NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

DERIVATION STRATEGIES
FOR
EXPERIENCED-BASED TEST ORACLES

by

Jose A. Hernandez, Jr.

December, 1989

Thesis Advisor: Timothy J. Shimeall

Approved for public release; distribution is unlimited.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (*If applicable*) Code 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (*If applicable*)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (*City, State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
DERIVATION STRATEGIES FOR EXPERIENCED-BASED TEST ORACLES (Unclassified)

12. PERSONAL AUTHOR(S)
HERNANDEZ, JR., JOSE A.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Master's Thesis	FROM _____ TO _____	1989 December	101

16. SUPPLEMENTARY NOTATION
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.

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Software Lifecycle; Software Testing; Requirements; Test Oracle; Oracle Derivation; Requirements Abstraction

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

Traditionally, large software systems are tested to demonstrate that the system satisfies the set of functional requirements and specifications from which it was derived. Various methodologies exist for conducting this type of testing. However, when the requirements document, or specification, has become outdated or incomplete to the point that they are irrelevant, then testing must take a different approach in order to verify and validate. There can be many reasons why a large software system gets developed without a clear specification; notwithstadnding testing must proceed even when confronted with a non-existent specification. Testing in such situations is difficult since there is no separation of specified function from implemented function, and thus no objective standard for judging the correctness of test results.

This research proposes a methodology for verification and validation of large software systems when no effective requirements specification exists. To derive an objective correctness standard, the methodology employs requirements information gained from a variety of

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Professor Timothy J. Shimeall	22b. TELEPHONE (*Include Area Code*) 408-646-2509 22c. OFFICE SYMBOL 52SM

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

19. ABSTRACT (continued)

sources: user conferences, developer conferences, new user manuals, inverse transformation of code to specification, a validated "kernel" system, and previous test strategies.

Approved for public release; distribution is unlimited.

Derivation Strategies
for
Experienced-Based Test Oracles

by

Jose A. Hernandez, Jr.
Captain, United States Marine Corps
B.S.C.S., University of Idaho

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1989

Author: Jose A. Hernandez, Jr.
Jose A. Hernandez, Jr.

Approved by: Timothy J. Shimeall
Timothy J. Shimeall, Thesis Advisor

Rachel Griffin
Rachel Griffin, Second Reader

Robert B. McGhee
Robert B. McGhee, Chairman
Department of Computer Science

ABSTRACT

Traditionally, large software systems are tested to demonstrate that the system satisfies the set of functional requirements and specifications from which it was derived. Various methodologies exist for conducting this type of testing. However, when the requirements document, or specification, has become outdated or incomplete to the point that they are irrelevant, then testing must take a different approach in order to verify and validate. There can be many reasons why a large software system gets developed without a clear specification; notwithstanding testing must proceed even when confronted with a non-existent specification. Testing in such situations is difficult since there is no separation of specified function from implemented function, and thus no objective standard for judging the correctness of test results.

This ~~research~~ Thesis proposes a strategy for verification and validation of large software systems when no effective requirements specification exists. To derive an objective correctness standard, the strategy employs requirements information gained from a variety of sources: user conferences, analyst conferences, new user manuals, inverse transformation of code to specification, a validated "kernel" system, and previous test strategies.

Keywords: computer program verification, (KR)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I owe a considerable amount of gratitude to my advisor, Professor Timothy J. Shimeall. His guidance, patience, and encouragement were instrumental in the completion of this research. Along the way, he insured that I stayed within the scope of my thesis, especially when I considered ideas that would have created complications. He taught me a great deal, not only in software testing theory but also in how to communicate ideas. It would have been difficult to complete this research without his advice and expertise.

I also am grateful to my second reader, LCDR Rachel Griffin, USN, for her willingness to evaluate this research for accuracy and readability. I appreciate her meticulous effort.

I thank the people at IRMD, MCLB, Albany, Georgia, for allowing me to study the M3S development environment and to use M3S as a case study in this research. Of particular note, I would like to acknowledge Mr. Nick Retza, Captain Doug Turlep, and Major Richard Miller for supporting my efforts. Despite their constantly busy and hectic schedules, they unselfishly tolerated my numerous inquiries, both in person and by phone. Their responsiveness significantly aided my efforts.

Finally, but by no means any less important, I am deeply indebted to my wife, Jan. There is absolutely no possible way I could have completed this research without her uncompromising, loving support. She provided necessary spousal encouragement and compensated for my slack in husbandly and parental duties during the many hours I spent studying away from home. However, her effort and attitude should come as

no surprise to me because she has always supported my career endeavors throughout our marriage. She is a true blessing. To my children, Audrey and Andrew, I thank you for your smiling, cheery faces and your playfulness that lifted my spirits after long, hard hours of studying.

I. INTRODUCTION

A. SIGNIFICANCE OF SOFTWARE TESTING

Testing is an activity that is applied to the development of products. Whether it be for an automobile, aircraft, stereo, or a dishwasher, some form of testing occurs before a product is released to the consuming public. Normally, the degree of consumer satisfaction with a product will depend on the quality of the product. The quality of the product will reflect how well it performs within the scope of its intended functionality. Testing is a means for achieving quality assurance, and, thus, the degree and techniques of testing a product will play an extremely important role in the success of a product, success both in terms of consumer satisfaction and financial return.

Software products are no different. Testing software products is not a new concept. In fact, ever since the inception of computers during the 1940's, software products developed and executed on computers have been tested in one fashion or another. However, it has not been until the last two decades, perhaps beginning with the first formal conference devoted to software testing - Computer Program Test Methods Symposium, University of North Carolina - in June, 1972 [Ref. 1], that software testing has been given any significant attention in the software development and life cycles. Previously various levels of understanding existed regarding the presence of errors and faults in software. Those errors and faults impacted not only the development environment but also the operational and maintenance environment. Testing was conducted in a somewhat ad-hoc manner with the intent of detecting and

removing obvious errors to provide a functional product. In current testing methods a systematic approach is employed in an effort to detect a wide range of errors and faults in requirements, design, and code.

Since that first conference there have been a number of conferences on software testing that have brought together the practical experience of industry leaders and the research efforts of academia. The result has been a greater emphasis on the importance of software testing with a variety of proposed systematic methodologies for conducting and managing software development testing. The techniques and methods include requirements-based, design-based, logic-based, constraint-based, semantics-based, syntax, structure, function, path, coverage, loop, transaction flow, state transition, mutation, unit, integration, performance, stress, and configuration testing, among others.

In essence, software testing in today's world of complex automation is extremely important if any degree of acceptable quality assurance is to be attained, for the sake of reputation, product image, and competitive standing. This is reflected in Miller's [Ref. 2] contention that the errors that software test methods can uncover are the *5% of product functionality* that make more than *95% of the difference* in users' perceptions of quality. One goal of testing is to demonstrate that a product satisfactorily performs its intended function. The source of this intended function is a requirements specification.

B. SIGNIFICANCE OF A REQUIREMENTS SPECIFICATION

A requirements specification is a document that specifies in a clear, definitive, and rigorous manner what it is a system is supposed to do. It is just as important to note that from the requirements specification explicit inferences can be made as to what the system is not suppose to do. The requirements specification is the result of

requirements analysis, an activity where the acquiring organization and the analyst/developer attempt to properly define a system. The acquiring organization brings to the table functional expertise as well as a perception of additional automation needs. The analyst/developer brings to the table analytical skills and software development expertise in solving real-world problems through automation. The requirements specification evolves during this iterative process between the user and analyst/developer to where the final version is acceptable and understandable from the user's perspective and is usable and implementable from the analyst/developer's perspective. The form of the specification may include English-narrative, structured-English, data flow diagrams with a data dictionary, and/or mathematical-based formalism.

The importance of a requirements specification can not be overstated. It is the foundation on which software development builds. It sets the tone for system development in terms of morale and ultimate success. Without a requirements specification, design will be based on the designer's perceptions and assumptions and not a negotiated statement of need. Furthermore, testing will require extra effort and resources to derive appropriate and valid test cases. So while it is possible to develop a system without a requirements specification, in almost every situation the result will not reflect the acquiring organization's desires.

Care is needed in the composition of a requirements specification because deficiencies will propagate through subsequent phases of software development in what is known as spoilage. Spoilage is defined by DeMarco [Ref. 3] as the effort dedicated to diagnosis and removal of the faults that were introduced during the development

process. It represents about 55% of the total lifetime cost of the average system. Figure 1-2 depicts the cost of spoilage. Spoilage becomes increasingly difficult and

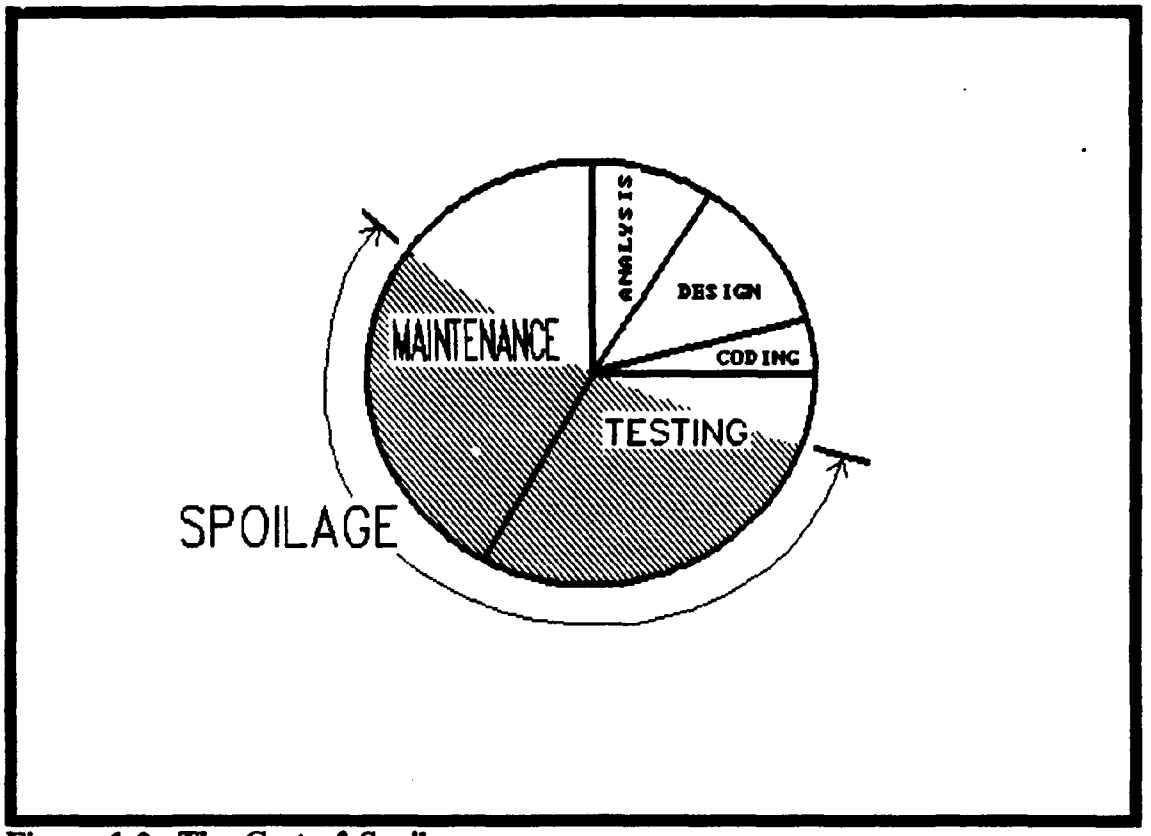

Figure 1-2 The Cost of Spoilage

expensive to correct in later phases of development. Figure 1-3 depicts a summary of experience at IBM, GTE, TRW, and Bell Labs on the relative cost of correcting software errors as a function of the phase in which they are corrected [Ref. 4]. This underscores the importance of the requirements specification to clearly define the problem at the outset of development. A clearly defined and complete specification will contain little, if any, spoilage. The result is less compounded spoilage during the total life cycle of a system.

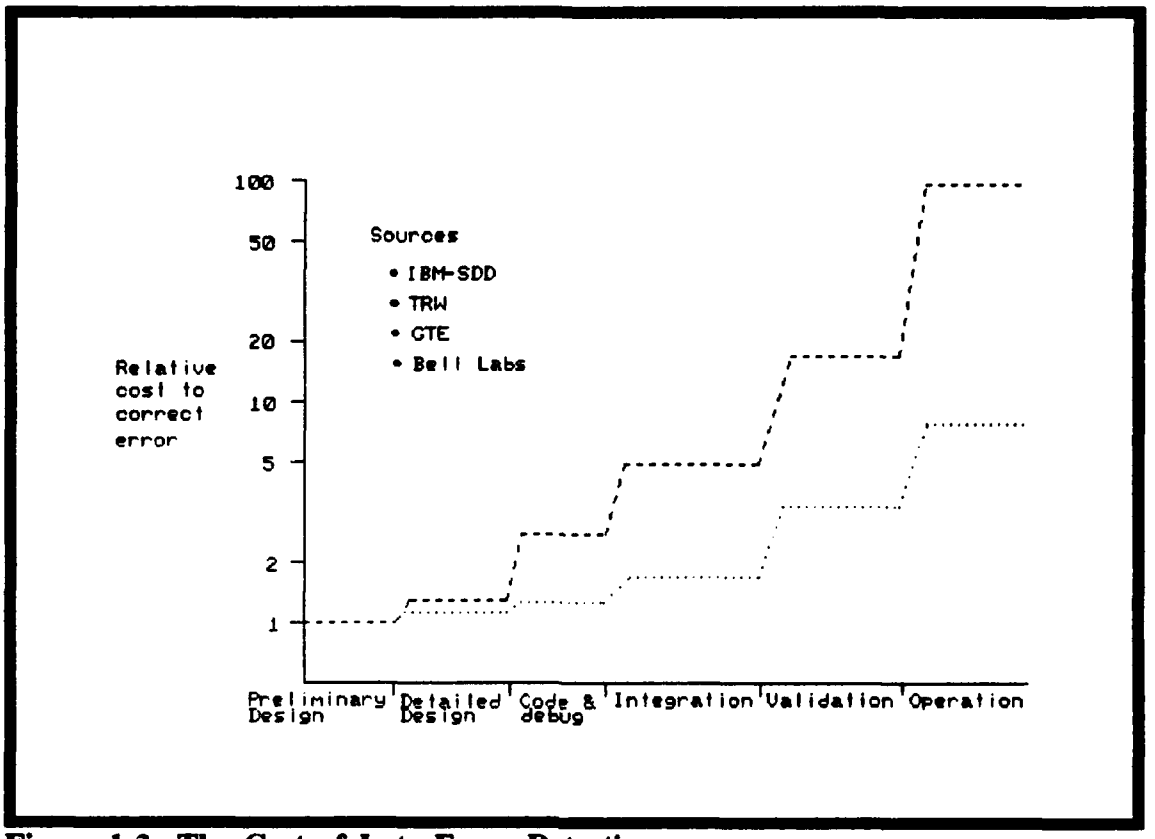

Figure 1-3 The Cost of Late Error Detection

C. PURPOSE OF THESIS

In the context of the previous discussion, large software systems are traditionally tested to demonstrate that the system satisfies its specified requirements. However, when the requirements specification does not exist then other sources of information must be examined to determine the user's desire. There can be many reasons why a large software system gets developed lacking a usable requirements specification. That list of reasons might include the following:

- the requirements specification was originally initiated but not kept up-to-date as requirements were modified and added as system development aged, especially where aging occurred over a period of years

- the requirements specification exists but is effectively useless because it was authored by individuals unknowledgeable or inexperienced in developing a requirements specification that is a rigorous, clear description of what a system is to do

- as a result of political influences, management forced design and coding to begin quicker than it should have, thereby preventing adequate time and effort to be expended in the analysis phase

- a change of methodology during system development failed to adequately convert the original requirements specification to reflect the new methodology

- the users were not adequately consulted because system management felt that enough functional knowledge was present on the development team; thus, the analysis phase is bypassed and the design phase begins

- the intended system is too complex, preventing an adequate requirements analysis to be conducted

The cost and effort of regressing to the analysis phase to capture a pure requirements specification could be prohibitive and in most situations is politically unfeasible. Notwithstanding testing must proceed even in an environment lacking a usable requirements specification. The purpose of this research is to propose a strategy for verification and validation of large software systems when no effective requirements specification exists.

