To the Graduate Council:

I am submitting herewith a thesis written by Mark A. Floyd entitled "Software Events Tracking System." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

J. H. Poore, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Vice Provost
and Dean of the Graduate School

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Master's degree at The University of Tennessee, Knoxville, I agree that the Library shall make it available to borrowers under rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of the source is made.

Requests for permission for extensive quotation from or reproduction of this thesis in whole or in parts may be granted by the copyright holder.

Signature Mark a. Floyd

Date April 7, 1988

SOFTWARE EVENTS TRACKING SYSTEM

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Mark A. Floyd

June 1988

Copyright © Mark A. Floyd, 1988
All Rights Reserved

DEDICATION

To Matthew and Michael, the two best sons that a father could hope to have, my father and mother, who taught me the value of a good education, and my wife Robyn, whose strength, support, and patience I have leaned on for eleven years.

ACKNOWLEDGMENTS

I wish to acknowledge the guidance and many contributions of Dr. Jesse Poore to this thesis project. I am fortunate to have studied under his influence, and I hope that he feels that his faith in me has been justified.

I wish to thank my management at Martin Marietta Energy Systems, Inc., especially Dr. Roy Morrow, Ms. Rose Wood, and Dr. Jim Rushton, for their encouragement and support.

I wish to express my appreciation of Dr. Ronald Leinius and Dr. Jeff Case for their service on the thesis committee and for their helpful suggestions in developing this thesis.

I wish to extend a special thanks to Kim Cobb and Bonnie Sexton for freely giving their time to participate in the walkthrough of the SETS code.

Finally, I wish to thank my wife Robyn, whose support and patience over the past five years have been appreciated more than she could ever know.

The following are registered trademarks of the Digital Equipment Corporation:

VAX

VMS

RMS

VAXTPU

ABSTRACT

Lack of reliability is a problem that continuously plagues computer software. One approach for combating unreliability in software is through study of failure data collected from developmental and operational software projects. A system for comprehensive collection of interesting software events is vital to the accumulation of failure data that may be helpful in understanding software reliability problems. This thesis prescribes a well-defined data base of information about software events as an approach to identifying and understandinging aspects of unreliable software.

The focus of this work is primarily upon the question of what data are to be collected and the method for collecting these data. The nature of software errors is discussed with emphasis on characteristics and relationships to reliability. A summary description of selected software reliability models and an explanation of the kinds of data that are required to use these models is provided.

The Software Events Tracking System (SETS) is described in detail. This program makes possible the collection of critically needed data about software performance and changes to software. This data base will promote greater insight into the error phenomena of modern software systems. This insight will lead to better understanding of the error phenomena and to the development of new tools for increasing the reliability of software.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A. Description of the Problem

In today's technology-reliant world, computer software has assumed a vital role. Modern society now entrusts control of many of its most important functions to a computer and its associated software. Airlines, the armed forces, hospitals, police and fire departments, public utilities and many other organizations depend upon computer software to perform their work effectively. In fact, without the aid of computer software, many essential tasks would be impossible.

But in spite of society's dependence upon computer software, the reliability of the software in most systems leaves much to be desired. In many cases an extreme amount of "downtime" is tolerated because no one seems to be able to correct certain problems. The creators of software continue to produce unreliable, failure-prone products – even when the software is designed for use in highly critical situations.

The buyers of complex, expensive software have begun to see the need for developers to provide a software "warranty" with regard to the probability that their product will perform as specified [24]. Specification of such reliability is customarily required of developers of hardware systems. However, the mechanisms that induce failure in each of the two systems are entirely different.

IEEE Standard P1044 defines a software error as "an element of software code that when compiled, employed, or otherwise used results in an anomaly."[17] An error, if encountered, may cause a software failure. IEEE Standard P1044 defines a software failure as "the inability of a system or system component to perform a required function within specified limits." [17] The failures of software depend entirely upon the number and kinds of errors it contains and the probability the errors will be encountered during execution. Reliability improves as errors are discovered and corrected; error-free software is inherently 100% reliable.

There is no counterpart to this situation in hardware systems. And aging and degradation, two elements related to hardware failures, play no role in software failures. These differences notwithstanding, the concepts developed in reliability theory for hardware systems have provided a starting point for modeling software failures [6]. The approaches include probabilistic models that aim at predicting reliability and other elements of software quality on the basis of program properties such as size and complexity, and statistical models that base reliability prediction on an analysis of failure data. Papers comparing the competing models have appeared in the literature [1,4,12,13], but because there is a paucity of data on which to base the tests, these comparisons appear to be premature. A principal problem in this area has been the lack of a data base of systematically gathered failure data from various projects.

B. Failure Reporting

Software failures present a diverse and difficult problem. They range from the trivial (e.g., a syntactic error that results from misuse of a language and is detected by the language processor) to the complex (e.g., a logic error that results only from the execution of a peculiar combination of program logic paths and is detectable only by constructing an input case that forces the execution of that combination) [11]. Because of the subtleties involved in identifying these problems, a software failure reporting system is necessary.

The purposes of a failure reporting system are:

- To ensure that the errors causing failures are corrected, not forgotten.
- To ensure that all error corrections are approved before changes are made.
- To enable the measuring of error correction progress.
- To provide feedback to the user on the failure status.
- To prioritize the order in which errors are corrected.
- To collect data on such items as frequency and type of failures and time and cost of fixing errors.
- To apply the data, using software reliability models, for predicting the future reliability of software.
- To correlate errors by the symptoms of the exhibited failures.
- To allow the detection of recurring errors.
- To calculate the mean-time-to-failure (MTTF) of software.

Unfortunately, in the typical software life cycle, many errors will occur before a formal tracking system is utilized [31]. Errors encountered in the analysis and design phases are generally not tracked, since they are usually corrected in the emerging specification and design as they are discovered. Even during the code and checkout phase, failures usually are not tracked; they may be numerous, and the time required to fix them is often less than the time required to report. Often, it is only after a software product begins acceptance testing or is released to the customer that a formal tracking system would be considered. However, it is well documented that the earlier within the product life cycle that an error is discovered, the less costly it is to fix [31]. Collecting software failure data early on may lead to the identification of the phase in a project in which most errors are introduced.

C. Methodology of the Solution

One approach to combating unreliability in software is to learn as much as possible about the nature of software errors by studying failure data collected from developmental and operational computer software. A suitable failure data collection system is vital to such an approach. This paper considers the establishment of a well-defined data base of information about software failures as a useful method for describing such failures and for understanding unreliability in software.

