Status := Fail;
end if;
Fatal_Error := Fatal;
Result      := Status;
end Run_1553_Test_Group;
```

Common_BIT Utility, Common_BIT_OLBT

```
-----
--*** (U) SUBPROGRAM NAME: Common_BIT_OLBT
--***
--*** (U) EFFECTS: 1553 Offline Loopback Test Module
--***           The 1553 Offline Loopback Test Module verifies the basic
--***           operation of the 1553 chipsets for each bus.
--***
--*** (U) EXCEPTIONS PROPAGATED: No propagated exceptions are explicitly
--***           raised in this unit.
--***
--*** (U) USAGE CONSTRAINTS: IOP Only
--***           Works in off-line loopback mode
--***
--*** (U) UNDESIRED EVENTS: None
-----
function Common_BIT_OLBT return Common_Bit_Status_Type is
  Status : Common_BIT_Status_Type := Fail;
begin
  -- For each 1553 chipset and valid pass count
  -- Initialize and write a 1553 message
  -- Alter the BC Control Word of the message to do offline self-test
  -- Send the message
  -- If message OK
  -- Increment pass count
  -- If pass count equals number of chipsets
  Status := Pass;
  -- Else, generate Fault Record for NVM
  return Status;
exception
  when others =>
    -- Generate Fault Record for NVM
    return Fail;
end Common_BIT_OLBT;
```

Common_BIT Utility, Common_BIT_MCPT

```
-----
--*** (U) SUBPROGRAM NAME: Common_BIT_MCPT
--***
--*** (U) EFFECTS: 1553 Mode Code Processing Test Module
--***           The 1553 Mode Code Processing Test Module verifies that
--***           both 1553 chipsets have the ability to send, receive
--***           and process mode codes.
--***
--*** (U) EXCEPTIONS PROPAGATED: No propagated exceptions are explicitly
--***                           raised in this unit.
--***
--*** (U) USAGE CONSTRAINTS: IOP only
--***                           Works in off-line loopback mode
--***
--*** (U) UNDESIREd EVENTS: None
-----
function Common_BIT_MCPT return Common_Bit_Status_Type is
    Status : Common_BIT_Status_Type := Fail;
begin
    -- For each 1553 chipset and valid pass count
    -- Test a valid mode code
    -- If OK
    -- Test an invalid mode code
    -- If OK
    -- Increment pass count
    -- If pass count equals number of chipsets
    Status := Pass;
    -- Else, generate Fault Record for NVM
    return Status;
exception
    when others =>
        -- Generate Fault Record for NVM
        return Fail;
end Common_BIT_MCPT;
```

Common_BIT Utility, Common_BIT_RAPT

```
-----
--** (U) SUBPROGRAM NAME: Common_BIT_RAPT
--**
--** (U) EFFECTS: 1553 RT Address Parity Test Module
--**               The 1553 RT Address Parity Test Module verifies that the
--**               RT register parity is valid for each mission bus.
--**
--** (U) EXCEPTIONS PROPAGATED: No propagated exceptions are explicitly
--**                           raised in this unit.
--**
--** (U) USAGE CONSTRAINTS: IOP only
--**
--** (U) UNDESIREED EVENTS: None
-----
function Common_BIT_RAPT return Common_Bit_Status_Type is
    Status : Common_BIT_Status_Type := Fail;
begin
    -- For each 1553 chipset and valid pass count
    -- If RAPT_Error() for chipset is still set to false then
    --   Count number of bits in RT address set in 1553 config register 5
    --   If number of bits + RT address parity is odd then
    --     Increment pass count
    --   Else
    --     Set RAPT_Error() to True for this chipset
    -- If pass count equals number of chipsets
    Status := Pass;
    -- Else, generate Fault record for NVM
    return Status;
exception
    when others =>
        -- Generate Fault Record for NVM
        -- Set VxWorks Error Number (RAPT Ada Exception)
        return Fail;
end Common_BIT_RAPT;
```

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE December 1996		3. REPORT TYPE AND DATES COVERED Master's Thesis
4. TITLE AND SUBTITLE AN APPROACH TO EVALUATE SOFTWARE EFFECTIVENESS			5. FUNDING NUMBERS	
6. AUTHOR(S) Timothy J. Schalick, Captain, USAF				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology 2750 P Street WPAFB, OH 45433-7126			8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/96D-24	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ AFOTEC/SAS ATTN: Jeff Wiltse 8500 Gibson Blvd SE Kirtland AFB, NM 87117-5558			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution unlimited			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) <p>The Air Force Operational Test and Evaluation Center (AFOTEC) is tasked with the evaluation of operational effectiveness of new systems for the Air Force. Currently, the software analysis team within AFOTEC has no methodology to directly address the effectiveness of the <i>software</i> portion of these new systems.</p> <p>This research develops a working definition for software effectiveness, then outlines an approach to evaluate software effectiveness-- the Software Effectiveness Traceability Approach (SETA). Effectiveness is defined as the degree to which the software requirements are satisfied and is therefore application-independent.</p> <p>With SETA, requirements satisfaction is measured by the "degree of traceability" throughout the software development effort. A degree of traceability is determined for specific pairs of software life-cycle phases, such as the traceability from software requirements to high-level design and low-level design to code. The degrees of traceability are combined for an overall software effectiveness value.</p> <p>It is shown that SETA can be implemented in a simplified database, and basic database operations are described to retrieve traceability information and quantify the software's effectiveness.</p> <p>SETA is demonstrated using actual software development data from a small software component of the avionics subsystem of the C-17, the Air Force's newest transport aircraft.</p>				
14. SUBJECT TERMS Software Effectiveness, Requirements Traceability, Software Development, Software Quality, Software Engineering.			15. NUMBER OF PAGES 204	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to *stay within the lines* to meet *optical scanning requirements*.

Block 1. Agency Use Only (Leave blank).

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C - Contract	PR - Project
G - Grant	TA - Task
PE - Program Element	WU - Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. (If known)

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

DOD - See DoDD 5230.24, "Distribution Statements on Technical Documents."

DOE - See authorities.

NASA - See Handbook NHB 2200.2.

NTIS - Leave blank.

Block 12b. Distribution Code.

DOD - Leave blank.

DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.

NASA - Leave blank.

NTIS - Leave blank.

Block 13. Abstract. Include a brief (*Maximum 200 words*) factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code (*NTIS only*).

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.