D. SCOPE OF THESIS

1. Need for Revised Approach to Testing

Testing in the absence of a usable requirements specification is difficult since there is no separation of specified function from implemented function, and thus no objective standard for judging the correctness of test results. A need exists, therefore, to derive an objective standard to determine if the software results match the user's expectations. The derived objective standard is called a test oracle. Howden [Ref. 5]

defines a test oracle as an external mechanism that can be used to check test output for correctness. Test oracles come in different forms, e.g., tables, hand-calculated values, simulated results or informal requirements and design descriptions. To derive a test oracle, this research employs requirements information gained from a variety of sources. The resulting test oracle will be a set of acceptability rules based on a collection of requirements that is important to capture not only for testing purposes but also to serve as a baseline requirements specification for subsequent maintenance.

2. Overview of Thesis

Chapter 2 provides a synthesis of software testing theory as well as an overview of the software life cycle. Chapter 3 is a case study of a real-world situation where no complete or effective requirements specification exists. This material is presented to illustrate the dynamics influencing a requirements specification and software development in general. However, this thesis is not attempting to provide an explicit test oracle to the case study. Chapter 4 provides the strategies for deriving a test oracle given the myriad of sources present in the development environment. The strategies presented are general enough to deal with the issues involved in the case study but are not specific to that case study. Chapter 5 summarizes this thesis and indicates directions for further research.

II. SYNTHESIS OF SOFTWARE TESTING THEORY

The primary motivation for this chapter is to provide the reader with the background material on which this thesis is based. This background material is a synthesis of testing methodology and theory as it applies to the software life cycle. Hence, sources for this information are drawn from both academic and commercial environments.

This chapter describes development and testing following the classical software life cycle, with all information about the application present in a functional specification. In other words, this material does not present a solution for the case where no usable specification exists. The aim, therefore, is to contrast what occurs in the preferable specification-based development process to the experience-based process on which this research is based. A strategy for successful and economic testing in a development process that does not include a formal specification is presented in Chapter 4.

A. SOFTWARE LIFE CYCLE

By definition, the software life cycle is the period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. [Ref. 6]

The software life cycle can be used as a framework for producing a product for a user in a manner that is conducive to cost-optimization and efficient resource- and time-utilization. Additionally, the chances of providing a maintainable product are greatly enhanced. Not using a software life cycle framework invites trouble in the form of time and cost overruns, as well as a non-maintainable product.

1. Requirements Phase

Before discussing this phase, a few definitions are necessary. A *requirement* is a condition or capability needed by a user to solve a problem or achieve an objective. It must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. The set of all requirements forms the basis for subsequent development of the system or system component. *Requirements analysis* is the process of studying user needs to arrive at a definition of system or software requirements. [Ref. 6]

The *requirements phase* is the period of time in the software life cycle during which the requirements for a software product, such as the functional and performance capabilities, are defined and documented [Ref 6]. Stated differently, the goal of the requirements phase is to explicitly state both the problem and the constraints upon the solution. Requirements identification is somewhat iterative with the requirement statement being subject to modification during design as the problem is better understood. These modifications must be documented to create a traceable record of the progress and evolution of the final product. [Ref. 7]

It can be argued that this phase is more important than any other. There must be a clear understanding of the software product to be produced. This clear

understanding must be shared between the developer and the customer. Without this clear definition, development will proceed in an ad-hoc manner with an incomplete understanding of the customer's desires and, in the end, the customer may be very dissatisfied with the result and possibly reject the product. Time well-spent in this phase in terms of customer-developer interaction and documentation will be returned many-fold later in the software life cycle. A popular saying certainly applies here - "Pay me now or pay me later."

2. Design Phase

Design is the process of defining the software architecture, components, modules, interfaces, test approach, and data for a software system to satisfy specified requirements. *Design analysis* is the evaluation of a design to determine correctness with respect to stated requirements, conformance to design standards, system efficiency, and other criteria. [Ref. 6]

The *design phase* is the period of time in the software life cycle during which the designs for architecture, software components, interfaces, and data are created, documented, and verified to satisfy requirements [Ref. 6]. The goal of this phase is to design a solution that satisfies the requirements and constraints. Alternative solutions are formulated and analyzed and the best solution is selected and refined. A high-level specification which defines information aggregates, information flows, and logical processing steps is generated and is refined into a detailed specification describing the physical solution (algorithms and data structures). The result is a solution specification that can be implemented in code with little additional refinement.

Project plans (schedules, budgets, deliverables, etc.) are reviewed and revised as appropriate. [Ref. 7]

The result of the design phase is a blueprint from which programmers literally build a system in the form of code. This blueprint shows both the functional description - "what" - and the logical description - "how". It's final form can be in a number of formats; examples of notation used to describe the design include (not exhaustive) flow charts, HIPO (Hierarchy plus Input-Process-Output) charts, pseudo code, structure charts, and data flow diagrams.

3. Implementation Phase

Implementation is the realization of an abstraction in more concrete terms; in particular, in terms of hardware, software, or both. In other words, it is the process of translating a design into code and debugging the code. The *implementation phase* is the period of time in the software life cycle during which a software product is created from design documentation and debugged. [Ref. 6]

Here is where the time investment of the earlier phases pays off. With a well-organized and clearly-stated design document, programming can proceed in an organized and structured manner that is conducive to system testing as well as future maintenance activities. Without a suitable design document, programming may be reduced to haphazard and ad-hoc methods that will introduce a rat's nest of spaghetti code, the outcome of which is a worthless product relative to the customer's needs.

4. Testing Phase

Testing is the process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified

requirements or to identify differences between expected and actual results. The testing phase is the period of time in the software life cycle during which the components of a software product are evaluated and integrated, and the software product is evaluated to determine whether or not requirements have been satisfied. [Ref. 6]

Three types of testing are performed: unit, integration, and system. Typically the programmer is responsible for unit testing. The responsibility for integration and system testing is determined by the project management, depending on project size and criticality. Unit testing checks for typographic, syntactic, and logical errors. Code modules are checked individually by the programmers who wrote them to ensure that each correctly implements its design and satisfies the specified requirements. Integration testing focuses on checking the intermodule communication links and on testing aggregate functions formed by groups of modules. System testing examines the operation of the system as an entity, sometimes in a simulated operating environment. This type of testing ensures that the software requirements have been satisfied both singly and in certain combinations. The final activity of this phase is to ensure readiness for the software installation, including revision of plans as necessary and completion of all other coding, testing, and documentation. The techniques used during testing will be covered more in depth in a later section. [Ref. 7]

Although the Federal Guideline [Ref. 7] states that unit testing is the responsibility of the programmer, in reality the typical programmer does not, for various reasons, perform suitably rigorous testing. Thus, the personnel assigned to conduct integration and system testing should also perform unit testing in some reasonably acceptable manner.

Various references, both academic and commercial, state that the largest part of software cost is caused by the presence of faults (bugs) and the process of detecting and removing those faults. If this is indeed the case, then the idea of emphasizing the requirements and design phases is strengthened. With a clearly stated problem and a subsequently clearly designed solution, the degree of bugs will be more restricted to the skill of the programmers instead of embedded problems in a poorly specified product.

5. Delivery Phase

Delivery is the point in the software life cycle at which a product is released to and/or accepted by its intended user for operational use. [Ref. 6]

During this phase the system is placed into operation. The first task, integrating the system components, may include installing hardware, installing the program(s) on the computer, reformatting/creating the data base(s), and verifying that all components have been included. The next task is to test the system in its complete operating environment. The test data from earlier phases is enhanced and used. The result is a system qualified and accepted for production use. The third task is the start of system operation. If a previous system exists, then strategies for its replacement include immediate total replacement, "phasing-in" of the new system, or parallel operation of both systems. A completely new program could either be phased into operation or could be implemented at once. This task also includes operator and user training. [Ref.7]

6. Operation and Maintenance Phase

This phase is the period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements. [Ref. 6]

Once the need for change has been established, a modified development cycle is begun all over. That is, requirements and design analysis must be conducted, implementation and debugging then occurs followed by testing, and finally delivery and installation takes place. Making a change, no matter how trivial, to a system is no simple matter. Great care must be taken to study the effects of a change and to understand its "ripple" effect, if any. Additionally, any maintenance changes made to a system must be reflected to the previous original state of the system in the way of updating all documents, e.g., requirements document, associated with all phases to include user manuals.

Again, the investment spent in the requirements and design phases to insure quality will reap benefits during the maintenance phase. Otherwise, poor requirements and design quality will lead to excessive maintenance costs.

B. THE ROLE OF THE SPECIFICATION

1. Definition and Scope

The IEEE Standard [Ref. 6] defines a specification as a document that prescribes, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or system component. Thus, a specification can be associated with each phase in the software life cycle. For instance, there are

requirements specifications, design specifications, test specifications, functional specifications, and so on. This research is primarily concerned with the requirements specification and its relationship to testing. Therefore, unless otherwise noted, future references to specifications should be considered to be a reference to requirements specification. A requirements specification is a more formal and precise way of stating the requirements of a system as opposed to the original user-written requirements. Formal here is used in the context of explicitness and not mathematical-based.

2. Process of Producing Specifications

Specifications are evolved through two forms of activities. The first form is the establishment of an entirely new specification. This occurs as a result of interaction between some or all of the following parties: customer, user, analyst, and developer. In fact, these people could all be contained in the same organization, be totally independent of each other, or a combination thereof. Regardless of whether two, three, or all four of the parties are present, a synthesis of ideas takes place predicated on knowledge from the perspective of each party. Knowledge from the customers and users is their needs and expectations along with the initial problem statement that was derived from those needs and expectations. The user provides terminology, semantics, and procedures peculiar to the application area. Knowledge from the developer includes experience in solving problems with software and formalizing the needs of the customer. Evolution of a new requirements specification is almost always an iterative process where the developer, who analyzes requirements, not only will rely on his own expertise, but will consult the customer as well. The goal is to produce a specification that clearly elaborates the behavior of a product.

Ideally, the stopping point for this iterative process would coincide with the end of the requirements phase. Realistically, however, the requirements specification continues to evolve because of undiscovered issues that arise during the design and implementation phases that must be resolved, and the resolution must then be reflected in the specification.

The second form of evolution is a specification derived from a previously existing specification. This type of evolution would most likely occur in the maintenance phase. The process entails applying change proposals to an existing specification. Then analysis and verification is conducted on the changed specification so as to insure it accurately reflects the intent on which the change was based. The result is a new specification sufficiently changed during the derivation process such that a distinction can be drawn between the old and the new specification.

3. Specification Quality

Since a specification reflects what a system or product is supposed to do, then it seems natural to argue that the quality of the system is directly tied to the quality of the specification. The IEEE definition [Ref. 6] of quality is defined by the totality of features and characteristics of a product or service that bears on its ability to satisfy given needs. Hence, the product (service) mentioned in the definition is the requirements specification in the present context. Quality can have many attributes. One list of attributes, taken from [Ref. 8], is:

- unambiguous
- complete
- verifiable
- consistent

- modifiable
- traceable
- usable during the operation and maintenance phase.

Refer to Table 2-1 for a description of each of these attributes. Figure 2-1, taken from Boehm [Ref. 9], shows an alternate way of classifying the characteristics of a quality specification. Here the list consists of four basic properties:

- completeness
- consistency
- feasibility
- testability

As can be seen, each property has sub-properties that further define it.

Now that the quality concept has been discussed, several reasons can be given as to why specification quality is necessary or desired. Firstly, the effect of a quality specification will cascade through all subsequent phases of the software life cycle. Design should reflect the requirements, implementation should reflect the design, and thus, less effort should be required in testing which eventually will verify that the software product reflects specified functionality. Specification quality also carries over into the maintenance phase where it will be easier to determine the feasibility and impact of making a change. Additionally, a quality specification will have the effect of causing less corrective maintenance than would have occurred with a poor specification.

Secondly, a quality specification will not be a cause of, or a contributor to, budget overruns or schedule delays. Many times when budget overruns or schedule delays occur, the incomplete, ambiguous specification is a main culprit because of the time and effort that must be spent in backtracking to correct early-phase mistakes. However, in the absence of a poor specification, the blame for budget overruns and

Table 2-1 Specification Attributes

unambiguous - every requirement stated therein has only one interpretation

complete - (1) inclusion of all significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces
(2) definition of the responses by the software to valid and invalid input values
(3) no use of the phrase *to be determined* (TBD)

verifiable - for every requirement stated in the specification there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement.

consistent - no set of individual requirements described in it conflict.

modifiable - structure and style are such that any necessary changes to the requirements can be made easily, completely, and consistently.

traceable - the origin of each of its requirements is clear and it facilitates the referencing of each requirement in future development or enhancement documentation.

useable during the operation and maintenance phase - specification is modifiable (as previously defined) and documented with any special provisions such as criticality and reason for origin of a function.

schedule delays must rest elsewhere, such as managerial decisions, personnel problems, and a change of environment, to name a few.

Thirdly, a quality specification should result in a quality product. Thus, the customer will be satisfied and more inclined to continue to do business with the developer. At the very least, the reputation of the developer will be maintained and probably even enhanced. On the other hand, a poor specification will probably result in a mediocre product that fails to meet the needs of the customer. The result: a

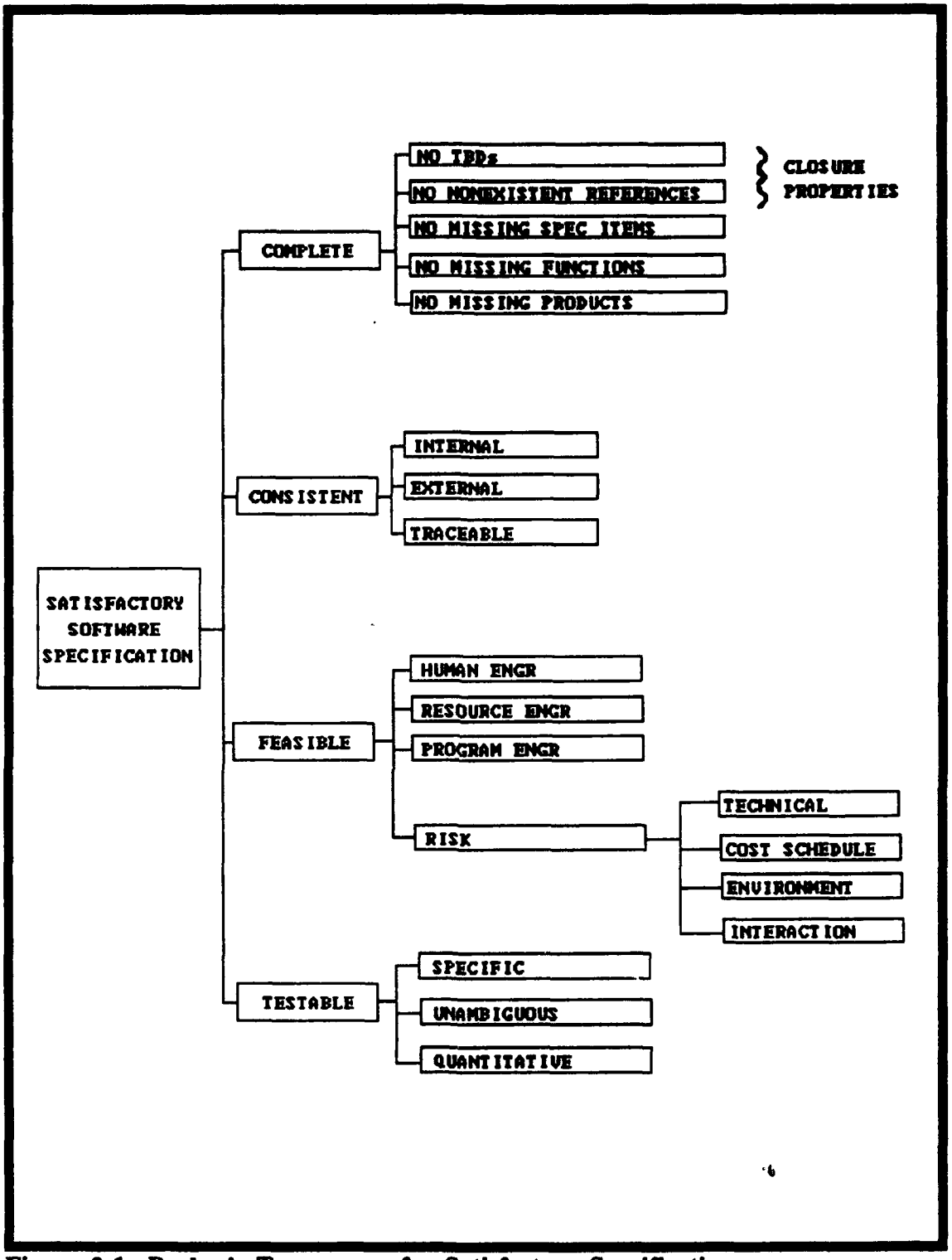

Figure 2-1 Boehm's Taxonomy of a Satisfactory Specification

dissatisfied customer, a developer with a damaged reputation, and probably a developer receiving less business in the future.