A study was performed to determine the data parameters needed to support management of software development and maintenance, research activities, model

developments, and analysis efforts in the areas of software research (software reliability, software error analysis, life-cycle cost analysis, software productivity, and complexity). This paper primarily focuses upon the question of the kinds of failure data that are to be collected and the method for collecting the data. The nature of software errors is discussed in Chapter 2, with emphasis on characteristics and relationship to reliability. An explanation of the kinds of data required to use software reliability models is provided in Chapter 3. The Software Events Tracking System (SETS), a software program that allows users to track significant software events, is described in Chapter 4. A quality evaluation of SETS, including standard logic and design structure metric analysis and the results of a walkthrough inspection analysis, is presented in Chapter 5.

CHAPTER 2

THE NATURE OF SOFTWARE ERRORS

A. Software Characteristics

Glass [11] states that software is the most critical element in a computer system because:

- It is the most expensive element.
- It has absolute control over computer system response.
- It provides essential adaptability.
- It is on the "critical path" in system development.

Because software is critically important, it is only logical that much effort would be expended in building the most reliable software possible. Attempts have been made to establish engineering standards for software similar to those that exist in the much older electrical and electronics engineering disciplines responsible for computer hardware. Computer hardware has been much more reliable than software, and hardware reliability theory is better understood. Some researchers [14,19,22,25,30,32] have attempted to look at software reliability by using hardware reliability techniques. Other researchers [15,23,26,27,36] are studying the

characteristics of software separate from those of hardware. They attempt to isolate characteristics inherent in software and to define a method for measuring each one. Characteristics or attributes that have been defined for objective measurement are referred to as quantitative; those that require some subjective evaluation are called qualitative characteristics. As more and more characteristics are identified and data on software parameters are collected, a better understanding of software relationships will improve capabilities to build more reliable software in a shorter time frame.

Some basic quantitative parameters, discussed in IEEE Standard P982 [18] are:

- McCabe's cyclomatic complexity metric
- Halstead Software Science numbers
- Module size (lines of code)
- Number of direct interfaces with other procedures
- Module cohesion
- Information flow between modules
- Number of comments
- Pages of documentation

Characteristics in the qualitative set, as suggested by Boehm [3], include:

- Completeness

- Conciseness
- Consistency
- Cost (dollars)
- Development time (months)
- Efficiency
- Effort (man-months)
- Maintainability
- Portability
- Reliability
- Speed (seconds)
- Structuredness
- Understandability

It is not the objective of this work to define each of these parameters precisely, but instead to illustrate the enormous complexity of the problem of understanding software and to indicate the uncertainties related to the usefulness of these factors. One of the most important elements in the measurement of software quality is reliability. It will be demonstrated that much can be learned about reliability – and, in turn, quality – from a study of the failure history of software.

B. Error Parameters

In the most basic sense, the fact that a failure has occurred in any context is subjective and requires an external human judgment and a comparison to the correct result or action. The degree of the deviation determines the seriousness of the failure. Depending upon the opinion of the observer, the occurrence of a failure may be serious or minimal. Whether a failure occurs or not depends upon its relative significance to a single individual or decision-making entity. The relative significance of a failure may change when system reorganization takes place or tasks are restructured to form new tasks or functions.

Studies in system psychology [7] reveal the following broad types of failures:

- Incorrect performance of a required action
- Failure to perform a required action
- Failure to perform a required action in sequence
- Failure to perform a required action in the allotted time period
- Performance of a non-required action

These categories suggest three basic concepts that are central to systems: 1) performance of required (predetermined) actions or operations; 2) selected sequence of the actions; and 3) a predetermined time limit. These concepts, which are applicable to general systems, are also appropriate for computer software. When software fails to perform its specified functions, a software error is said to exist. Shooman defines a software error as "an event which occurs when the

software system is subjected to an input condition such that, due to a flaw in information transfer, the resultant output will be different from its true value according to design specifications." [30]

Some of the more basic questions dealing with software errors were posed by Endres [9].

- Where was the error made? What is the distribution of errors by modules? Are there any accumulations, i.e., modules, that were hit especially hard? If so, what are their functions? How are they structured?

- When was the error made? Errors may be made in each phase of the development cycle, beginning with the external design of the project. They can occur during detailed planning of the logic structure, during the original coding phase, while correcting the error, etc.

- Who made the error? This question may be evaluated in terms of the responsibility of individual groups during the project cycle (design or implementation) or in terms of individual sub-projects or even programmers.

- What was done wrong? Which particular programming task was not solved?

- Why was this particular error made? What caused the error?

- What could have been done to prevent this particular error?

- If the error could not be prevented, by which procedure can this type of error be detected?

Answering these questions provides a major stimulus to the gathering of data on software failures. These data should provide a broad description of the error event.

A lack of basic data has hindered solutions to these and other questions [29]. Other important questions involve the relationships:

- between errors and the structure of the software
- between errors and complexity
- between efficiency and reliability
- between size and number of errors or types of errors

The list could go on and on. Each characteristic attached to software errors may be related or correlated with every other characteristic to introduce factorial combinations of relationships that could produce a staggering amount of data to be analyzed. The creation of the data base is the first step in the analysis of the data.

There are a number of quantitative measures we can apply to software errors [10,18].

- Error Rate $(E) = \frac{\text{Number of errors}}{\text{day}}$
- Error Density $= \frac{\text{Number of errors}}{\text{Number of lines of functional code}}$
- Operational Error Ratio $= \frac{\text{No. of errors in time period } t}{\text{No. of errors in time period } t-1}$
- Error Distributions

Increased understanding of the nature of errors can be instrumental in increasing the level of reliability present in software.

C. Software Reliability

The ideal computer program would be one which would perform exactly as specified, with zero failures. Computer practice is far from this ideal. Only Mills and colleagues [5,21] at IBM's Federal Systems Division employ a methodology which has zero defects as an explicit goal. Problems exist in translating logical models into software, and there are many more problems involved in getting the errors out after the transaction is finished. Since errors prevent software from performing its function properly, the concepts of software error-proneness and software reliability are inverse functions of one another.

It is possible to itemize at least several of the factors that contribute to unreliability [9]:

- Lack of Intellectual Control
- Incomplete user's requirements
- Faulty system design
- Lack of experienced programmers
- Lack of programming management
- Lack of documentation
- Faulty systems integration

- Lack of user's training
- Hardware/software interface
- Operational problems

Most of these factors involve problems in the transfer of information, which substantiates Shooman's definition of software errors [30]. Thus, knowledge of information and its transfer would appear to be helpful in solving problems associated with reliability.

Today, many reliability questions are being researched [29] as software designers seek methods of producing less costly and more reliable software. Some of these questions include:

- What is the methodology of optimizing module size per error rate?
- What number of instructions could a programmer code in one day with zero errors or with minimum errors?
- Is it possible to standardize blocks of code to handle many different problem situations while combining the blocks into a highly reliable program?

These questions are only a few of the many awaiting answers from reliability research. A properly constructed data base can be a valuable tool in this effort.