C. VERIFICATION AND TESTING TECHNIQUES

1. Verification

Verification is the process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase [Ref. 6]. For example, the process of verifying a specification is that of determining if the requirements (in the specification) state the user's wants and/or needs in a clear and precise manner. As another example, the process of verifying the design document is that of determining if the design reflects the specification.

Verification of the requirements specification can be achieved through the use of several techniques such as reviews, checklists, scenarios, prototypes, and automation. A review is where someone other than the author reads the specification. This is the iterative process mentioned previously where the specification is subjected to different points of view (e.g., users, testers, maintainers, developers) to iron out misconceptions and inconsistencies. A checklist is a specialized list composed from experience that aids in evaluating a specification for some of the characteristics listed in Table 2-1 and Figure 2-1. However, a checklist should not be considered absolute but rather a flexible guide for evaluating a specification. A scenario describes how the system will work once it is in operation. It is normally a man-machine interaction and is most useful in clarifying human engineering aspects in the specification. A prototype is a throw-away representation of the system that is most useful for

demonstrating the feasibility of a product and refining the developers understanding of the user's needs. It is just a shell of what the final product is supposed to be and is developed quickly to help resolve accuracy, real-time, and feasibility issues. Automation requires the use of a specification language and its associated language processor. A specification written in some specification language can be automatically analyzed for many of the properties listed in Table 2-1 and Figure 2-1. The automated method, if available, saves significant time over the manual methods because the specification language will permit fewer ambiguities and inconsistencies by way of its structure and syntax, while allowing extensive cross-referencing that otherwise is not conducive to manual methods. The drawback to automated verification tools are that they are currently not well-developed. [Ref. 9]

2. Testing

Testing is ideally based on the system specification, because it is the specification that expresses exactly what the system is supposed to do. This section concentrates on the process and methodology of testing. The reader should note that the concern here is with testing, which is the process of detecting the presence of faults, and not debugging, which is the process of locating and removing the faults [Ref. 10].

The goal of testing is not to demonstrate absolute program correctness, because that would be impractical as well as, in most cases, impossible. Rather the goal of testing should be to suitably demonstrate a reliable and quality product. There should be a "warm and fuzzy" feeling versus heartburn. What is "suitable" is relative

and should be addressed within the context of the product being developed, such as considering costs and long and short term goals. [Ref. 10]

3. Analytic Techniques of Testing

a. Static Analysis

Static analysis is the process of evaluating a program without executing the program [Ref. 6]. Some examples are mechanical analysis, manual analysis, and reviews.

The best example of mechanical analysis is a source language processor that provides statement identification, lexical scan, parsing, symbol tables, and label tables. This process is static because it performs the operations necessary to convert the source code to object code before any execution of that code can be done. [Ref. 10]

Two forms of manual analysis are desk checking and walkthroughs. Desk checking should be a predictable and finite activity. Some techniques of desk checking are code reading, examining a variable cross-reference list and a label cross-reference list, reconciling a cross-reference list for subroutines, macro, and functions, and examining equates and constants. A walkthrough is a review process in which a programmer leads a group of people, such as his peers and supervisors, through an explanation of the code he has written. As a result, a walkthrough forces the programmer to be more meticulous about his work. A walkthrough (properly done) causes the programmer to impose quality control on himself by making sure his code is "right", by ensuring his code is understandable, and by explaining the logic to himself and his audience. Manual analysis, therefore, is a form of testing because it

uncovers deficiencies that would otherwise be discovered by dynamic analysis. By discovering errors during static analysis, the scope of fault-finding during dynamic analysis is narrowed, resulting in a more efficient process. [Ref. 10]

A review is an evaluation of technical matter and performance by an individual or (more commonly) a group of individuals working together. The objective is to obtain reliable information as to status and/or work quality. Reviews can be conducted on specifications, design, coding, and test plans, among others. Reviews provide many benefits such as providing a framework for reliably evaluating progress, bringing individual capabilities to light, discovering batches and classes of errors at once, giving early feedback on potential problems and training and educating the participants. [Ref. 1]

b. Dynamic Analysis

Dynamic analysis is the process of evaluating a program based on execution of the program [Ref. 6]. It evaluates the code's behavior during execution by addressing structural, functional, and computational aspects of the code. Howden [Ref. 5] classifies dynamic analysis into three strategies: requirements-, design-, and code-based testing. The difference between each strategy is the perspective on which a program is examined. However, employing all three strategies ensures a set of comprehensive tests.

In requirements-based testing, the primary emphasis is on testing the software product for functionality as defined in the requirements specification. Functionality is the result delivered by a function after processing input data. In other words, regardless how a function is implemented, given a certain input the function will

produce a certain output. This description of functionality is known as a *black box* and is depicted in Figure 2-2. To conduct *black box* testing, the requirements specification is analyzed to determine input data classes, functions, and output data classes. Testing input and output data classes should include values not only on boundaries but also out-of-bounds values that are expected to cause errors. In essence, the goal of requirements-based testing is to determine if observed behavior matches expected behavior. [Ref. 5]

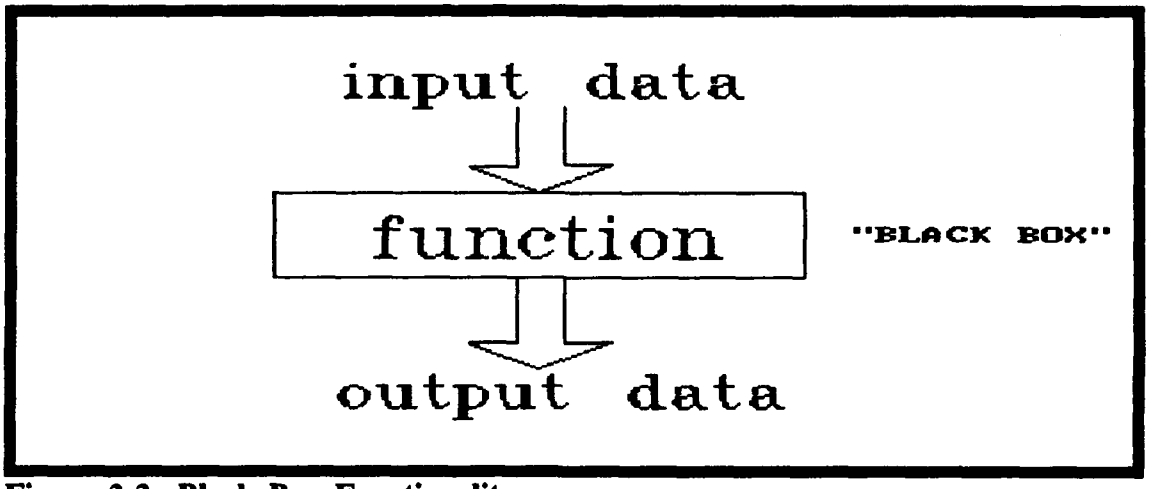

Figure 2-2 Black Box Functionality

In design-based testing, the focus is on testing design elements such as data structures, paths, interfaces, states, and intermediate values. Here the test effort concentrates on verifying that the software product performs as specified by the design specification, i.e., the "how" inside the box in Figure 2-2 is tested. This is known as *white box* testing.

In code-based testing the idea is to test specific computational aspects of the code, which is also known as coverage testing. Coverage testing methods include statement testing, branch testing, and path testing. Statement testing is the most restrictive; it requires executing each statement at least once. Branch testing executes each branch in the code at least once. The problem with branch testing is that it may not uncover faults inherent in certain combinations of branches. This problem is resolved by path testing, which corresponds to testing each logical path in a program at least once. [Refs. 1, 5]

c. Formal analysis

Formal analysis is the use of rigorous mathematical techniques to analyze the algorithms of a solution. The algorithms may be analyzed for numerical properties, efficiency, and/or correctness [Ref. 7]. Predicate calculus, specification languages, and symbolic execution are a few examples of formal analysis. Formal analysis is not a different type of analysis than static and dynamic analysis; it is rather another method that can be used to conduct either static or dynamic analysis. Formal analysis has not yet gained widespread use, primarily because of the high learning curve associated with it which results in a shortage of knowledgeable personnel, not to mention the generation of long and involved proofs.

4. Levels of Testing

The three most common levels of testing are unit, integration, and system testing. This subsection provides an overview of these three levels. Various techniques for testing at each level include one or more of those listed in the discussion on static

and dynamic analysis. However, these techniques represent only a portion of testing techniques used in practice.

a. Unit Testing

In the literature, module and unit are essentially synonymous. The IEEE Standard [Ref. 6] defines a module as a program component that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine. It is a logically separable part of a program that is usually the work of one programmer. The unit is developed based on some description (ideally a specification) and it is that description on which unit testing is based. Unit testing is the testing of a module for typographic, syntactic, and logical errors, for correct implementation of its design, and for satisfaction of its requirements [Ref. 7].

Unit testing is usually performed by the programmer, most commonly in an informal manner, although some organizations take a more formalized approach to unit testing. With an informal approach, the programmer decides when he has sufficiently tested his module. Depending on the programmer's level of experience and the trust management has in him, this method of testing can be adequate for unit testing. However, a more formalized approach where a test designer and the programmer conduct the testing together would be more rigorous and reliable. [Ref. 1]

One result of successful unit testing is the reduction of effort required in integration testing. If testing and debugging at the unit level is successful such that the unit performs according to its specification, then the effort at the integration testing level can be concentrated on testing the connections between modules. [Ref. 11]

b. Integration Testing

Integration testing is an orderly progression of testing in which software elements, hardware elements, or both, are combined and tested until the entire system has been integrated [Ref. 6]. "Elements" in the previous definition refers to modules or a collection of already integrated modules.

Before conducting integration testing, an integration test plan should be developed addressing what style of integration will take place. Style comes in two forms: direction and quantity. Direction is a top-down or bottom-up approach, or a combination of the two. Top-down testing would require stubs but no drivers while bottom-up would require drivers but no stubs. Integration in either case is only in the direction of one level at a time. In the top-down case, units are integrated with the modules they call where in the bottom-up case, units are integrated with the modules that call them.

Quantity is either incremental or phased. The incremental approach calls for adding only one module at a time. The advantage is being able to attribute any new errors after adding a module to that module, although there is no guarantee the new module is the culprit. The disadvantage is that, especially with a large system, the number of integration steps will be much more numerous than in a phased approach. A phased approach calls for creating an element (or skeleton) from a group of related modules and then the element gets added to the system for subsequent testing. The advantage is fewer integration steps. The disadvantage is increased difficulty in locating the cause of an error that was created by adding the group of modules. [Ref. 11]

c. *System Testing*

System testing is the process of testing an integrated hardware and software system to verify that the system meets its specified requirements [Ref. 6]. System testing begins after successful completion of integration testing.

There are three types of system testing: alpha, beta, and acceptance. Alpha testing is conducted by the developers and is considered "in-house." Beta testing occurs outside of the developing organization by a group of users who are given some sort of incentive by the developer in exchange for their thorough practical test and honest appraisal of the product. This group of users agree not to distribute the product outside of their own environments. Acceptance testing occurs after the developer has completed alpha and beta testing and is confident of his product. It is conducted by a customer or user to determine if the system meets their expectations before deciding to accept (buy) the product. [Ref. 11]

The goal of system testing is to validate the final product against its specification. This validation is accomplished by the following types of testing:

- stress
- load and performance
- background
- configuration
- recovery
- security

Stress testing attempts to break the system by stressing all of its resources by putting peak loads on it and exercising extreme time constraints. Load and performance testing verifies that performance objectives are satisfied. Background testing is testing under a real load of simultaneous active transactions. The aim is to force a conjunction of a test path with a normal (background) path to uncover faults

not anticipated in the test plan. Configuration ensures that all functions work under all logical/physical device assignment combinations. Recovery testing evaluates the system's ability to recover from hardware and/or software malfunctions without losing data or control. Security testing confirms that the system's security mechanism is not likely to be breached by illicit users. [Ref. 10]

5. Regression Testing

Regression testing is selective retesting to detect faults introduced during modification of a system or system component, to verify that modifications have not caused unintended adverse effects, or to verify that a modified system or system component still meets its specified requirements. [Ref. 6]

Regression testing is not a level of testing. It is a testing concern that applies equally to all levels of testing when an error has been corrected. The main question that must be addressed with regression testing is how much of the previous test scenario has to be re-executed. Was the bug trivial enough that a reasonable assurance exists that, as a result of correcting the bug, there will be no side effects on the rest of the system? Are the effects of a modification so uncertain that it would be in the best interest of system quality that the entire test suite be run again? These are tough questions, especially when one must take into account time and resources that are required by additional tests. In any event, old test results must be maintained and then used for comparison for any degree of regression testing.

6. Test Design

Designing the test plan ideally should begin as soon as the requirements phase is complete and a clear specification has been produced. (As was previously

mentioned, the test plan is ideally based on the specification.) In fact, the test design experiences its own development cycle that mirrors or parallels the development cycle. The test plan must be specified, designed, and then implemented. If this test cycle is not done concurrently with the development cycle, then it is left to be done at the end of the development cycle, the result of which is a delay in software delivery. The test plan should be prepared and ready to execute by the time programmers complete individual modules.

D. CONNECTION BETWEEN SPECIFICATION AND TESTING

To conduct testing of anything, you must know exactly what is to be tested. Thus, there can be no testing without an understanding of intentions [Ref. 10]. The connection between testing and a specification is that a test attempts to show that an element contains faults, i.e., the element does not satisfy the specification [Ref. 12]. To develop a test, certain characteristics of a specification must be present. It must be understandable, complete, consistent, feasible, and, of course, testable.

The specification is understandable (unambiguous) if the test designers are able to design a structure of tests to demonstrate whether an element satisfies the specification. If the test designers can not understand the specification, the probability exists that the specification needs to be restated. [Ref. 13]

A specification is complete to the extent that all of its parts are present and each part is fully developed. To be complete there must be no TBDs ("to be determined"), no nonexistent references, and no missing specification items such as verification provisions and interface specifications. A specification is consistent to the extent that its provisions do not conflict with each other or with governing specification and

objectives. A specification is feasible if its life-cycle benefits exceed its life-cycle costs, which implies that feasibility should be analyzed before committing to detailed development. A specification is testable if an economically feasible technique can be developed to determine if an element satisfies its specification. Generally speaking, specifications that are precise, unambiguous, and quantitative are conducive to testing. [Ref. 9]

E. TEST TEAM COMPOSITION AND ITS ROLE

1. Goal of Software Test Designer

The goal of the software test designer is to execute a program with the intent of finding errors [Ref. 12]. Bill Hetzel [Ref. 1] believes that this definition is too restrictive and defines it as the aim of evaluating an attribute or capability of a program or system and determining that it meets its required results.

2. Personnel and Criteria

The more diverse the makeup of the test team, up to a point, the more likely the test plan will be rigorous and successful. The team should consist of at least two types of personnel: one (the tester) who is independent of the development process and one who is knowledgeable of the system design. The tester that has not been involved in the system design will less likely be prejudiced and is more likely to contrive extreme cases. Likewise, the designer would not be able to generate the same rigorous test cases because of presumption he will make as a result of being unduly influenced by the design. However, the designer will be able to offer efficiency to the process by eliminating needless test cases and explaining more thoroughly intended functions of the system. Another type personnel to have on the test team, if possible,

is a member from the user community for whom the system is intended. He can add his expertise derived from day-to-day experiences. The aim of test team composition should be a balance between unbiasness (ignorance) and knowledge. [Ref. 14]

The criteria for test team personnel should be suspicious, uncompromising, hostile, and compulsively obsessed with utterly destroying the programmer's software [Ref. 14]. However, these characteristics are meant only from the standpoint of achieving the tester's goal, that is, detecting the presence of errors. It does not mean that dissension should exist between the tester and programmer. On the contrary, a friendly rivalry should develop between the two whose mutual goal is quality software.

F. SUMMARY

As mentioned in the introduction to this chapter, the material just presented is a representative (not complete) picture of the testing process within a specification-based software development environment. One easily-drawn conclusion is that the only way to conduct testing of a product and eventually validate it is to have a solid specification that was derived during the requirements phase. However, considering a large software system development environment where a usable requirements specification does not exist, the cost of regressing back to the requirements phase to construct a pure specification may be prohibitive. This is especially true when millions of dollars have already been expended and expending additional millions is not only impractical or inefficient, but politically undesirable as well. However, the alternative of attempting to release a product not adequately tested or highly unreliable from the user's perspective is not wise either. The consequences are enormous problems and headaches

in the maintenance phase, and could very well end in the demise of the product. Such a scenario is not favorable to reputations and careers.

Thus, the aim of this research will be to find a suitable strategy to test a non-specified system with a reasonable amount of quality assurance and confidence that the product performs as would be expected. With that in mind, the scope of this thesis is then narrowed to the following questions.

Question 1: What benefit can be derived from a user's and analyst's conference?

Question 2: Can useful functionality descriptions be derived from a predecessor system and old user manuals?

Question 3: Can the user's manual for the new system be used as an informal specification?

Question 4: Are current techniques of design recovery and inverse transformation of code to specification applicable?

Question 5: Even though a usable requirements specification does not exist, can some requirements be salvaged from any existing fragmentary descriptions?

Question 6: Can testing on an informally derived specification provide the assurance that the product is valid and reliable?

III. CASE STUDY - MARINE CORPS STANDARD SUPPLY SYSTEM (M3S)

A. INTRODUCTION

This chapter examines a software development project where a requirements specification does not exist. Although ideally a requirements specification should exist for any software development project, in fact, it is not uncommon for a project to lack a requirements specification, for many valid reasons. Thus, this case study is used merely for illustrative purposes and is not intended as an audit or critique of the M3S project.

B. BACKGROUND

The United State Marine Corps is an integral part of the Department of the Navy, and is at all times subject to the laws and regulations established for the Department of the Navy. Within the department there are two services, the Navy and the Marine Corps. Each is a separate service; although individuals and forces of one may be assigned service with and become a part of specified units of the other. The Marine Corps' primary missions are to:

- provide Fleet Marine Forces combined arms and supporting air for the seizure and/or defense of advanced naval bases in land operation essential to naval campaigns;
- provide detachments for service on armed vessels of the Navy and security detachments for the protection of naval stations and bases;
- develop, in coordination with other services, the tactics, techniques, and equipment for landing forces in amphibious operations; and similarly, to develop doctrine, procedures, and equipment of interest to the Marine Corps for airborne operations, which are not provided by the Army;

- be prepared, in accordance with joint mobilization plans, for wartime expansion;
- perform other missions as the President of the United States may direct.

[Ref. 15]

1. Structure of the Marine Corps

a. Fleet Marine Force (FMF)

The FMF is the combat element of the Marine Corps containing the deployable mobile air/ground forces which consist of combat, combat support, and combat service support units.

b. Supporting Establishment

The supporting establishment is a generic term that includes all posts, camps, and stations. The supporting establishment collectively constitutes the "non-FMF" elements of the active forces. To permit rapid deployment of the combat elements, the supporting establishment and the FMF are distinct and separate command elements with autonomous operational, administrative, and funding channels.

c. Marine Corps Reserve

The Marine Corps Reserve are the inactive forces that permit rapid expansion in time of emergency. The organized Marine Corps Reserve is structured into a mirror image of the active FMF unit and is designated as the Fourth Division/Air Wing.

2. Mission of the Marine Corps Supply System

To support its missions, the Marine Corps has been authorized by the Secretary of the Navy to develop a separate and distinct supply system. The mission

of the Marine Corps Supply System is to provide and manage those items necessary for the maintenance and operation of the Fleet Marine Forces, supporting establishment, and the Marine Corps Reserve. The system is controlled by the Commandant of the Marine Corps and is designed for effective operation in both peace and war, with the capability of rapid transition from one to the other, thus making the Marine Corps essentially self-sustaining in logistics operations. The supply system is dedicated to the single purpose of providing the necessary support to Marines in combat and is structured to be responsive to the needs of the operating and supporting forces, no matter where they are located. The supply system is characterized by centralized management, decentralized distribution, and maximum use of automated data systems. [Ref. 15]

3. Purpose of M3S

The purpose of M3S is to provide a single, standard supply system incorporating consumer, intermediate, and wholesale functions. The system will consolidate those functions currently performed by the Supported Activity Supply System, Marine Corps Unified Material Management System, Direct Support Stock Control, and Organic Property Control Accounting. The system will provide common files that will contain those basic data elements that are necessary for effective inventory control and accounting capabilities for all Marine Corps activities. The replacement of aging supply systems is as essential as the replacement and updating of ADP equipment. [Ref. 16]

4. M3S History

Figure 3-1 provides a summarized snap-shot of various events that have taken place with respect to M3S. The significance of Figure 3-1 is that it portrays the dynamic environment of M3S. The effect of this dynamic environment has been schedule delays and budget overruns. For example, the M3S development team originally had designed an Executor Module for M3S. This module required assembly language coding. However, assembly language coding was prohibited by Marine Corps policy, causing the M3S development team to request a waiver of this policy from Marine Corps Headquarters. When the waiver was denied, the M3S development team replaced the Executor Module design with a table-driven architecture, requiring major modification to virtually all specification documents. As another example, in 1982 the development methodology changed. The original methodology was something similar to the classical lifecycle approach (as described in Chapter 2) driven with HIPO diagrams. (HIPO stands for Hierarchy plus Input-Process-Output.) The new methodology was a structured analysis and design approach espoused by Edward Yourdon [Ref. 17]. Transitioning from one methodology to another impacted M3S across the board. It meant retraining personnel, and reviewing and converting all paperwork to reflect the new methodology. Making such a transition requires time and a great deal of effort.

Finally, Figure 3-1 does not reflect other dynamics that have affected M3S. One dynamic occurring at various times during M3S development was the temporary diversion of resources and personnel committed to M3S to effect USMC requirements and policy changes in other data processing areas. Another dynamic is the turnover

DATE	ACTIVITY
1975	USMC conceives M3S.
Mar, 1977	USMC forms system development team to begin writing the ADS Development Plan for M3S.
Mar, 1979	USMC approves ADS Development Plan. M3S project officially begins when USMC issues charter for M3s development team.
1980	M3S development team selects INQUIRE as DBMS for M3S.
1981	M3S development team cancels INQUIRE as DBMS for M3S.
1982	USMC selects ADABAS as standard DBMS for the Marine Corps, thus, M3S development team incorporates ADABAS as DBMS for M3S.
1982	M3S development team replaces Executor Module design with a table-driven architecture.
Feb, 1982	M3S development team reorganizes and restructures development environment, where necessary, in order to comply with DOD Directive 5920.1, Life Cycle Management of Automated Information Systems. Previously, the Marine Corps Automated Data System (ADS) Manual provided guidance for system development.
1983	New ADS documentation standards are drafted based on M3S system development methodology.
1985	USMC adopts new version of ADS development standards.

Figure 3-1 M3S Chronology Summary [Ref. 16, 18]

of personnel. For example, military personnel transfer every two to three years. With many of these individuals being in key positions on the development team, it is difficult to maintain a consistent effort. One last example of another dynamic is the

constantly changing requirements directed by the Department of Defense for commonality in supply-related procedures in the Armed Forces. [Ref. 16, 18]

The point of this discussion is to illustrate the ever-changing environment surrounding M3S. This environment impacts all aspects of M3S including the existence of a requirements specification for which this research is concerned.

C. EXISTING SYSTEMS

Marine Corps supply activities operate at three different levels - wholesale, intermediate, and consumer. Wholesale activities are conducted at Inventory Control Points (ICPs) by designated Integrated Materiel Managers whose responsibilities are to procure, stock, issue, and distribute their items to all branches of the Armed Services and foreign military sales. The ICP distributes items through warehouse facilities at logistic installations. Intermediate activities are responsible for obtaining, stocking, issuing, and distributing supplies to consumer units. The intermediate supply manager obtains supplies from the wholesale level or the local economy, and controls storage and distribution of these supplies through warehouse and storage facilities known as issue points. Consumer activities occur at the lowest organization at which supply items are tracked and accounted. Thus, the consumer level is the end user. The consumer unit obtains supplies from an intermediate or wholesale organization and, where necessary, the local economy. Figure 3-2 depicts the USMC supply organization. [Ref. 19]

Currently there are multiple systems that are used for recordkeeping in all three levels of supply through a combination of automated and manual means. M3S will consolidate four of these systems which are Supported Activity Supply

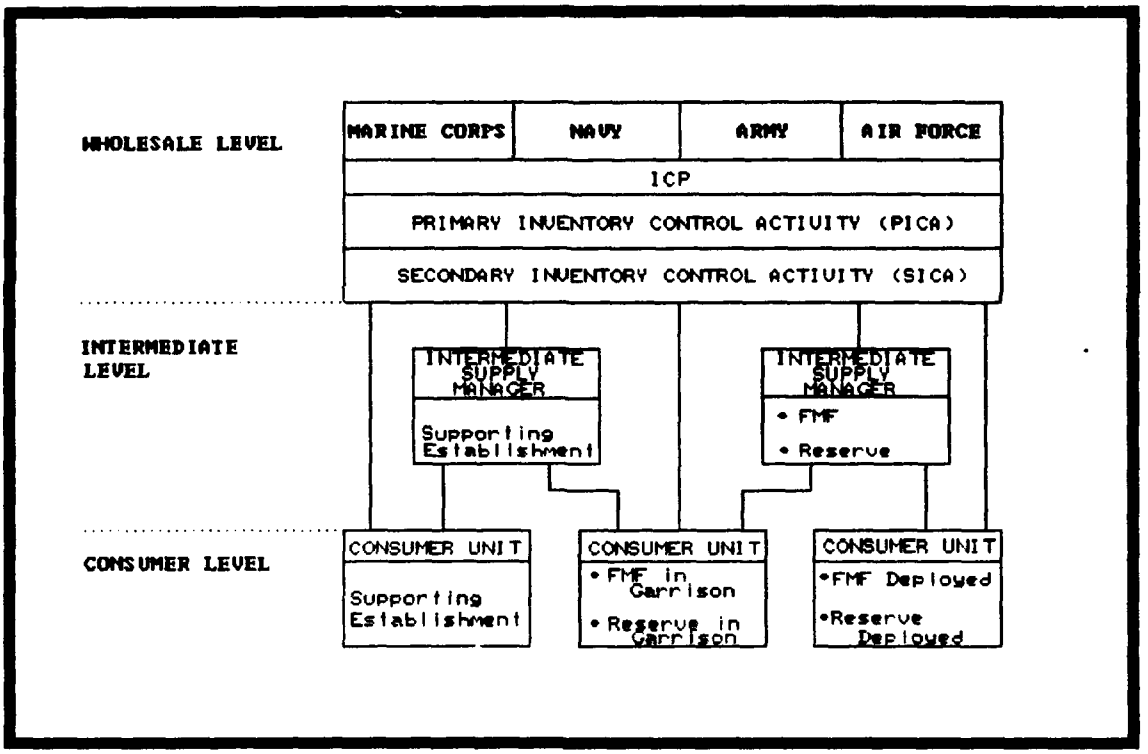

Figure 3-2 USMC Supply Organization [Ref. 19]

System (SASSY), Marine Corps Unified Material Management System (MUMMS), Direct Support Stock Control (DSSC), and Organic Property Control Accounting.

1. Supported Activity Supply System (SASSY)

SASSY is the automated supply management system specifically developed to support the FMF. SASSY functions as a centralized recordkeeper, stock manager, forecaster, and as a central data bank or information point for the using units without negating command responsibility. SASSY attempts to remove supply accounting and recordkeeping functions from the using unit and, in return, provides management reports as an aid to the unit commander, who is responsible for the materiel readiness of his command. SASSY provides computer-produced documentation that reflects the receiving, issuing, and accounting for materiel. Automation reduces the load of

mathematical and clerical functions at the using unit and, thus, reduces manual reporting to daily transaction reporting between the using unit and the SASSY Management Unit. This transaction reporting is the source for the automatic updating of the central files and the initiation of multiple applications; i.e., stock replenishment, increases and decreases to on-hand allowance items, and computer-generated management reports. [Ref. 15]

SASSY utilizes computers to centralize and automate the accounting and recordkeeping functions of the using units. SASSY has the following features:

- reduces the amount of manual reporting, handling, and processing of documents and records to a minimum within the using units
- provides flexibility to operate in combat equally as well as in peacetime, absorb policy changes without disruption, and permit latitude at the using unit level to adjust to various situations
- takes advantage of automated data equipment
- provides for automatic replenishment of operating stock and allowances to the using unit
- provides for management reports as a byproduct of normal operation

[Ref. 15]

2. Marine Corps Unified Material Management System (MUMMS)

MUMMS is an integrated supply management system which satisfies internal and external requirements by taking advantage of modern management techniques and automatic data processing equipment. MUMMS deals with activities primarily at the wholesale level. The MUMMS concept converges the elements of requirements determination, inventory control, financial management, and warehouse management into a single integrated system. The objectives of MUMMS are more effective

management, improved response time to requisitions, and accurate and timely response to DOD reporting requirements. The system provides for a standardization of management policies, adequate organization structure, timely and responsive control devices, and definitive assignment of responsibilities. This, in turn, produces effective supply support to consumer units, increased inventory control through item accounting, timely accumulation and production of inventory data for more efficient financial management, and the maximum effective use of data processing and communication equipment. [Ref. 15]

The administrative tasks and functions required in the operations of MUMMS are organized into fifteen subsystems. These fifteen subsystems operate as one integrated system from the Inventory Control Point (ICP) through a large-scale computer program. The subsystems are interrelated and data in each subsystem are accessible by the other subsystems. The subsystems fall into three general areas. Two areas, supply/financial and technical, pertain to the functions at the ICP. The third area pertains to Remote Storage Activities (RSA) functions. There are eight subsystems in the supply/financial area that deal with inventory control, accounting, procurement, budget, and supply management reports. There are five subsystems in the technical area that deal with provisioning, technical data, war reserve, data control, and applications. The remaining two subsystems in the RSA area are Mechanization of Warehousing and Shipping Procedure (MOWASP) and Direct Support Stock Control. [Ref. 15]

3. Direct Support Stock Control (DSSC)

DSSC activities occur at the intermediate level and are assigned a base support mission. This responsibility includes support of tenant FMF units for selected housekeeping and administrative items. Material stocked in the DSSC activities is generally limited to low-cost, fast-moving consumable items, but also includes stocked items such as lumber, gravel, petroleum, ammunition, and subsistence supplies. Utilizing a retail concept, DSSC positions materiel at issue points close to on-base customers. Assets at these issue points are generally available for personal selection by authorized customers or on demand without formal requisitioning. The Marine Corps Stock Fund finances all materiel held in DSSC activities. [Ref. 15]

4. Organic Property Control Accounting

Each Marine Air Group, battalion, separate squadron, separate company, and separate battery has a property account and is administered as a supply element. Procurement, control, and disposition of materiel are accomplished at the unit supply level. Materiel required by subordinate units is reflected on property records and custody records prepared and maintained by the supply element. In order that unit commanders may exercise command responsibility in supply operations, it is essential that item control, based on established allowance tables and/or usage data, be rigidly applied. Specific allowances of items are established in individual Tables of Equipment (T/E) for all Fleet Marine Force air and ground units. The quantities contained in the T/E are mandatory allowances that the unit is to have on hand or on order. [Ref. 15]

D. DEFICIENCIES IN THE EXISTING SYSTEM

1. Incompatibility and Duplication of Effort

The multiple, automated supply systems mentioned in the previous section manage various inventories. In many cases, the various systems require the same information about the same item of supply. Data redundancy occurs because file-oriented processing requires information to be present in each system. Maintaining this information across system boundaries with any degree of integrity is very difficult. Additionally, the independent nature of the existing systems leads to duplication of effort and poor utilization of resources, both in personnel and hardware. Since the various systems are virtually interdependent, they require separate maintenance. Thus, the maintenance effort strains personnel and financial resources, especially since these systems are at or past their expected service life. [Ref. 16]

2. Technology

The current systems were created over twenty years ago using now-obsolete hardware and software technologies. As a result of advancing technology in the data processing arena, these systems have become obsolete. Maintenance on these systems is limited by the original design and implementation constraints, so maintenance activities could not have incorporated technological advances. In the cases where maintenance could not implement new functionality, that functionality was resigned to manual processes, increasing the load on clerical personnel. At the same time managerial techniques and demands have advanced to the point where the current system can not satisfy managerial information requirements, providing incomplete data, providing the data ineffectively, or failing to provide it in a timely manner. Thus the

current system must be replaced with a new system to meet and keep pace with the demands of the user community. [Ref. 16]

E. TARGET SYSTEM

The final M3S product will be a single supply system that provides standard inventory management programs and integrates the wholesale, intermediate, and consumer functions of supply. As a result, MUMMS, SASSY, DSSC, and organic property systems as they currently exist will have been replaced. The new system will be more responsive, conform to military standards, be flexible to change, and provide uniformity.