CHAPTER 3

DATA BASE DATA REQUIREMENTS

A study was performed to determine the data parameters needed to support management of software development and maintenance, research activities, model developments and analysis efforts in the areas of software research (software reliability, software error analysis, life-cycle cost analysis, software productivity and complexity). The various software reliability models discussed in the literature have been classified as measurement, estimation, or prediction models [16]. Measurement implies that the software operates over a period of time and segments of operation are scored as failure or success. Estimation involves taking sample reliability measurements to estimate future reliability. Prediction is a reliability statement not based on a measurement of the operation of the software, but on the actual or anticipated attributes of the software (such as the number of lines of code).

A review and comparison of nine software reliability models is discussed by Cobb [4]. The required data for these models include the date the failure was detected. From this information, the calendar time between failures, the number of failures per reporting period and cumulative number of failures detected at a certain date can be computed. Additional information that provides more meaning to the results includes: dates for testing phases; the date the error was corrected; failure descriptions (including type and severity); operating mode and

processing rate of the computer; stress type and measure (i.e., a measure of the amount of code exercised); module attributes; resources spent in correcting the error; and the phase in which the error was introduced [8]. The cumulative number of corrections made and the number of errors detected and corrected per time-interval may be determined from these data.

Shooman [29] states that the following information is needed to establish a software reliability database:

- A paragraph or two describing the nature and scope of the projects, the organization involved, the time frame, the outcome, field performance, and any predecessor and successor projects.
- A brief, high-level summary of the requirements and specifications.
- The program language, compiler, operating system, major tools used and target machine.
- The size of the program (source and object, comments and executable instructions), the development approach, key milestones and the number of man-hours expended on the program.
- The number of errors discovered and corrected each week (or month) during integration testing and, if possible, during module testing.
- For development testing and for simulation testing, the number of tests, level, severity, running time to failure and running time without failure.

The usefulness and accuracy of these models depends on the availability of good failure data. Progress in software reliability modeling is seriously impeded

by lack of a sufficiently large data base of failure data. If companies and military agencies required adequate data collection on projects, there would be an ample database for retrospective studies. Unfortunately, much of the data collected are either fragmentary or are missing key facts necessary for software reliability modeling [29].

A. IEEE Classification for Software Failures

The IEEE Draft Standard P1044 provides a common approach to the classification of failures in software and its documentation [17]. This standard describes a minimum set of primitives and classifications deemed necessary for a complete data set. The classification scheme in this standard considers the environment and the activity in which the failure occurred, the symptoms and the software or system cause of the failure, where the failure originated, the resolution and disposition of the failure, and the corrective action appropriate to prevent similar failures from occurring in the future.

IEEE Draft Standard P1044 identifies those essential classifications and failure categories needed to establish a common definition and provide a common terminology and concepts to communicate among projects, business environments and personnel. This standard does not attempt to be exhaustive; some projects will have requirements which are unique, but which that project will want to track and classify. The standard makes no attempt to specify a standard format and data base structure. This is left to the requirements of each user.

The development of a software system for collecting these data specified by IEEE Draft Standard P1044 must be flexible and allow for unique project requirements. The system must be applicable to any software project, be easily adaptable to research and management needs and be easily changed as the standards mature and change. The interactive computer program described next will allow a user to easily collect the data necessary to establish a software failure data base. The program makes extensive use of a free-text data entry interface. A free-text interface allows users and organizations to establish their own unique data requirements for a software project. These data may then be entered into the system with a standard text editor. Very few formal structured data fields exist in the program. Those that do exist are data fields that will be common to all software projects. Chapter 4 will discuss the Software Events Tracking System which has been developed to establish a software events data base.

CHAPTER 4

SOFTWARE EVENTS TRACKING SYSTEM (SETS)

An interactive software system named Software Events Tracking System, hereafter referred to as SETS, has been developed to allow a user to collect and track significant software events. These events include software failure data and software change data. Data describing software projects, software modules and software enhancements may also be collected and tracked.

A. Conceptual Model

Basic entities present in the conceptual model for SETS are:

- Project.
- Module.
- Events (e.g., Failures, Software Changes, etc.).

Relationships present in the model are:

1. One project can be composed of several modules.
2. Certain events (failures, changes, enhancements, etc.) have occurred during a project's life.
3. Certain data entries have been recorded about these events.

These entities and relationships are depicted in Figure 1.

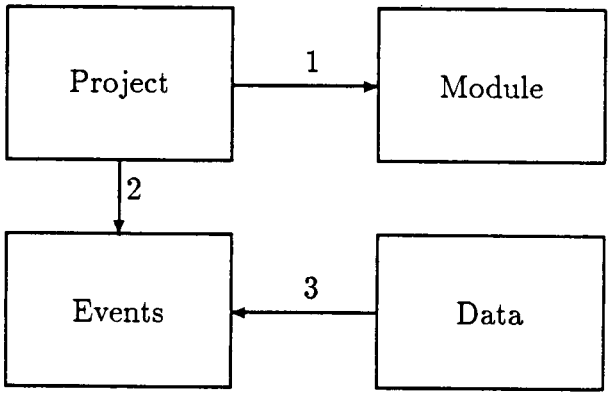

Figure 1. Conceptual Module for SETS

Using an entity-relationship diagram [33], in which rectangles are entities and arrows are relationships, we see a one-to-many relationship between a project and its modules (relationship 1). Relationship 2 is also one-to-many in that an event is recorded for one project, although a project may have many events.

If we assume that the relationships in Figure 1 are represented in our data structure as links, then if link 1 is traversed for a particular project, a record of all its modules could be obtained. A traversal of links 2 and 3 for a particular project gives all the data recorded for software events.

B. Physical Design of the System

SETS was implemented as a relational system on the University of Tennessee Computer Center VAX computer which operates under the VMS V4.7 operating system. The data attributes for each entity shown in Figure 1 were defined as *relations*. Each relation was designed to be in the third normal form. Third normal form relations provide freedom from insertion and deletion anomalies and freedom from data redundancy [33]. A list of the relations and their associated files can be found in Appendix A.

File Design

The obvious way to represent a relation is as a file whose record format consists of one field for each attribute of the relation scheme. Each relation has a set of attributes that uniquely determine the other attributes in the relation (i.e, a key). The seventeen permanent relations in SETS were implemented as indexed-sequential files with the attributes that form a key for the relation serving as a key for the file.

All of the data files contain the project number as part of the key. The project number provides the link between a project and its associated module and event data. The free-text data files were implemented by storing one line of text as one record. A line number field is stored as part of the text key. The line number field ensures that SETS retrieves the text data in the same order in which it was

entered. A maximum of $2^{16} - 1$ (65, 535) lines of text can be stored in each of the free-text data files.