M3S is designed around two main features - a centralized data base and a Data Base Management System (DBMS). The centralized data base and DBMS will help eliminate or reduce the problems mentioned in the previous section.

A typical DBMS will provide the following advantages:

- data independence
- data shareability
- reduction of data redundancy
- integrity
- security
- access flexibility
- administration and control

A brief discussion on each of these advantages is contained in Table 3-1 [Ref. 20].

Table 3-1 Advantages of DBMS Technology

data independence - independence or insulation of application programs or users from a wide variety of changes in the specific logical organization, physical organization, and storage considerations of the computerized data base.

data shareability - permitting existing applications and even new applications to access a computerized data base without having to create new data base files.

reduction of data redundancy - the elimination of redundant data as it occurs in non-DBMS, application-specific systems. However, data redundancy is permitted in a DBMS in some cases for technical and performance reasons but this redundancy is *controlled*.

integrity - the coordination of data accessing by different applications, propagation of update of values to other copies and dependent values, and the preservation of a high degree of consistency and correctness of data.

security - the ability to assign, control, and remove the rights of access of any users to any data items or defined subset of the data base. Protects against unauthorized intrusion, whether it be accidental or malicious.

access flexibility - the ability to access any part of the data base on the basis of any access key(s) and logical qualification via a high-level non-procedural query language for browsing through the data base or via IO statements in conventional procedural programming languages.

administration and control - the centralized authority resting with a **data base administrator (DBA) and data base manager (DBM)**. The DBA and DBM are responsible for overall data base design, data definitions, and the procedures for users to access the data base.

The advantages listed in Table 3-1 are reflected in the objectives of M3S. Those objectives are:

- 40% reduction in hard-copy output
- real-time ad-hoc inquiries
- 20% reduction in personnel training requirements
- 20% reduction in inventory cycle processing time
- elimination/control of data redundancy
- integration/interfacing with other USMC systems
- reduce the impact of DOD-directed changes

M3S will execute on IBM-compatible mainframe computers at seven regional centers. The centers are located at the three Information Resources Management Directorates (IRMDs) in Quantico, VA, Albany, GA, and Kansas City, MO, and at the four Regional Automated Service Centers (RASCs) in Camp Lejeune, NC, Camp Pendleton, CA, Camp Smith, HI, and Camp Kinser, Okinawa, Japan. Additionally, mobile mainframe facilities called Deployable Force Automated Service Centers (DFASCs) will support deployed FMF units. The hardware facilities located with the DFASC will provide the deployed units with the computer capability to run the same applications available at the RASC. Thus, the deployed unit will be able to continue processing of supply requirements. [Ref. 19]

F. M3S REQUIREMENTS SPECIFICATION

To simply state that the M3S requirements do not exist without further explanation would be extremely unfair, not only to the M3S system but also to the hard-working, energetic personnel currently in the M3S development environment who are doing the best with what they have. In this case the lack of a requirements specification resulted from the overall nature of software development in the military as well as the evolutionary track of M3S. The lack of a requirements specification in

M3S can not be traced to any one event or decision made by an individual or group of individuals.

If various personnel in the M3S development arena were asked the question - "Does a requirements specification exist?" - the answer would depend on who was asked. The functional user would say yes. Without a full understanding of what is meant by a requirements specification, the functional user would proceed to refer to a slew of Department of Defense publications and Marine Corps publications. Indeed, these publications contain essentially all the information necessary to explain behavior, functionality, and procedures concerning any number of aspects regarding supply functions. This information could be classified as requirements, but it falls far short of being requirements with attributes such as those listed in Table 2-1. Additionally, the information/requirements in these publications have been defined and modified at various times over the past three decades and, thus, were written as a requirement with regard to information and not with regard to software engineering. Finally, the information/requirements in those publications is bundled with additional information that does not directly relate to the everyday processing of supply matters, requiring, at best, extra effort at filtering the necessary from the unnecessary.

If the same question were asked of the software developer, he would respond negatively because he expects to have a document (or set of documents) that specifies the user's requirements. This document would be the result of a requirements analysis and verification process, such as mentioned in Chapter 2, and would be called a requirements specification. Of course, a requirements specification can be represented in various forms using various tools and techniques. The English-narrative

representation of a requirements specification is only one type. In the M3S environment, a structured analysis and design methodology is being employed. That type of methodology represents a requirements specification in the form of a functional requirements definition that consists of a current physical model, a current logical model, and a new logical model.

The new logical model is the key element in the functional requirements definition. (Current literature questions the value of producing a current physical model and current logical model [Ref. 17].) The new logical model defines the essential functions which the new system must perform, as well as the essential stored data to support those functions. It consists of data flow diagrams (DFDs), miniature specifications (minispecs), a data dictionary, and entity-relationship diagrams (ERDs). The DFD is a network/top-down representation that portrays a system in terms of its component pieces and their interfaces. The DFD symbology accounts for data flows, processes, data stores, and terminators. The minispecs are written for each bottom-level process in the DFD. They describe what happens for each process in terms of transforming inputs into outputs according to the user's policy for that process. The data dictionary provides definitions for data flows and data stores. The ERD is a graphical representation of all data stores showing their relationships. It is balanced against the contents in the DFDs, the data dictionary, and the minispecs. [Ref. 21]

Regardless of how a requirements specification is represented, the fact remains that M3S does not have a usable requirements specification as defined in the software engineering literature, structured analysis or otherwise. One major reason that M3S does not have a usable requirements specification is that the requirements analysis

phase was essentially absorbed by the design phase. Although some requirements analysis was conducted, the effort was not extensive enough, producing only fragmentary descriptions. From the point at which the problem statement and economic analysis for M3S was given, the design phase began using knowledge in the heads of expert functional users involved in M3S development, who knew what existing systems did and therefore knew what M3S needed to do. Design was based on interaction between expert functional users and the developers as well as a number of conferences between developers and functional users outside of M3S development. The result is that, informally, any requirements analysis and verification that took place occurred during design activities. Furthermore, design has taken place over greatly varying time periods (years) that allowed new and varying interpretations of requirements to creep into the project. Thus, even though the design reflects some requirements analysis and verification, any requirements analysis and verification that has taken place has not been captured in a formal document, a document that plays an important role during the parallel test plan development and during the subsequent maintenance phase after a system is installed.

Another major reason that M3S does not have a usable requirements specification is due to the change of methodology that occurred well into the development cycle. Design had begun when M3S adopted the structured analysis and design approach. Since then management has decided not to produce a functional requirements definition as called for by their methodology.

In summary, in one sense requirements do exist for M3S, but they exist in the heads of users, in informational-type publications, and in fragmentary descriptions.

Nonetheless, it is entirely possible that these requirements may still be useful in testing. However, in a software engineering sense a requirements specification does not exist insofar as completeness and sufficiency is concerned.

G. CURRENT TESTING APPROACH

M3S has hired an independent contractor to conduct the integration and system testing. The independent contractor has not been involved with M3S development as it relates to design and coding, thus enhancing the probability of an independent and unbiased test plan. In fact, the testing contract stipulates that testing activities will be conducted separate from the development environment. The scope of the independent contractor's test responsibilities is system testing, integration testing, and functional end-user acceptance testing. The semantics of system testing and integration testing used in M3S is slightly different than that described in Chapter 2 but the effect is the same. M3S has 21 different applications that are being developed by independent contractors as well as in-house USMC personnel. These applications are considered systems in M3S terminology but can be viewed as units in Chapter 2 terminology. Therefore, each developer delivers a system (unit) that has undergone unit, integration, and system testing at the developer level. Then M3S conducts its own system (unit) test on each application followed by integration testing of all the systems (units). Finally, the M3S system is tested as a whole, which is a culmination of integration testing. This is equivalent to system testing mentioned in Chapter 2.

The objectives of M3S testing, as stated by the testing contract, are to:

- minimize system life cycle costs;
- reduce risk of system failure;

- ensure that programs and documentation perform functions included in specifications (general and detailed design specs);
- ensure that programs and documentation perform functions required by functional end users.

The inherent value of these objectives should be obvious. However, in the context of this research, there is one objective that stands out - "ensure that programs and documentation perform functions required by functional end users." Without a requirements specification detailing the desires and needs of the user, this objective is more easily stated than accomplished. Furthermore, the current test plan does not address in a clear, definitive manner the discrepancy of testing user requirements when those requirements do not formally exist.

In the three levels of M3S testing, the independent contractor has various responsibilities. These responsibilities are depicted in Figure 3-3. System testing calls for verifying an implementation (code) against design specifications. Integration testing calls for verifying system data interfaces and measuring system performance using a document called the Interface Definition and Control specification. In system and integration testing, the test plan requires user involvement in reviewing test plans, reviewing test data, evaluating user manuals, and participating in validating test results. Functional end-user acceptance testing verifies system behavior and functionality against user requirements and is performed by the user community with assistance from the independent contractor.

The independent contractor employs a functional or "black box" approach to system testing. When a developer delivers an implemented application, the independent contractor assumes the developer has conducted unit and integration testing within that application. (This should not be confused with the testing levels in Figure 3-3.) Each

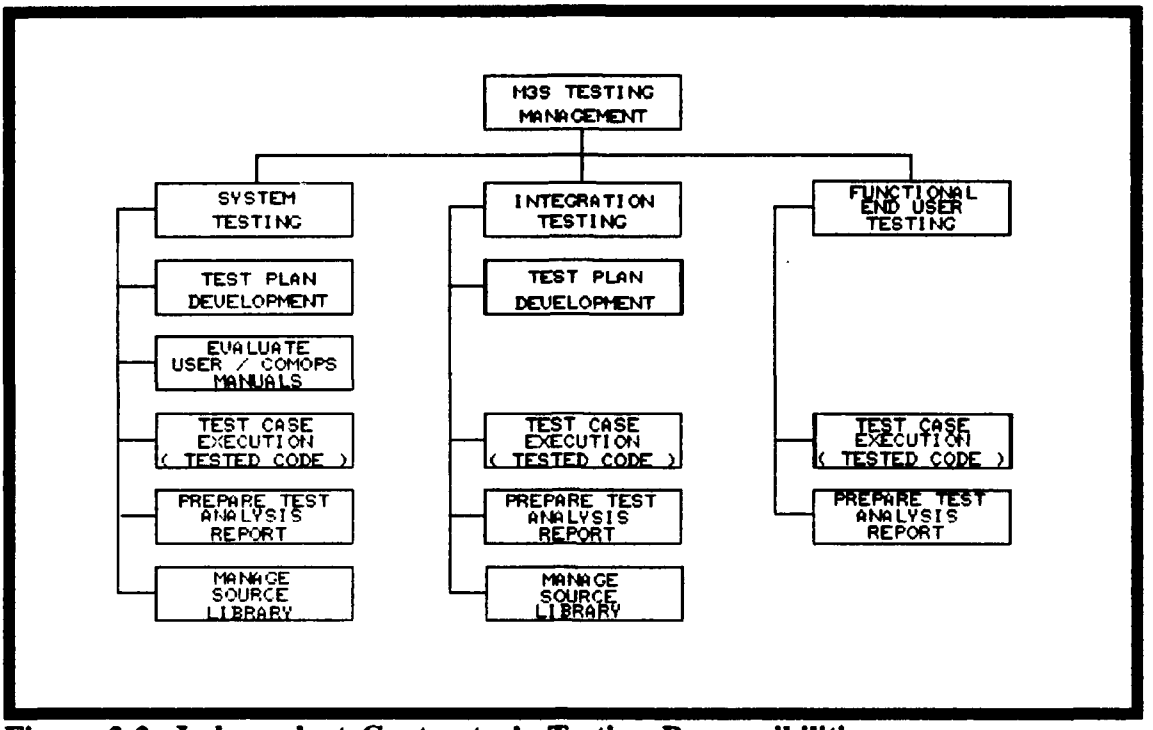

Figure 3-3 Independent Contractor's Testing Responsibilities

developer is free to employ in-house techniques of testing and quality assurance, which include methods commonly used in industry to include those mentioned in Chapter 2. Thus, the independent contractor tests each delivered system for functionality, i.e., whether it produces the expected output when given a specific input.

Additionally, the testing contract requires the independent contractor to create, maintain and update engineered and system test beds for all the M3S applications. An engineered test bed contains data, transactions, and instructions required to perform engineered tests [Ref. 22]. Basically, an engineered test is the range of testing activities that occur during unit and integration testing within a M3S application. Likewise, system test beds mirror engineered test beds except its contents are related to system level testing activities.

H. SUMMARY

Despite the lack of a usable requirements specification, M3S still must be validated. The M3S development personnel undoubtedly will explore and employ techniques and methods, from their perspective, until they feel comfortable with the validation process and result. This would occur in any development environment for a large software system. However, due to the lack of a usable requirements specification, the M3S personnel need techniques to derive an objective correctness standard. The techniques might include, among others, interviewing users, examining user manuals, and examining past experiences.

What form this derivation may take is the subject of the next chapter. The goal is to provide a framework or strategy for validating a system without a requirements specification, dealing specifically with the production of a test oracle. This framework not only captures the requirements needed for testing but provides a baseline for future needs, primarily maintenance.

IV. TEST ORACLE DERIVATION STRATEGY

A. INTRODUCTION

In an environment where no effective requirements specification exists, some document or mechanism is needed to guide testing and validate the results. In such an environment, one simple solution would be to regress to the analysis phase to capture a pure and complete requirements specification. However, in a large software system development effort, the project management may lack the resources (funds, personnel, time) and the political will to undertake such an approach.[1]

Realistically, the testers will use a number of approaches to build test cases and determine the expected results. The list of methods include consulting the user again for requirements, reviewing user manuals for functionality, and examining existing systems for output that should be provided by the new system as well. The creation, maintenance, and augmentation of test beds that reflect previously validated results is also another employed technique. One problem is that these efforts may be highly informal and uncoordinated. A second problem is that the testers may fail to document the byproducts of these efforts (e.g., requirements) for immediate and future use.

This chapter proposes a strategy for deriving a test oracle in an organized manner. Three benefits result. First, the tester will have a mechanism to determine if observed behavior matches expected behavior. Second, tracing test failure to code failure will be easier. Third, this mechanism may serve as a baseline source during

---

[1]It very well may be that the solution proposed in this chapter is no more palatable. Nonetheless, in the suggested scenario some tough decisions will have to be made.

maintenance when subsequent failures must be corrected, when modifications are necessitated, and when increased functionality is requested.

B. FRAMEWORK SYMBOLOGY

The material in this chapter covers many ideas toward deriving a test oracle. In order to organize, connect, and model those ideas in an understandable manner, the framework uses the symbology of structured analysis.[2] The symbology is used strictly as a vehicle for discussing ideas from a textual standpoint and does not imply the automation of those ideas. In fact, Yourdon [Ref. 17] specifically notes that structured analysis symbology is not only ideal for modeling information-processing systems, but also is very useful as a tool for modeling various types of planning.

The discussion in this chapter will start with the context provided by the testing environment and proceed to focus on the specific issues and concepts related to test oracle derivation. The discussion will proceed in a layered fashion, with the higher level abstractions expanded to appropriately illustrate the ideas and methods being discussed.

C. TESTING ENVIRONMENT

The testing environment is depicted in Figure 4-1. The illustration does not portray all factors and issues present in a testing environment. What is shown are the elements necessary to provide context and relevancy to subsequent discussion.

---

[2]For the reader unfamiliar with this symbology, numerous references exist discussing structured analysis and its symbology, e.g., Yourdon [Ref. 17]

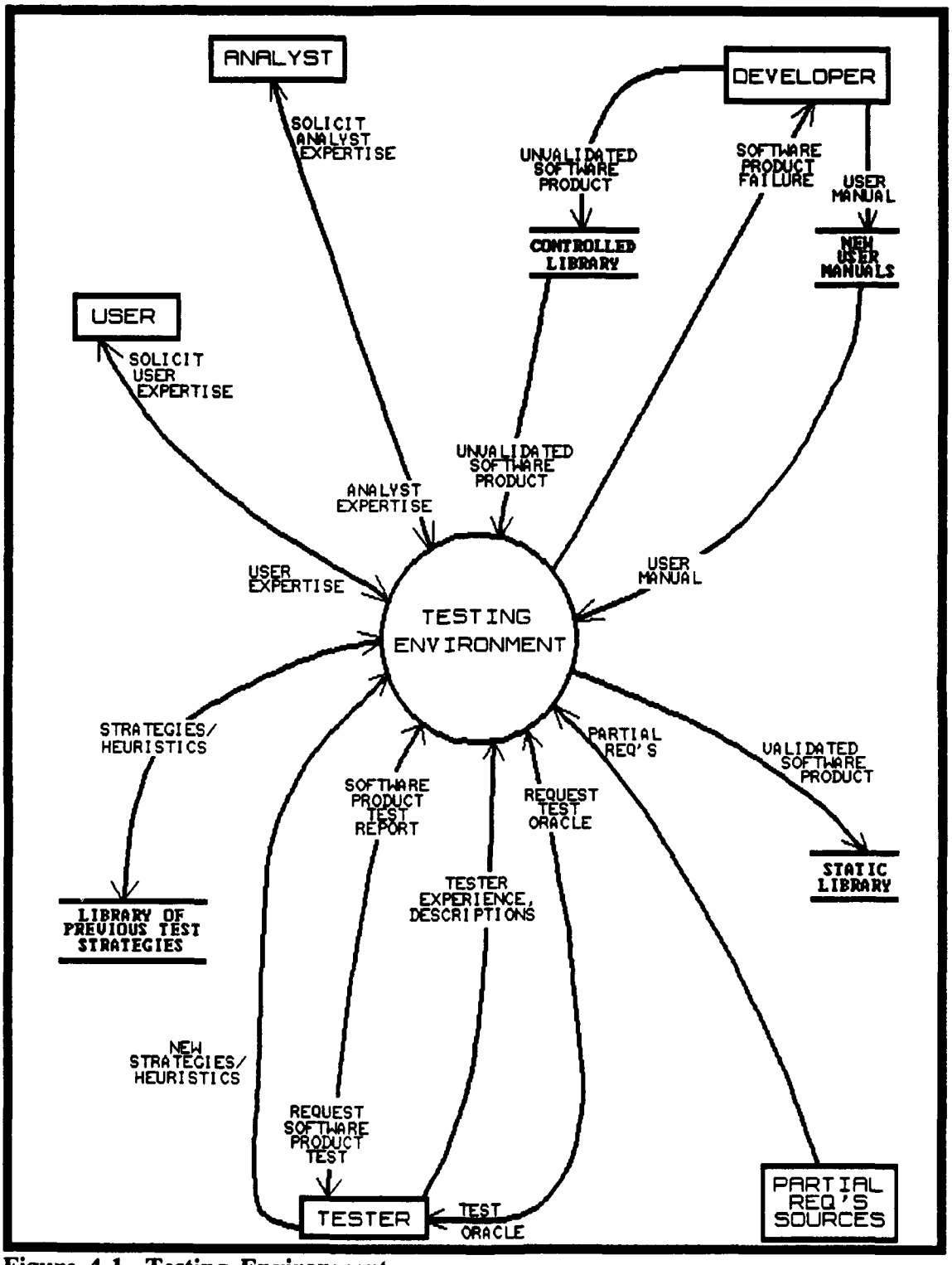

Figure 4-1 Testing Environment

When attempting to derive a test oracle, there are a number of sources available that can be utilized for information gain. In the ideal environment, the majority of the test oracle can be produced by analyzing the requirements specification, especially when the requirements specification possesses the quality attributes discussed in Chapter 2. However, when no effective requirements specification exists, requirements information must be gleaned from other sources. These sources can be people and/or documents and include the user, analyst, tester, developer, source code, user manuals, and fragmentary requirement descriptions. These sources provide products, knowledge, scenarios, and expected results. Various combinations of interaction among these sources occur within a development environment. However, any interactions pertaining to test oracle derivation occur within the testing environment where they can be coordinated and controlled. Interactions unrelated to test oracle derivation are not considered here.

The testing environment entails three key areas of discussion - test oracle derivation, system validation, and the physical test oracle. The test oracle serves as a common repository of information accessible to the activities of deriving the test oracle and validating the system. This lower level expansion is illustrated in Figure 4-2.

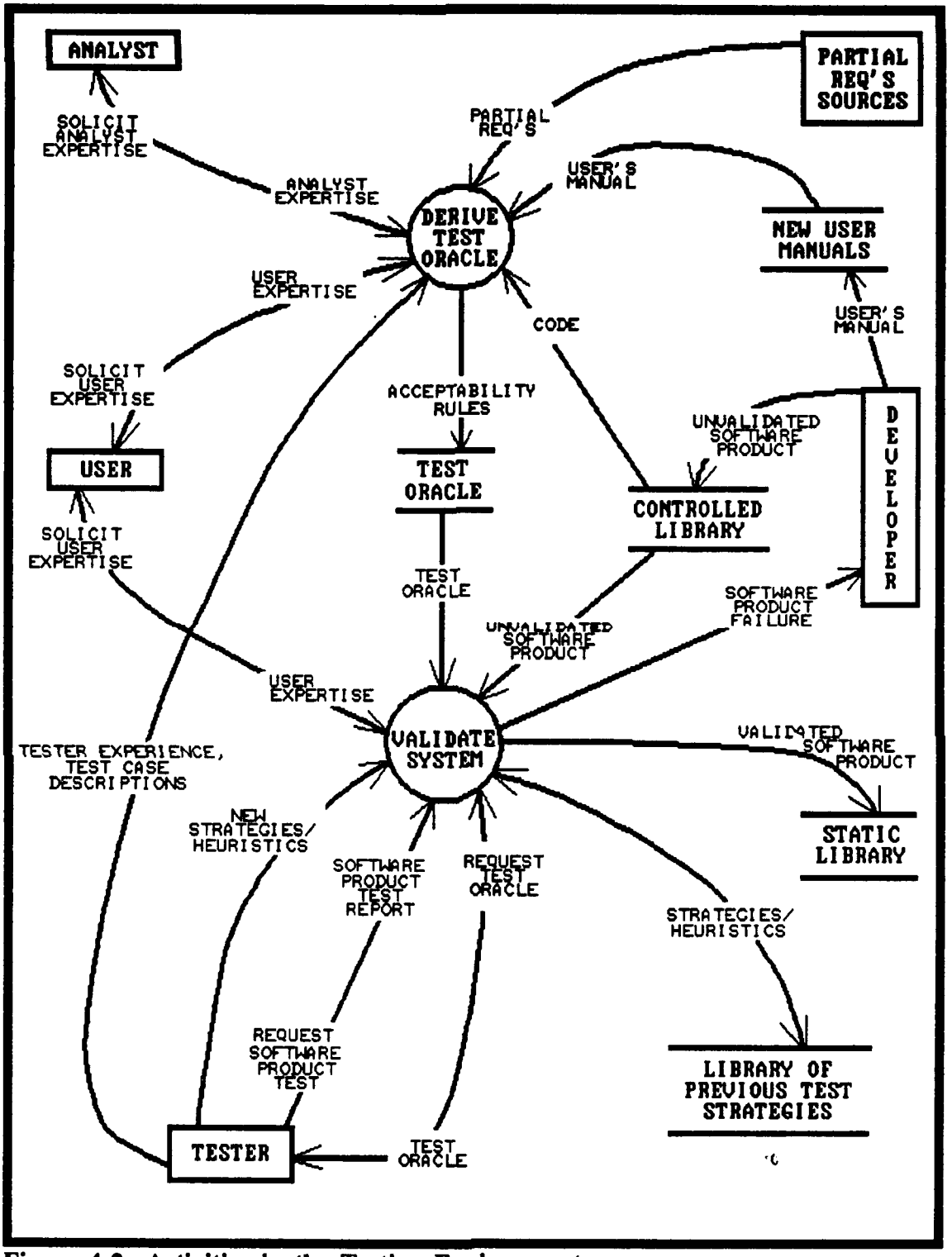

Figure 4-2 Activities in the Testing Environment

1. Derive Test Oracle

The primary area of concern in Figure 4-2 is the process labeled *derive test oracle*.[3] Six activities for deriving a test oracle are examined here. They are transforming code to specification, conducting user/analyst conferences (again), processing any existing, possibly useful, requirements, verifying new user manuals, reviewing abstracted requirements, and producing the test oracle. These activities are shown in Figure 4-3, which is a next level expansion of the *derive test oracle* process.

a. Produce Test Oracle

One definition of a test oracle was given in Chapter 1. At this juncture, however, further elaboration on the semantics of a test oracle is necessary. A test oracle is a statement of expected results or acceptability rules. In the ideal situation, the requirements specification serves as the primary source of information on which the test oracle relies for its information. The tester analyzes the requirements specification to determine sets of input data and the associated functions that process the input data. Also from the requirements specification the tester can determine the expected output for each associated pair of functions and input data sets. This expected output not only includes results when a function processes valid data (median plus boundary values of the input data set) but also includes results when the function processes invalid data (values outside the boundary of the input data set). Also, when

---

[3]The process labeled *validate system* is present to illustrate the interaction between the test oracle and validation. Also, *validate system* contains some ideas that, although not directly component to deriving a test oracle, can be used in conjunction with a test oracle to validate a system. These ideas will be discussed in section C.2.

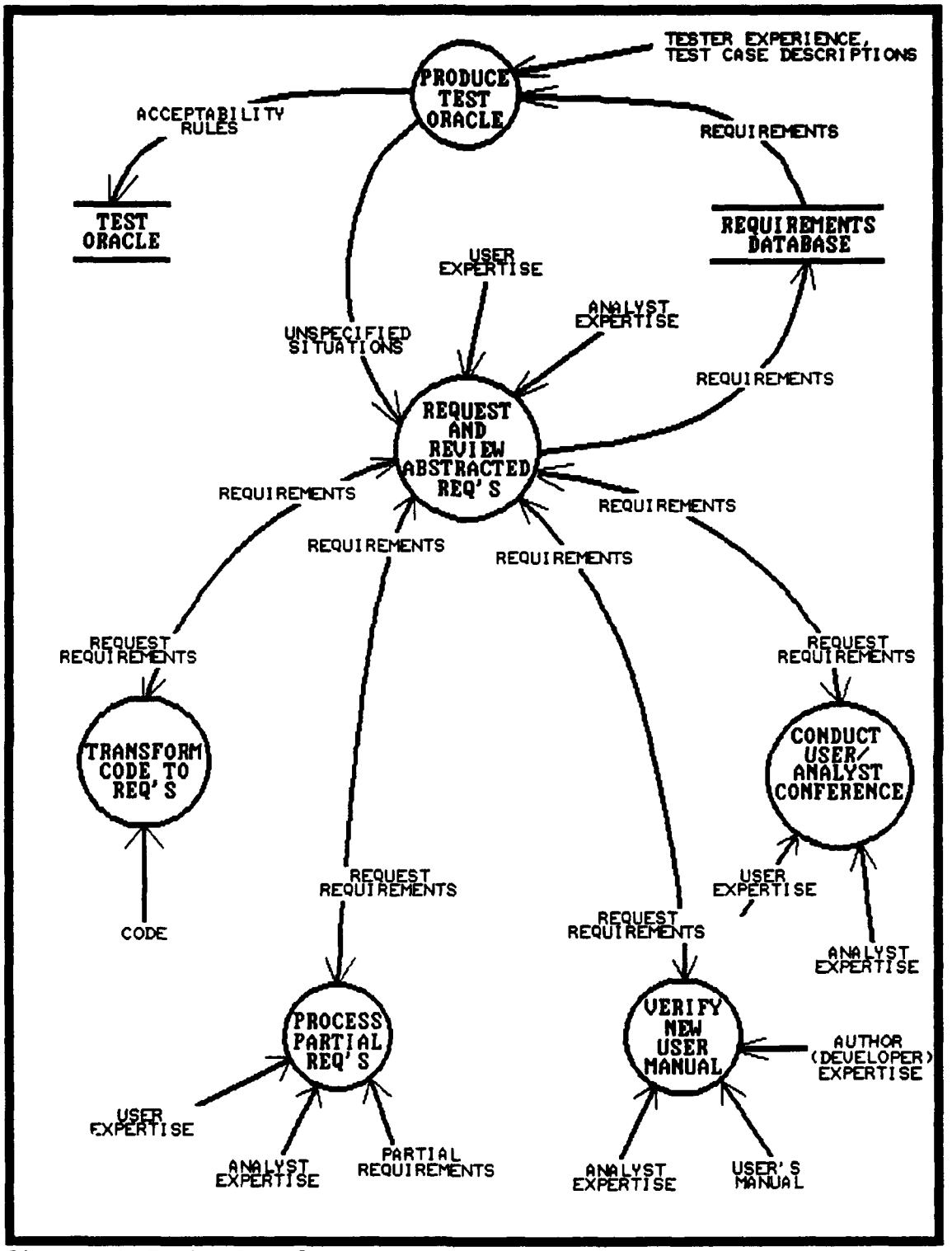

Figure 4-3 Derive Test Oracle

selecting the input data, the tester might consider looking at special values in the output data set to select input data that would generate those specific output data.

Thus, from the requirements specification, the tester can derive various, distinct combinations of input data, functions, and expected output. These combinations reflect the testing of sequence-dependent functionalities in addition to the simpler singular functionalities. The expected behavior then becomes the mechanism for determining the correctness of actual test results, i.e., observed behavior.

In the non-ideal situation where a usable requirements specification does not exist, the tester's job of producing a test oracle will require more effort. Furthermore, the tester still needs requirements information. Defining these requirements remains an integral element in deriving a test oracle in this situation. The process is different here. In this case the tester, based on his experience, determines the types of functions that need to be tested. He discerns these functions from a general flavor of system intent, design and code review, and intuition. These functions, however, may lack an explicitly stated description of their required behavior. The tester identifies these functions, and the conditions under which they are invoked, to a requirements derivation process for analysis. From these resultant requirements, the tester produces the test oracle.

A test oracle produced in an ideal situation is different from one produced in the non-ideal situation. The difference lies in requirements coverage. In the ideal situation the requirements specification is a thorough, wide-ranging source specifying the user's needs. From this source the test oracle reflects general expectations across the board. In the non-ideal situation, requirements coverage deals

with more specific situations. In the former case, the requirements are generated by the user and analyst based on what is envisioned. In the latter case, requirements are based on the tester's experience and intuition as applied to an artifact, i.e., the developed product, and verified by a requirements review process. The tester's challenge in the latter case is to evaluate the developer's product against what the user expects. Most likely that evaluation will involve determining what is acceptable behavior instead of what is desired behavior. The tester lacks information to define what is "best" and must use the less stringent requirement of what is "good enough".

For further elaboration on producing a test oracle, consider the following discussion of a trivial example, which is depicted in Figure 4-4.

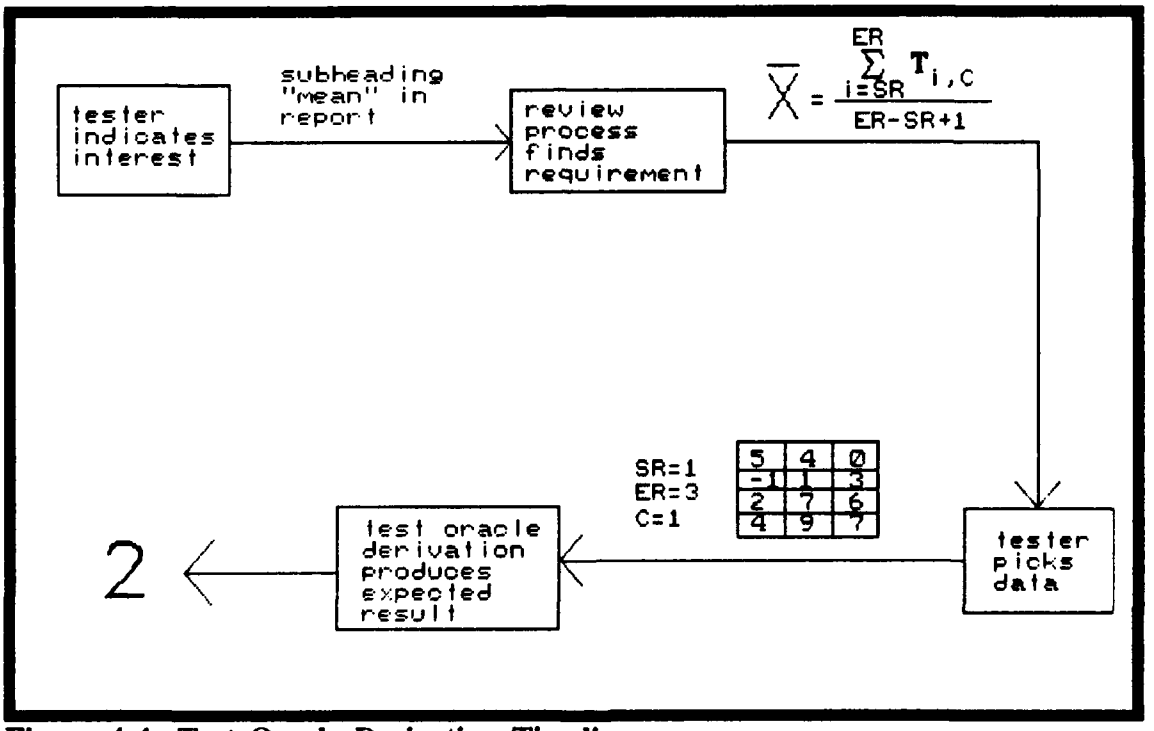

Figure 4-4 Test Oracle Derivation Timeline

Based on his own experience, the tester identifies a need to test a function computing the mean of some input values. However, he needs to know the requirement for this

function so he identifies this need to the requirements review process. The resultant requirement might be similar to the following statement:

> The function shall accept as input three values, denoted SR, ER, C representing a start row, an end row, and a column of table T, that are used to compute the output value $\overline{X}_{SR,ER,C}$ according to the following formula:

$$\overline{X}_{SR,ER,C} = \frac{\sum_{I=SR}^{ER} T_{I,C}}{ER - SR + 1}$$

From this requirement the tester selects the set of data to exercise the function. From this data the test oracle deriver produces the expected result. In this example, for SR=1, ER=3, and C=1 the expected result is 2 according to the formula.