C. SETS Software

The different components of the SETS are depicted in Figure 2, SETS's System Overview Chart. SETS was written exclusively in the machine-independent C language and utilizes the CURSES Screen Management Facility [34]. CURSES is comprised of functions and macros that create and modify defined sections of the terminal screen and optimize cursor movement. SETS also uses the VAX specific Record Management Services (RMS) routines. RMS provides the interface between SETS and the Indexed Sequential Access Method (ISAM) files that are used for data storage. The VAX Text Processing Utility (VAXTPU) [35] is used to provide the free-text editor interface to SETS. SETS consists of approximately 11,000 lines of code (excluding comments) and 211 modules.

Data Structures

Most of the data structures used by SETS are volatile, created when they are needed and destroyed when their useful life is finished. These data structures have similarities of form, although their memory requirements vary. Linked lists were chosen for the SETS data structures because of the need to preserve a sequence of data in the order in which they were entered. Text data will easily conform to a list data structure. Linked lists are:

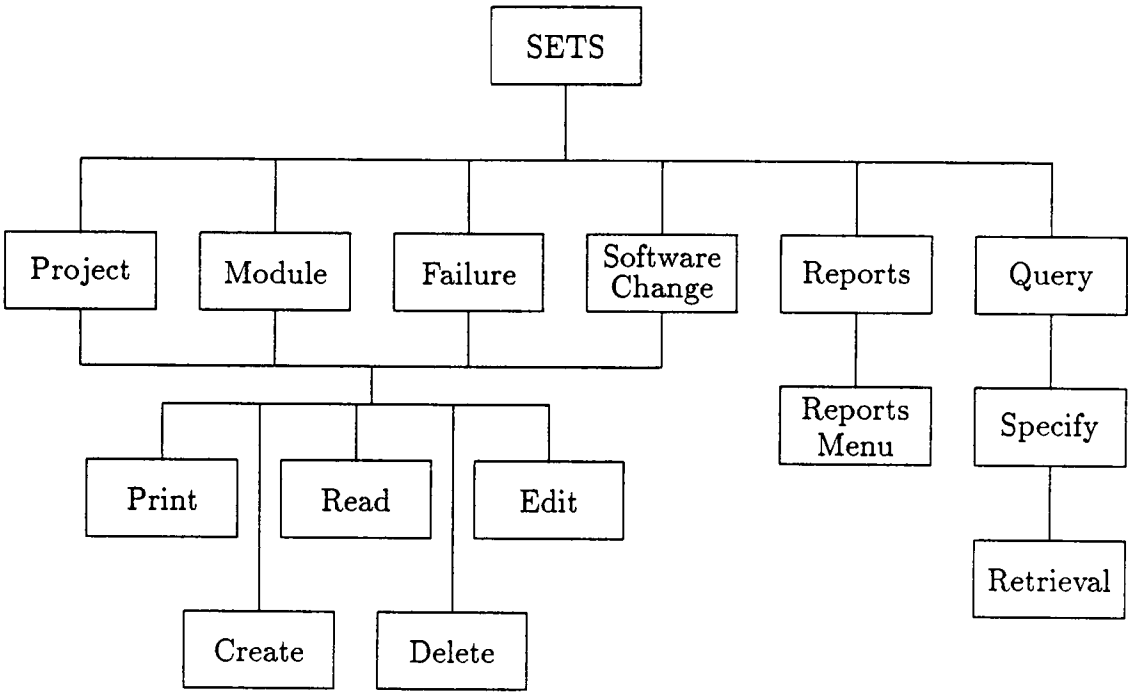

Figure 2. SETS System Overview Chart

- Dynamic. They can grow and shrink to suit the needs of the moment; the storage space they require usually shrinks with them.
- Blind. It is not necessary to know in advance how large or small they will become as they are used.
- Sparse. The potential data space may be much larger than the set of data in the list at any given time.

Linked lists allow SETS to have an unspecified number of such structures nested together.

The list data structures have a common structure. Each is a doubly-linked list with a listhead which points to the head and tail block in the list. Each structure in the list contains two self-describing fields:

1. The size (in bytes) of the data structure.

2. The data structure type code.

The size and type are defined in bytes eight through ten of the data structure, leaving the bytes zero through seven available to link the data structure into a list. Figure 3 shows the standard list structure format.

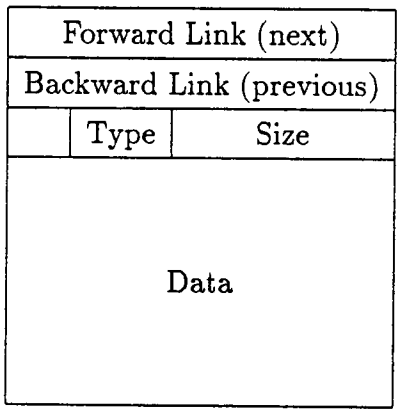

Figure 3. List Data Structure Example

Each structure is linked to a predecessor and successor structure by way of the members *next* and *previous*. The *type* field enables SETS to distinguish different data structures and to confirm that a piece of dynamic storage contains the expected data structure type.

Forms

Many interactive applications may be greatly simplified by using a forms interface. A form displayed on a terminal screen serves as the equivalent of a form on a piece of paper. A user interacts with the form by filling in the different fields on the form with data, which the application then manipulates. The application can also communicate information to users by displaying data on the form. Thus, forms may be used for both input and output. Because it displays data in a well organized manner, a form is a good user interface; the form may be designed to highlight the most important information in the display.

A forms interface offers several advantages. First, forms display simultaneously all data with which users are working. Users obtain a specific data item in context, since other data associated with the item are also displayed. Because the data items appear simultaneously, users may enter data in any order. Furthermore, if a user realizes a mistake has been made after entering a particular data item, the user can correct the mistake before completing the whole form. This contrasts with a "line-by-line" interface, in which users can only enter one item of data at a time and cannot conveniently back up to previous items.

A second advantage of a forms interface is that the form indicates to the user the data items appropriate to a command. The application can check the values filled in by the user to verify that the values are reasonable and return error messages if they are not.

Screens

A screen is composed of both a form and a menu of operations. While a form is a good device for data input and output, a user must also be able to execute commands that cause the software to perform various functions. In order to control the actions in software, users need a menu of operations. An operation may affect data on the form, data in the data base, or the form itself.

A screen is the principal vehicle for interaction between SETS and the user. Figure 4 shows an example of a screen. The form is displayed on the upper part of the screen, and the menu of operations appears on the penultimate line of the screen. A mini-help line that briefly describes the highlighted menu operation is displayed on the bottom line of the screen. The form may be used to enter, to update, and to display project data. To execute any of the operations, the users presses the MENU key (Ctrl-Z) and types an unambiguous combination of the operation's initial letters. The user can alternatively move the menu selection light bar to the desired operation by pressing the left and right arrow keys. The current menu operation is highlighted with the light bar. A brief description of the highlighted menu operation appears on the mini-help line of the screen. The highlighted menu operation may be executed by pressing the RETURN key. Appendix B displays the screens implemented in SETS.