Again, this example is trivial and doesn't address issues such as the type (e.g., integer, real) of values, the boundaries of input values, or the testing of responses to erroneous data. A more realistic example would involve producing a series of calculated results based on an initial set of input data and/or describing a series of system actions based on these calculated results.

b. Request and Review Abstracted Requirements

As has been previously noted, requirements are still necessary for test oracle derivation when no usable requirements specification exists. Since pure requirements analysis is not feasible or desirable, a strategy is needed to specify how and to what degree requirements will be abstracted. This strategy relies on the experience of the tester, a number of requirement-abstracting processes, and a

requirements review. The tester identifies a set of unspecified situations in the software product in question. He makes this identification based on his experiences. This set of unspecified situations then serves as a catalyst for abstracting requirements. A central authority will determine which (one, some, or all) of the requirement-abstracting processes to invoke that will best satisfy the unspecified situation. The resultant requirements are then reviewed, reconciled, and compiled for use by the test oracle deriver.

There are four requirement-abstracting processes discussed in this material. They are the transformation of code to requirements through reverse engineering, processing partial requirements, verifying new user manuals, and conducting a user/analyst conference. In these activities, the potential exists that numerous requirements will be identified that have not been designed or coded. Furthermore, some of the derived requirements may contradict each other, to a greater or lesser extent. These contradictions and incompletenesses must be resolved.

In the case of varying or contradictory requirements, the user and analyst must confer to resolve the discrepancy. If two requirements contradict and one emanated from the reverse engineering process, then the user and analyst should determine the acceptability of the reverse-engineered requirement. If the reverse-engineered requirement is at all acceptable in comparison to the conflicting requirement, then the reverse-engineered requirement should be retained based on the resources and cost expended thus far in its development. If two requirements contradict and neither are reverse-engineered requirements, then if either correlates to an acceptable reverse-engineered requirement, the correlating requirement should be

retained. If two requirements contradict and neither is reverse-engineered and neither corresponds to a reverse-engineered requirement, the user must decide which of the two requirements most accurately reflects his needs.

In the case where requirements are identified that have not been designed or coded, the user and the developer must independently and mutually determine the cause. The basis for a starting point for this determination is the original contract (Statement of Work) authorizing the developer to produce a product. Most likely any determination will be political and opinionated from one party's perspective against the other party's perspective. In the worst case, resolution will require legal action. In the best case, both the user and the developer will agree and one will accept responsibility. If the user accepts responsibility, then he must determine the fundamental necessity of the requirement relative to system functionality and the additional costs of implementing it. One consideration would be to defer implementing the requirement until the maintenance phase. If the developer accepts responsibility, then he has to bear the burden of implementing the requirement in an efficient and timely manner with no additional cost to the user.

The output of this requirements review process is a concatenation of requirements centralized in a requirements database. This requirements database is not considered a requirements specification primarily because it will not be complete. What it will reflect are specific cases of information that can serve as a basis for test oracle derivation. However, it may be useful as a baseline for subsequent requirements development and maintenance activities.

The discussion on the requirements-abstracting processes that follows assumes that the abstraction of requirements is initiated and motivated by the tester based on his experience. However, the material is presented in a way that lends flexibility to obtaining a more complete set of requirements in the case where an organization has the resources (e.g., time and money) and the desire to do so.

c. Transform Code to Requirements

It has only been in recent years that techniques and methods have been pursued for converting code to a specification, either design or requirements. This type of research is known generically as reverse engineering. As recently as 1988, Sneed and Jandrasics [Ref. 23] presented their efforts on transforming COBOL code to a requirements specification using automated tools. Currently, reverse engineering is applied in the maintenance phase. However, the context of this chapter calls for using reverse engineering in the development phase. The context, basis, and reasoning for using reverse engineering in the development phase and the maintenance phase are different.

(1) Reverse Engineering in the Maintenance Phase. During the maintenance phase a system previously has been validated, has been installed, and is generally operational. Reverse engineering is used primarily for two reasons. Firstly, the requirements and/or design specifications do not exist or have not been kept current, either of which can easily result from an aging system. Secondly, the operational system is a set of monolithic, unstructured programs that do not lend themselves to maintainability. Hence, the goal of reverse engineering is to abstract

from code a higher level view (design and/or requirements) in an effort to create a more conducive environment for maintenance. [Ref. 23]

Sneed and Jandrasics illustrate software levels by drawing a parallel to the conceptual, logical, and physical levels of database technology. The physical level of software consists of modules, maps, data descriptions, access paths, and command procedures. The logical level is the design language or methodology that describes logical processing units such as modules, data capsules, and interfaces. The conceptual level is a set of abstract entities such as data objects, data elements, processes, and relationships among the abstract entities. Sneed and Jandrasics refer to this conceptual level as the system specification (i.e., in the terminology of this research, a requirements specification). They propose abstracting the logical level from the physical level, and then to abstract the conceptual level from the logical level. This process is depicted, in a general way, in Figure 4-5. [Ref. 23]

The Sneed and Jandrasics process provides an exact representation of the actual software in terms of its elements, structures, and relationships. This representation may differ from the user's view, with the disparities reconciled by interaction with the user until a maintainable and testable description of the system is achieved. This description provides a baseline for defining and implementing new programs, based on the requirements of the previous programs. [Ref. 23]

(2) Reverse Engineering During Development. The context in which reverse engineering is used during the development cycle is different than when it is used in the maintenance phase. The system has yet to be validated; thus it has not been installed and consequently may not be completely operational. The justification

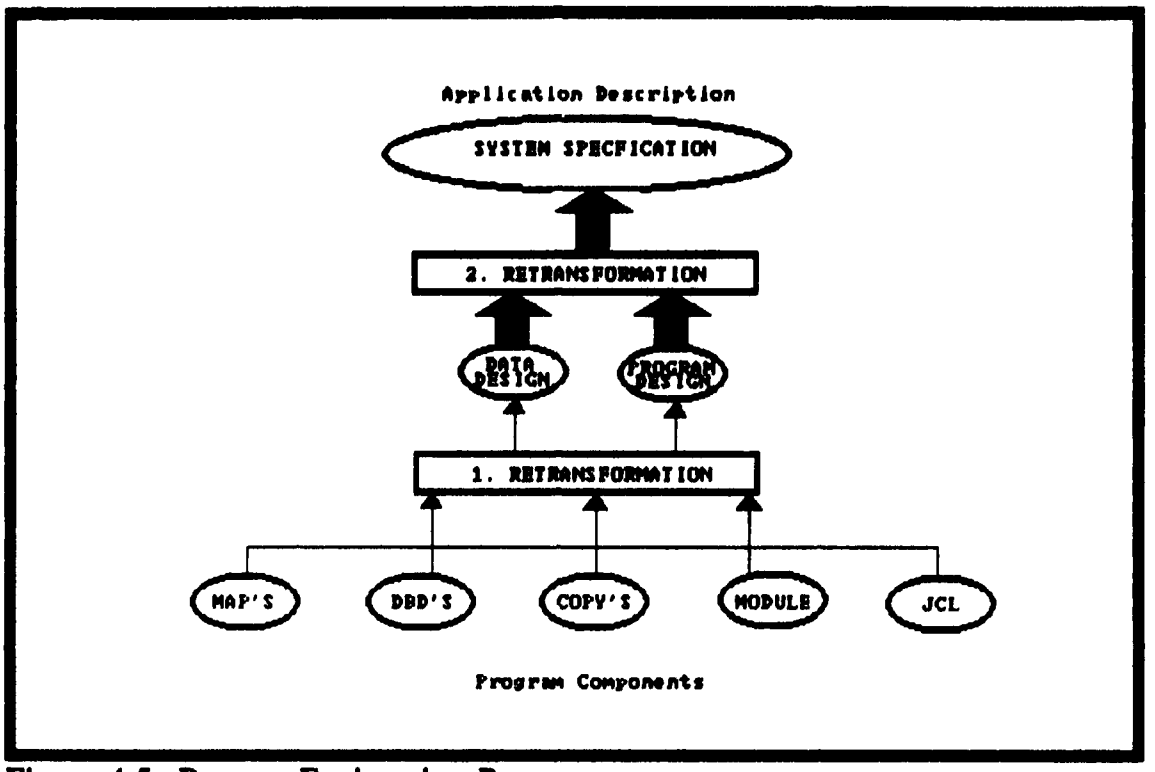

Figure 4-5 Reverse Engineering Process

for using reverse engineering in the development cycle prior to validation is to recover requirements that are implicit in the software source code and/or design. In the case where design specifications are current and accurate, the process can begin from that point. However, when the design specification is not reliable the transformation process must begin with the most recent product - the software source code. The result of the transformation process is a set of descriptions of the implemented operations. Of course, the completeness, quality, and quantity of these descriptions (abstracted requirements) will depend on the efficiency and validity of the reverse engineering process.

It might be argued that to abstract these requirements and then test against them only serves to validate the reverse engineering process and doesn't

really validate the software code against the user requirements. This is precisely why the user must be consulted at the end of the transformation process to approve the abstracted requirements. This interaction with the user to approve the requirements may produce several results. In the best case the set of requirements will reflect the user's needs and very few, if any, adjustments will have to be made. In other words, the reverse engineering process ideally will yield an accurate set of requirements from the informal requirements analysis embedded in the design and code. Two other cases are that the set of requirements produced will either be inadequate or will be a superset of what is needed. In either of these two cases, the issues will have to be resolved by a subsequent requirements review process. At a minimum, the abstracted requirements will serve to augment the requirements data base for the purposes of documentation, testing, and future maintenance activities.

d. Process Partial Requirements

It is likely that some requirements exist in fragmentary descriptions and documents. These requirements, once collated, may be useful for testing, especially if they reflect the user's needs and desires. Thus, the user must once again be consulted to determine the current status or validity of these collated requirements. Given a system where development has been continuing over a period of years, a requirement might not be current due to changed constraints, a changed usage environment, or a change in the perception of the requirements.

The collation process begins with a survey of all documentation relating to system development. This survey will produce a list of documents and/or sources (e.g., data stores) that contain system requirements. The next step is to extract

requirements from these documents for consolidation into a document (or sets of documents) strictly related to requirements. Since the survey will encompass both chronological as well as evolutionary documentation, the potential exists that requirements will surface that are contradictory and/or in different formats. Contradictory requirements that are not easily resolved should be noted and deferred for resolution in the subsequent global requirements review process. For those requirements that agree but are in different formats (e.g., natural English versus structural analysis), then the requirement closest to or resembling current methodology format should be retained and the other requirement discarded.

During collation, simultaneous prioritizing of requirements should take place. When the results of collation will be used for test oracle derivation, the priority should be placed on functionality. Thus, from a hierarchical standpoint, the largest functional requirements should be identified first. Remaining requirements can be classified as belonging to these functional groups with sub-levels of functionality identified as well. The degree of functional hierarchical leveling will depend on the complexity of the requirement as well as what is immediately necessary for the test oracle.

In essence, the point is that some existing, fragmentary requirements are still useful in a development environment where no effective requirements specification exists. These requirements should be condensed for further analysis during test oracle derivation.

e. *Verify New User's Manual*

The new user's manual can be used, to an extent, as an informal reflection of the functional requirements of the new system. Without verification of the new user manual by the user, testing against the user manual will only provide assurance that the developer was successful in reflecting the functionality of his code into text. Thus, the user must verify that the user's manual accurately states the procedural steps of his daily processing environment.

It is important to note that testing against a verified user's manual cannot provide complete system validation. The reason is that the user's manual describes the human interface, i.e., interaction with the user. What is not given in a typical user's manual are temporal and control events. In any case, a typical user's manual will describe input/output devices, formats for input and output data, and the procedural steps and timing necessary to exercise system functionality. Thus, if the verified user's manual states that a certain action will occur given a certain input, then part of a validation process should confirm that such activity occurs. From the functionality described in the user's manual, requirements can be synthesized to an appropriate format. In essence, if the user's manual is verified as acceptable by the user, then it is indeed an informal reflection of requirements.

One advantage (and thus a difference) of using the new user's manual to glean requirements information versus using the old user's manual (discussed in the next subsection) is the likelihood that the author is still readily available. The author's availability becomes important when features in the user's manual are inexplicable from the user's perspective. The author can explain the presence of these features in terms

of functionality and necessity. These features become additional requirements if the user accepts the explanation. If the user rejects the explanation, it is likely that unnecessary design and coding is embedded in the product which has other ramifications not considered here.

f. Conduct User/Analyst Conference

The last activity in Figure 4-3 to be discussed is *Conduct User/Analyst Conference* The initial reaction to this idea might be the thought that this is nothing more that regressing to the original requirements analysis phase to capture requirements. As previously mentioned, regressing to pure requirements analysis in a large software system development environment where the implementation phase is nearly complete would be costly and highly undesirable. However, the assumption here is that a full-scaled requirements analysis is not necessary. The degree to which the user and analyst need to confer will depend on the extent and presence of current requirements and a test oracle. The tester will also influence the extent of user/analyst interaction based on his needs to discern required behavior in various described situations for the product to be tested. Hence the idea of the analyst and user interacting to abstract requirements is proposed in the context of doing it in conjunction with the other activities in Figure 4-3 that were just previously discussed. Figure 4-6 is a next-level expansion of the process *Conduct User/Analyst Conference* and it depicts the main ideas occurring in a user/analyst conference.

If there is a predecessor system, it can be an excellent source of functionality when the logic of all or part of that system has been retained in the new system. This would be true when the implementation of the old system has become

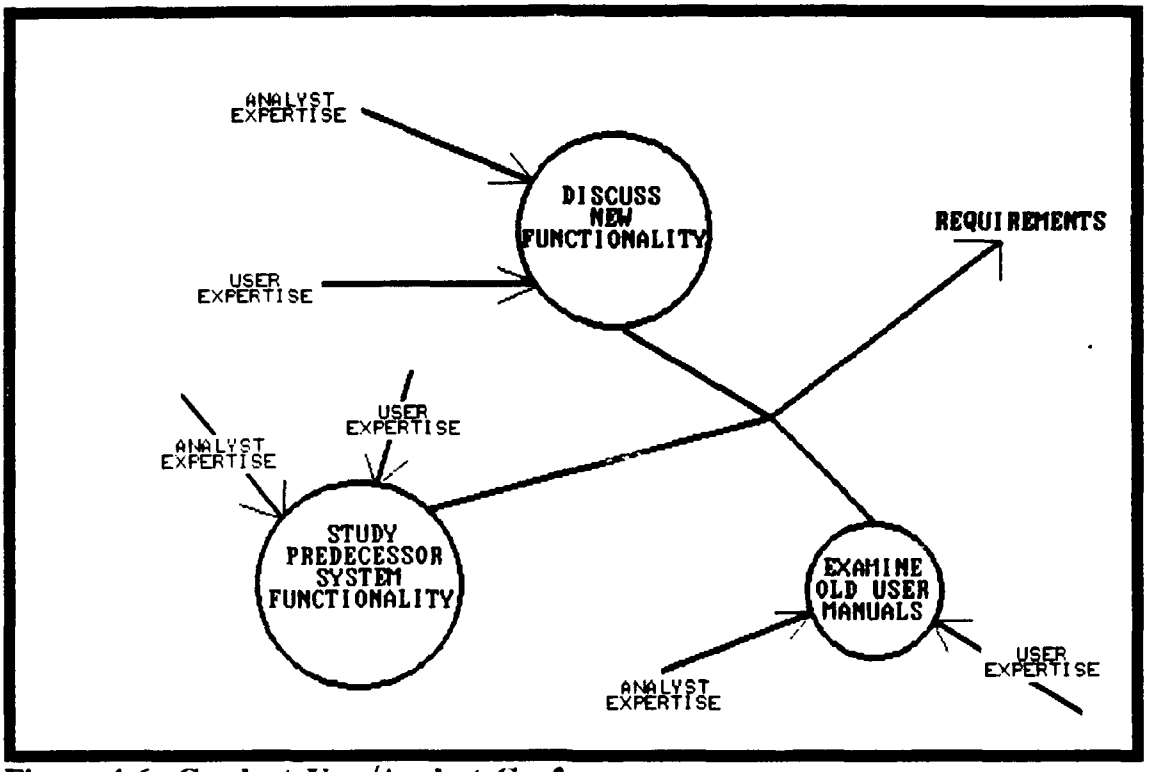

Figure 4-6 Conduct User/Analyst Conference

obsolete requiring a new system to bring current data processing in line with new technology and techniques. In this case the old system can provide certain input/output scenarios that should be present in the new system as well. These input/output scenarios provide oracles to test against the new system. Note also that the predecessor system does not have to be automated. Manual processes can serve as a source for expectations in just the same manner. However, more effort might be required to document manual procedures, especially when the knowledge is in the mind of an individual who is serving as a specialist.

Besides observing the predecessor system for functionality, other sources for requirements are old user manuals. In fact, this activity may or may not be executed separately from the activity of observing the predecessor system functionality.

In either case, the user manuals will describe some aspects of daily processing that should be inherent in the new system. The same arguments apply here that were presented earlier in the discussion on new user manuals (with the exception of access to the author).

Now of course where the new system implements new and additional functionality, the predecessor system and its user manuals will not be helpful. In this case the user and analyst will have to discuss those new functional requirements. The scope of their discussion, as directed by project management, will determine which type of requirements are discussed. Furthermore, consideration must be given to non-functional requirements that have not been recovered. Non-functional requirements are primarily constraints that deal with concerns such as response times, hardware, languages, ambient environment, reliability, and security. Besides evaluating functionality, the tester will also need to evaluate other characteristics of the software product. For example, the tester may want to evaluate that a function not only executes in a certain manner, but that it executes within a specified amount of time.

2. Validate System

There are two activities shown in Figure 4-2. One is *Derive Test Oracle* which has just been discussed, and the other is *Validate System*. The purpose of illustrating *Validate System* is not to explore the various means and techniques for system validation. Various methods for validation exist such as those discussed in Chapter 2. The purpose here is to abstract validation at a high level to depict its reliance on a mechanism for test generation and to lay the groundwork for discussing other considerations indirectly related to a test oracle.

The first consideration is a library of previous test strategies. The idea here is to build and maintain a library of testing techniques while recording the situation for which each technique is used. Additionally, this library would also contain a compilation of any rules of thumb, i.e., test heuristics, that were used. In the situation where a test oracle is derived, it is unlikely that all situations or requirements will be uncovered on the first pass, e.g., infrequently occurring events may not be specified. When such events do get discovered, the test oracle gets updated and test generation continues. However, this library of previous test strategies will facilitate test generation by providing the path to a level one higher than what is needed. Thus, what drives or influences different/various scenarios can be studied. The form and content of this library is a subject requiring further research.

The second consideration is that of a *kernel* system. In Figure 4-2 the kernel system would be the static library, which is defined by the IEEE Standard [Ref. 24]. This idea considers the development environment of a large software system where costs and schedules have been excessively exceeded, causing great anxiety and pressure to deliver. The kernel system would be a bare-bones system that would provide minimally acceptable functionality to allow the user to continue and accomplish his daily routine. Defining "minimally acceptable" might present a problem. Indeed, it might be that the only minimally acceptable system is the fully developed system that was originally envisioned. However, it is probably more likely that the absence of some functionality might be tolerable. By installing this kernel system, the development cycle would end and the maintenance phase would begin. One advantage would be that maintenance funds expenditures would begin while the expenditure of

development funds would cease. These two sources of financial funding are significantly different, especially as it applies to governmental budgetary procedures. Another advantage would be the psychological achievement of installing the system. This psychology would affect everyone involved, i.e., project management, developers, and the users. The disadvantage is that the maintenance phase would immediately begin with the task of completing the additional functionality that was intentionally left out. This would be an extra burden on maintenance activities, where the emphasis is to mitigate the cost and effort of making changes.

D. DRAWBACKS TO TEST ORACLE DERIVATION

The strategy proposed for deriving a test oracle presumes extensive access to the user. In reality, frequent and spontaneous access to the user may not be practical or possible. Yet even if frequent access is possible, it is likely that where a large software system is developed, the user community will be quite large (thousands) with varied levels of expertise and knowledge. The problem would be to determine which user level is required and to take advantage in the most efficient way of user interaction time. Also, the user/analyst interaction will occur frequently (more than one sitting), making it unlikely that the same user will always be consulted. The problem with this is the possibility and probability of receiving slightly different interpretations.

When deriving a test oracle, inevitably the process will uncover requirements that have not been implemented in the system waiting for validation. Resolving this issue will not be an easy one and will rest on the shoulders of the user, analyst, developer, and project management. Contracts, money, time, politics, and shrewd negotiation will

determine the final resolution. There are no easy solutions, especially where the missing functionality is fundamentally essential.

The proposed strategy relies heavily on the tester's experience and the ability of the tester to identify adequate situations that will lead to sufficient testing of the system. Since the tester decides which oracles will be generated, his proficiency will impact the degree of validation quality. Moreover, even with extremely efficient testers, the resultant test oracle represents a cross section of system functionality and not the more general, global view that is obtained in the ideal environment. This strategy does not attempt to derive a test oracle analogous to one derived in the ideal environment. Without having to expend additional large sums of money and time, the goal is to validate a system with reasonable assurance that the product is acceptable to the user.

Finally, the proposed strategy calls for transformation of code to requirements. Currently this science is still in the research phase: thus, many may consider it unperfected for practical use. When this technique is perfected, some concern must be given to the software tools performing the inversion. Also, the result of the inversion may be in a format different than the development methodology employed in the current environment, although this potential problem may be outweighed by the benefits of the reverse engineering results.

E. SUMMARY

Finding a specific solution for the M3S environment was not the goal of this research because of the extensive effort and time that would be required to find a specific solution for a large development environment. Rather, M3S was used as a

case study to motivate ideas as well as to provide perspective to the reader. What this research provides is a general framework that can be tailored to an individual situation. Some or all of the ideas can be used, depending on project management policy, the availability of resources, political circumstances, and time. For example, M3S may not be prepared to conduct reverse engineering of code to abstract requirements. However, assuming that M3S wants to recover requirements, the other processes besides automated reverse engineering offer some promise. Some form of reverse engineering could possibly be done manually, though the result is not likely to be as thorough and rigorous as the automated approach. For certain, requirements-based testing is necessary. Designed-based testing alone is not sufficient, since it does not reveal omitted functionality.

The major result of this thesis is a set of strategies used not only to recover requirements implicit in the system, but also used to derive an adequate test oracle for system validation. The strategies certainly are not the only ones that can be considered. For example, simulation might be useful as a source in deriving oracles. Also, a predetermined, existing table of values might be another source. However, since neither of these strategies were applicable to the M3S case study, they were omitted from this discussion.

V. CONCLUSION

This research is more of a general overview of a proposed strategy in that it is not presented as a detailed technique. The research concentrated more on identifying the issues and concepts, and incorporating those issues and concepts into a coordinated, understandable framework. Thus, no actual implementation of the proposed strategy is described. Additionally, there was no attempt to prove or measure this strategy through some set of metrics and actual experimentation. Although this would be possible, such activity is outside the scope of this research and is left as future research.

The material in this thesis addressed several questions (posed at the end of Chapter 2) relevant to current testing research. The user/analyst conference gleans requirements from the user, analyst, predecessor system functionality, and old and new user manuals. Additionally, based on the planned test cases, existing fragmentary descriptions were also noted as a source for requirements although a collation and review process is needed. Inverse transformation of code to requirements may provide some of the needed information but is unlikely to be a sufficient source of information for oracle derivation. Irrespective of the information sources used, conducting a valid and reliable test based on informally derived requirements hinges on the extent of the tester's experience and ability to identify appropriate situations since there is no approved specification to provide an objective global view of the product.

A. RESEARCH CONTRIBUTIONS

Within the limitations described above, this research has provided several contributions. First, this thesis identifies a number of sources for the information needed to determine the expected behavior of the software for a particular set of inputs. Several strategies are presented to obtain this information for conversion into a useful test oracle.

Second, this research describes a number of processes as well as the information flow between these processes. Each of the processes has been characterized and, in a few cases, a detailed description of the actions performed has been provided. For some of the processes (such as conduct user/analyst conference) no general, detailed description is possible since the context and content of such processes will vary widely from application to application. Others of the processes described in this thesis are theoretically possible to describe in detail, but further research is needed to provide such a description. The information provided by each process has been characterized, but more work is needed to identify useful forms and representations of this information.

Third, the combined effect of the identified processes and information flow is to derive a test oracle. By specifying the processes and information flow, this thesis provides a framework that may lend discipline, coordination, and thoroughness to an activity previously left to informal, ad-hoc efforts.

B. FUTURE DIRECTION

During the description of the process of test oracle derivation, a number of possible research topics have been identified. First, a number of the processes need

to be more thoroughly described by exploring the current manual (ad-hoc) activities and regularizing them into a consistent process description. Second, research is needed to automate a number of the individual processes. Such automation would reduce the reliance on analyst and user participation and allow the tester to proceed expeditiously with verifying the system under test. Third, research is needed to provide a strategy for automating the management of the overall test oracle derivation process.

LIST OF REFERENCES

1. Hetzel, B., *The Complete Guide to Software Testing*, 2nd ed., QED Information Sciences, Inc., 1988.

2. Miller, E., "Testing and Verification Problems in Industry: Technology Transfer", *Proceedings, Second Workshop on Software Testing, Verification and Analysis*, IEEE Computer Society Press, 1988.

3. DeMarco, T., *Controlling Software Projects: Management, Measurement, and Estimation*, Yourdon Press, 1982.

4. Nam, C. W., *Software Requirements Engineering: Experience and New Techniques*, PH.D. Dissertation, University of California, Berkeley, 1981.

5. Howden, W. E., "A Survey of Dynamic Analysis Methods," *Tutorial: Software Testing and Validation Techniques*, 2nd ed., IEEE Computer Society Press, 1981.

6. *IEEE Standard Glossary of Software Engineering Terminology*, ANSI/IEEE Std. 729-1983, The Institute for Electrical and Electronics Engineers, Inc., 1983.

7. *Guideline for Lifecycle Validation, Verification, and Testing of Computer Software*, Federal Information Processing Standard 101, National Bureau of Standards, 1983.

8. *IEEE Guide to Software Requirements Specifications*, ANSI/IEEE Std. 830-1984, The Institute for Electrical and Electronics Engineers, Inc., 1984.

9. Boehm, B. W., "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, January, 1984.

10. Beizer, B., *System Testing and Quality Assurance*, Van Nostrand Reinhold Company, Inc., 1984.

11. Lamb, D. A., *Software Engineering: Planning for Change*, Prentice Hall, Inc., 1988.

12. Myers, G. J., *The Art of Software Testing*, John Wiley & Sons, Inc., 1979.

13. Vyssotsky, V. A., "Common Sense in Designing Testable Software," *Program Test Methods*, W.C. Hetzel ed., Prentice Hall, Inc., 1973.

14. Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold Company, Inc., 1983.

15. *SASSY Management Unit Procedures*, Marine Corps Users Manual, UM 4400-124 w/Change 3, 28 April, 1989.

16. *M3S Development Status*, July, 1983.

17. Yourdon, E., *Modern Structured Analysis*, Yourdon Press, 1989.

18. *Audit/Review Report for Marine Corps Standard Supply System (M3S)*, Electronic Data Systems Federal Corporation, Bethesda, Md., March, 1988.

19. *M3S General Design Specification*, April, 1987.

20. Cardenas, A. F., *Data Base Management Systems*, 2nd ed., Allyn and Bacon, Inc., 1985.

21. *Functional Requirements Definition*, IRM 5231-04, United States Marine Corps, July, 1987.

22. *Marine Corps Standard Supply System (M3S) Test Plan*, September, 1989.

23. Sneed, H., and Jandrasics, G., "Inverse Transformation of Software from Code to Specification," *Proceedings, IEEE Conference on Software Maintenance*, IEEE Computer Society Press, 1988.

24. *IEEE Guide to Software Configuration Management*, ANSI/IEEE Std. 1042-1987, The Institute for Electrical and Electronics Engineers, Inc., 1987.

BIBLIOGRAPHY

Andriole, S., ed., *Software Validation, Verification, Testing, and Documentation*, Petrocelli Books, Inc., 1986.

Arango, G., and others, "TMM: Software Maintenance by Transformation," *IEEE Software*, May, 1986.

Biggerstaff, T., "Design Recovery for Maintenance and Reuse," *Computer*, July, 1989.

Brooks, F., *The Mythical Man-Month - Essays on Software Engineering*, Addison-Wesley, Inc., 1975.

Chow, T., *Tutorial: Software Quality Assurance - A Practical Approach*, IEEE Computer Society Press, 1985.

Freeman, P., and Wasserman, A., eds., *Tutorial: Software Design Techniques*, 4th ed., IEEE Computer Society Press, 1983.

Howden, W., *Functional Program Testing and Analysis*, McGraw-Hill, Inc., 1987.

Howden, W., and Miller, E., eds., *Tutorial: Software Testing and Validation Techniques*, 2nd ed., IEEE Computer Society Press, 1981.

Quirk, W., and others, *Verification and Validation of Real-Time Software*, Springer-Verlag Berlin Heidelberg, 1985.

Vick, C., and Ramamoorthy, C., eds., *Handbook of Software Engineering*, Van Nostrand Reinhold Company, Inc., 1984.

Eleventh International Conference on Software Engineering, IEEE Computer Society Press, 1989.

Third International Workshop in Software Specification and Design, IEEE Computer Society Press, 1985.

Fourth International Workshop on Software Specification and Design, IEEE Computer Society Press, 1987.

IEEE Software Maintenance Workshop, IEEE Computer Society Press, 1983.

IEEE Conference on Software Maintenance, IEEE Computer Society Press, 1985.

IEEE Conference on Software Maintenance, IEEE Computer Society Press, 1988.

Proceedings, Workshop on Software Testing, IEEE Computer Society Press, 1986.

Proceedings, Second Workshop on Software Testing, Verification, and Analysis, IEEE Computer Society Press, 1988.

Proceedings on Computer Software and Applications Conference (COMPSAC) 83, IEEE Computer Society Press, 1983.

Proceedings on Computer Software and Applications Conference (COMPSAC) 84, IEEE Computer Society Press, 1984.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Commandant of the Marine Corps Code TE 06 Headquarters, U.S. Marine Corps Washington, D.C. 20380-0001	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	1
5.	Professor Timothy J. Shimeall, Code 52Sm Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	10
6.	LCDR Rachel Griffin, USN, Code 52Gr Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	3
7.	Professor Elaine J. Weyuker Department of Computer Science New York University Courant Institute of Mathematical Sciences New York, New York 10012	1
8.	Professor William E. Howden University of California, San Diego Department of Computer Science and Engineering Muir Campus, Mail Code C-014 La Jolla, California 92093	1

9. Nicholas J. Retza — 1
3104 Harvest Lane
Albany, Georgia 31707

10. Colonel Larry Richards — 1
Principal Director, Code 70
IRMD Building 3700
MCLB, Albany, Georgia 31704

11. Major Richard Miller — 1
3626 Berkeley Rd
Albany, Georgia 31707

12. Major John Kraus — 1
1942 Georgia Ave
Albany, Georgia 31705

13. Capt J. A. Hernandez, Jr. — 4
42 Forest Ave SW
Moultrie, Georgia 31768

14. Capt Doug Turlep — 1
2300 Bluewater Dr Apt F54
Albany, Georgia 31705