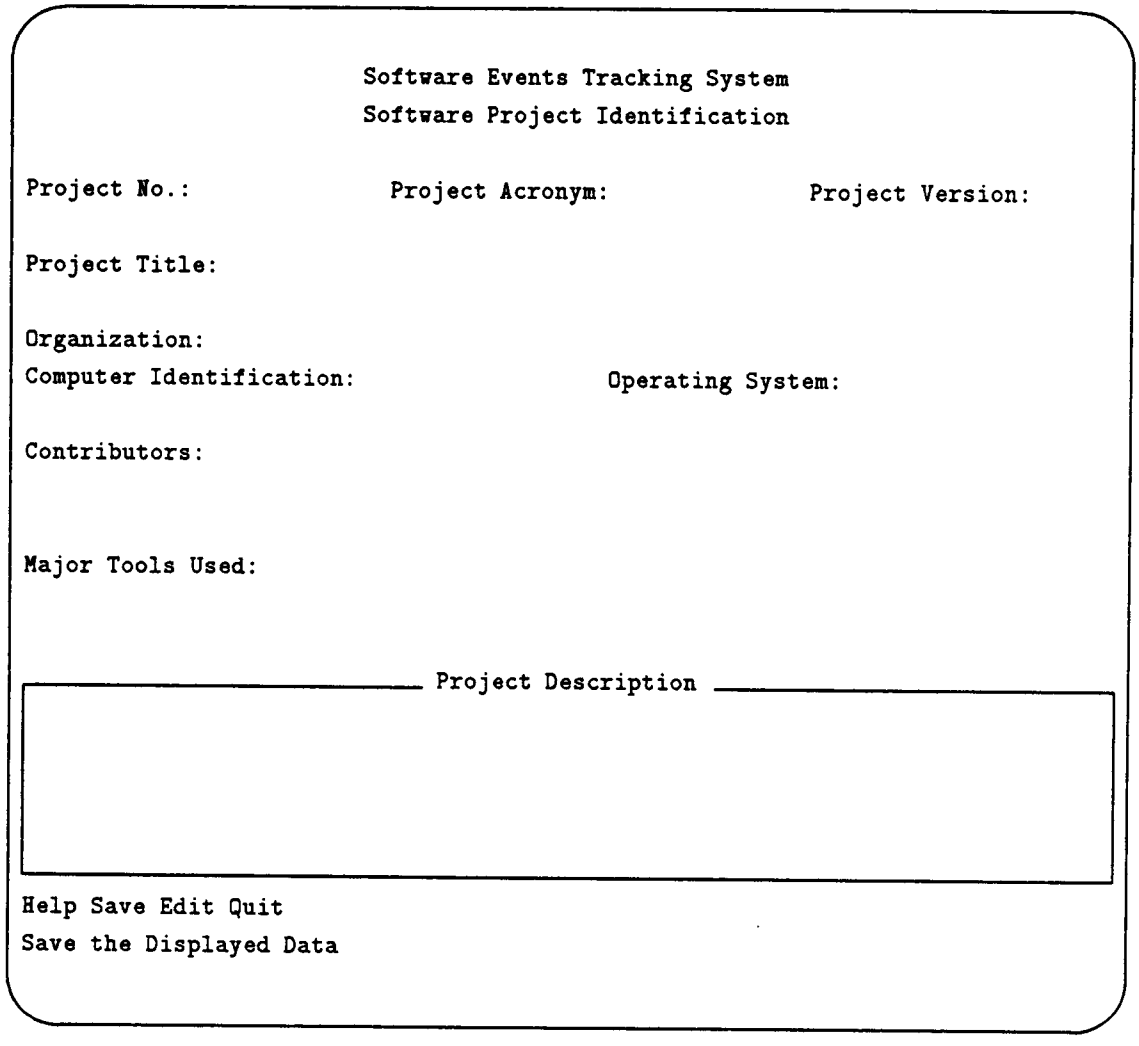

Figure 4. Screen Example

Data Presentation

When possible, SETS data are displayed in fixed regions. When there is a possibility of having more information than can fit in the available space (e.g., Free-Text Data), data are displayed in scrolled regions. Information in all fixed regions and that at the top of all scrolled regions is displayed on the screen. If a scrolled region has more data or text than can be displayed in its allotted space, the remaining information is initially hidden from view.

To view the hidden information, the desired scrolled region must be selected and the information scrolled. The currently selected scrolled region is identified by highlighting its title. A given SETS data display may have several scrolled regions, each with hidden data.

Once the desired scrolled region has been selected, its data may be scrolled up and down. The scrolling menu of operations is described in Table I:

Table I. Scrolling Menu of Operations

Operation	Meaning
Next_Page	Scroll forward, one view page
Former_Page	Scroll backward, one view page
Top	Scroll to Top of Page
Bottom	Scroll to Bottom of Page
Up Arrow (↑)	Scroll forward one line
Down Arrow (↓)	Scroll backward one line

Help

The **Help** operation is available from all the SETS screens. **Help** provides a description of each data element and each menu operation available for the current screen. This operation usually provides more information than can be displayed on a single screen. Therefore, **Help** is presented as a single, large scrolled region. All of the scrolled region commands described above are available. Selecting the **Quit** menu operation from the Help screen causes SETS to exit **Help** and return to the data display.

Printing

Printing of the current screen in its entirety, including the full contents of all scrolled regions, may be accomplished by selecting the **Print** option. After selecting the **Print** option, the Print menu of operations appears on the screen menu line. The Print menu of operations is described in Table II.

Table II. Print Menu of Operations

Operation	Meaning
Help	Get help about the Print Options
Personal_Printer	Print Data on a printer physically connected to the terminal
System_Printer	Print Data on the System Printer (SYS$PRINT)
File	Print Data to a File
Quit	Return to the Previous Menu of Operations

Terminal Input

The terminal input modules within SETS allow validation of data to be performed one character at a time as data are entered. Validation is implemented by passing a "picture" string to the terminal input modules each time a terminal input is requested. The "picture" string is a string of bytes in which each byte corresponds to one character of the field where the data are entered. Each byte specifies a legal character type for that character position. As each character is entered, it is validated and then echoed, advancing the cursor position. Invalid characters are not echoed to the screen. The validation modules will also convert entered characters if necessary. For example, if the "picture" string specifies that uppercase letters are valid, then all lowercase letters entered will be converted to uppercase. A list of validation picture string characters may be found in Table III.

The terminal input modules define a bounded set of line editing functions. These functions are available through control keys. Cursor movement is provided in single character increments (Left Arrow or Ctrl-D, Right Arrow or Ctrl-F), beginning of line (Ctrl-B) or end of line (Ctrl-E) movements. The input modules support both insert character and overstrike character modes. The Insert/Overstrike mode is set to overstrike for left justified input and to insert for right justified input at the beginning of a read operation; it may be changed dynamically with the toggle insert/overstrike key (Ctrl-A). Deletion of characters is supported in character (Ctrl-H, Backspace, or DELETE), word (Ctrl-J or Line Feed) and beginning of line (Ctrl-U) increments.

Table III. Validation Picture String Characters

Picture Character	Meaning
X	Any Character
P	Punctuation
Y	Yes / No
T	Force Uppercase
L	Uppercase Alphanumeric
U	Uppercase Alphabetic
A	Alphabetic
9	Numerical
D	Date String
Z	Time String

Free Text Interface

The development of a software system for collecting data on software events must be flexible to allow for unique project requirements. The system must be applicable to any software project, be easily adaptable to research and management needs, and be easily changed as the standards mature and change. SETS uses a free-text data-entry interface to accomplish all of these objectives. A free-text interface allows users and organizations to establish their own unique data requirements for a software project. These data may then be entered into the system with a standard text editor.

SETS implements the free-text interface by "calling" the VAX Text Processing Utility (VAXTPU) editor from within the program. When the user needs to edit

one of the free-text regions, the text data are first written to a temporary ASCII sequential file. The VAXTPU editor is then "called" from within the program and the temporary file name to be edited is specified. When the user has finished the editing session, VAXTPU creates a new version of the temporary file. The data in this file are then read into memory and later stored in the appropriate data file.

IEEE and Local Standard Text

Although SETS provides a flexible free-text interface, many organizations may wish to employ a standard vocabulary for those software events that frequently occur. The failure and software change screens (See Figure 4 and Appendix B) each have two menu options, **IEEE** and **Local**, that allow the user to select a pre-defined phrase from a list of text. The selected phrase is then inserted into the appropriate free-text data field. The **IEEE** option displays text suggested by IEEE Draft Standard P1044 [17]. This text list provides numerous expressions for describing software failures and software changes. The **Local** option displays text that has been defined by the local site. The use of these standard text lists will provide a common terminology for communication among projects and personnel.

Query

SETS provides a visually oriented, form-driven query interface that allows users to retrieve data from the data base. Ease of operation for the user is a

primary design factor for the query interface: forms on the terminal screen are used to query and to retrieve data. The user enters the query in a form and the results are printed in the same form.

The Query function has two states: the Query Specification state and the Go state. While in the Query Specification state, the user may specify a query by filling in a form with the values to be sought. After the form is filled in, SETS enters the Go state and retrieves those records that match the specified query. After retrieving the matching records, SETS returns to the Query Specification state. The second query and each succeeding queries search only those records retrieved in the preceding query.

After the user selects the "Query" menu option, SETS displays the screen presented in Figure 5. This screen displays the level of the query (i.e., number of searches performed) and the number of "hits" (i.e., number of records) located that match the specified query. From this screen the user can specify the data file to be searched.

The user specifies the data base to be queried by selecting one of the following: "Project," "Module," "Failure," or "Software_Change". When the data base has been selected, a blank form is displayed for the user to enter data. This form is the same form used for data entry. The simplest way to select data is to leave all of the fields blank. If all fields are blank, SETS assumes the user wants to retrieve all rows in the query target. To select a subset of the data, the user fills in fields with the values desired. If data are entered in some of the fields, the fields left blank are not used in the query.

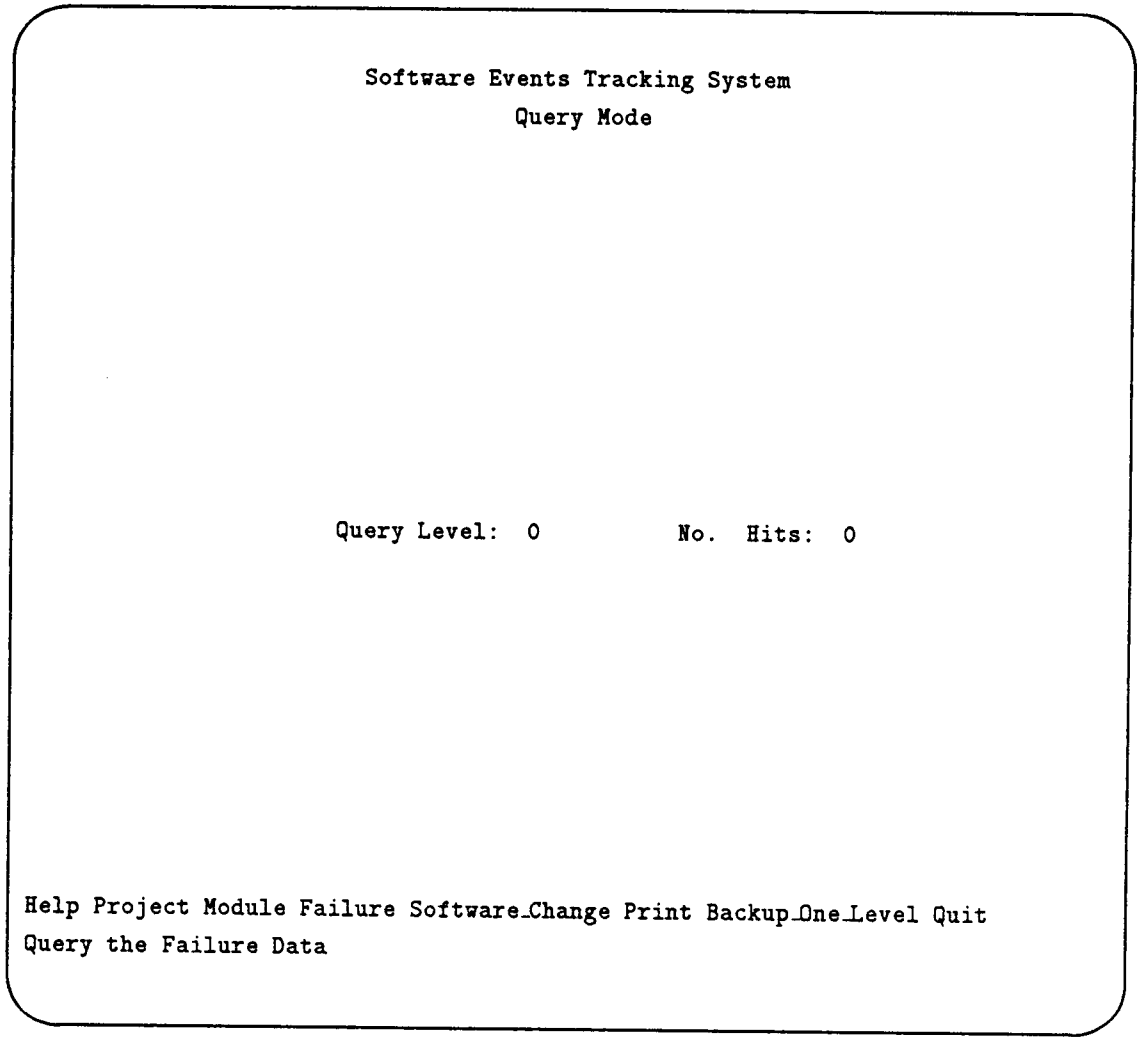

Figure 5. Query Specification Screen Example

SETS constructs a query using the following rules:

1. Each non-blank field on the form is used in the qualification.

2. The specification in the fields are joined by the logical connector "and".

When specifying a query, comparison operators may be used to retrieve more complicated sets of data. This task may be accomplished by placing a valid operator in front of the value in the field. The valid operators are described in Table IV:

Table IV. Valid Query Operators

=	Equal
-	Not Equal
<	Less Than
<=	Less Than or Equal
>	Greater Than
>=	Greater Than or Equal

Fields containing only values with no comparison operator are assumed to have an implicit Equal operator. An example of a query screen showing a valid query is displayed in Figure 6. The query specified in Figure 6 may be recast in English as "Find all failures of Category S with Originator not equal to FLOYD and date of failure greater than 1/1/88 and number of runs before the failure less than or equal to 50".

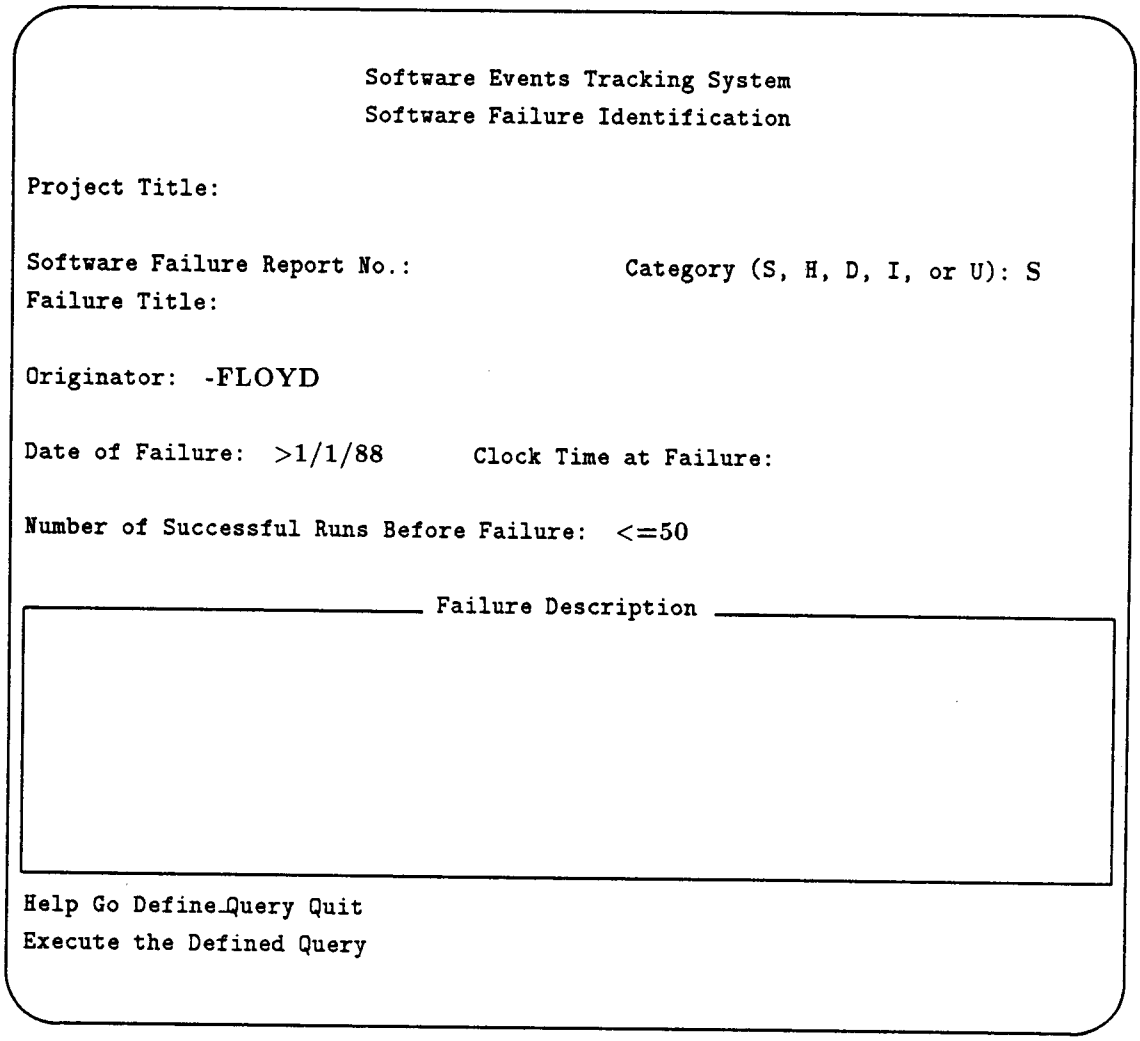

Figure 6. Query Example

After the query has been specified, the user selects the "Go" menu operation. This operation instructs SETS to retrieve the data records that match the query. After the data have been retrieved, SETS is again placed into the Query Specification state. SETS displays the number of records ("hits") located that matched the query and increments the query level by 1. If a new query is initiated, only those records that matched the previous query will be searched.

After the data have been retrieved, the user can list these data by selecting the "Print" menu operation. The retrieved data will be displayed using the form. The user can exit the **Query** option by selecting the **Quit** menu operation.

CHAPTER 5

QUALITY ANALYSIS OF SETS

This chapter presents a quality analysis of the SETS software. The methods of analysis used are:

- Logical structure metric analysis
- Composite metric analysis
- Analysis of design metrics and principles of coupling and cohesion
- Walkthrough inspection with an emphasis on reliability, defects, impurities, and maintainability issues

For detailed descriptions of these metrics, see Appendix C. Because the names of the modules in SETS cannot be uniquely identified by truncation of the module name, a unique three-character code has been assigned to each of the modules in SETS in order to reduce the space requirements for the appendices. A table of abbreviated names and corresponding module names is shown in Appendix D.

A. Logic Structure and Composite Metrics

This section presents an analysis of various software metrics associated with the logic structure of SETS. The metrics chosen for analysis are the McCabe cyclomatic complexity metric [23] and the Halstead Software Science metrics [15]. These metrics are customarily used as measures of the following aspects of program quality:

- Effort in developing software
- "Complexity" of software
- Error-proneness of software

All of the analyses are based on measurements made at the module level. For purposes of the logic structure metric analyses performed, each function (subroutine) in SETS is considered to be a module.

McCabe and Halstead Analysis of SETS

McCabe's cyclomatic complexity, $V(G)$ [23], and Halstead's software science measures [15] were computed for all modules of SETS. A table of the metrics computed for each module may be found in Appendix E. The trend of the values computed for $V(G)$ may be seen in Figure 7.

Referring to Figure 7, one can see that most modules have a complexity measure, $V(G)$, less than 10. The thirteen modules that have complexity measures greater than 10 are shown in Table V. Because of the high $V(G)$ values for these

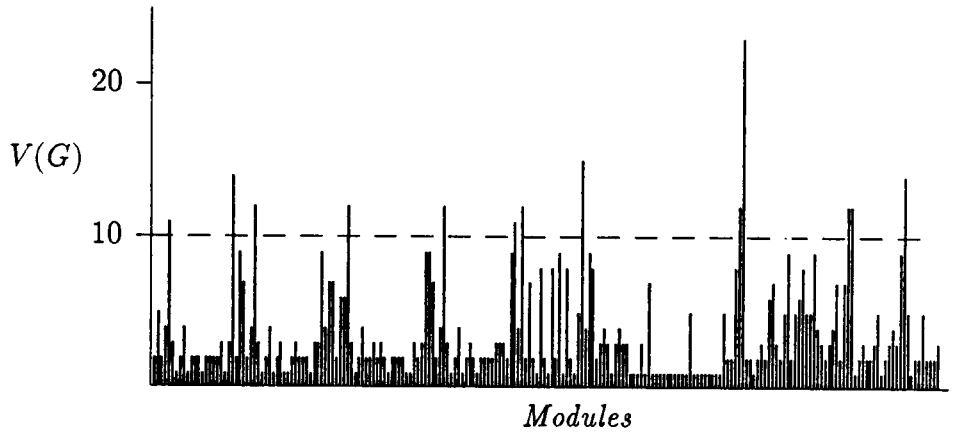

Figure 7. Cyclomatic Complexity Values for SETS Modules

Table V. Modules with Cyclomatic Complexity Greater Than 10

Module Name	$V(G)$
DEFINE_PROJECT	11
SELECT_PROJECT	14
DEFINE_MODULE	12
DEFINE_FAILURE	12
DEFINE_SCP	12
QUERY_DATA	11
PARSE_QUERY	12
WREAD_STRING	15
CHECK_PICTURE_STRING	12
CHECK_RECORD	23
MOVE_TO_NEXT_TEXT	12
MOVE_TO_PREVIOUS_TEXT	12
SCROLL_TEXT	14

thirteen modules, these modules should be reviewed with regard to function, complexity, and ease of maintenance.

B. Design Metrics

Coupling Analysis

Coupling refers to the interconnection between modules. It is important to know how modules in a program are coupled in order to maintain the program properly. If modules are strongly coupled, any modification or correction to one module may require that other modules be modified. Coupling is also an indicator of complexity, i.e., the difficulty in understanding a program; this complexity affects other aspects such as testability and reliability. As the coupling between modules becomes stronger, the complexity of the relationships between the modules increases, and the program thus becomes more complex.

Coupling analysis of SETS reveals data coupling. All 211 modules of SETS are minimally connected with each module having one entrance point and one exit point. Data are communicated between modules by the argument/parameter relationship of function calls. A list of the modules to which each module is coupled appears in Appendix F.

Cohesion Analysis of SETS

There are five logically cohesive modules, four temporally cohesive modules, seven procedurally cohesive modules, four communicationally cohesive modules, eight sequentially cohesive modules and 183 functionally cohesive modules in SETS. A list of the modules that are not functionally cohesive and their level of cohesion is shown in Table VI. Any module not listed in this table is functionally cohesive. For purposes of cohesion analysis for SETS, a module with the primary function of "calling" subordinate modules is classified as procedurally cohesive.

C. Code Walkthrough

A code walkthrough is a peer group review of a computer program. Walkthroughs have been shown to be an effective way to improve the quality of the source code of a computer program and the documents that describe the program [38]. Three experienced programmers examined the SETS code for completeness, correctness, and quality. The results of the walkthrough were:

- The SETS code is a complete, correct implementation of the design model.
- The variable names in the code provide a reasonable description of the meaning and content of the data. This increases the understandability and maintainability of the program.
- The code is neatly organized. Levels of loops are indented properly. This improves the readability of the program.

Table VI. Cohesion Analysis of SETS Modules

Module	Type of Cohesion
SETS_MAIN	Procedural
DEFINE_PROJECT	Sequential
DISPLAY_PROJECT_LOG	Sequential
FORMAT_PROJECT_DATA	Communicational
INITIALIZE_PROJECT	Temporal
PROJECT_SCREENS	Logical
DEFINE_MODULE	Sequential
DISPLAY_MODULE_LOG	Sequential
FORMAT_MODULE_DATA	Communicational
INITIALIZE_MODULE	Temporal
MODULE_SCREENS	Logical
BROWSE_FAILURE	Procedural
DEFINE_FAILURE	Sequential
DISPLAY_FAILURE_LOG	Sequential
FAILURE_SCREENS	Logical
FORMAT_FAILURE_DATA	Communicational
INITIALIZE_FAILURE	Temporal
FAILURE_SCREENS	Logical
DEFINE_SCP	Sequential
DISPLAY_SCP_LOG	Sequential
FORMAT_SCP_DATA	Communicational
INITIALIZE_SCP	Temporal
SCP_SCREENS	Logical
QUERY_DATA	Procedural
QUERY_PROJECT	Procedural
QUERY_MODULE	Procedural
QUERY_FAILURE	Procedural
QUERY_SCP	Procedural

- The code is written so that it is easily understandable by other programmers. Two of the walkthrough participants had no previous experience with the C programming language, yet had no trouble reading and understanding the code.
- The code is well documented.
- The code is written using structured constructs.
- Two constructs were identified for improvement to the code.

D. Summary of the Analysis

The suggestions and recommendations summarized here are intended to provide a framework for any possible actions that might be taken as a result of the quality analysis:

- Modules with high metric values (see Table V and Appendix E) should be reviewed with regard to function, complexity, and ease of maintenance.
- Modules that are not functionally cohesive should be reviewed with regard to their function.
- The impurities identified during the structured walkthrough should be removed.

CHAPTER 6

SETS: THE PRODUCTION SYSTEM

A. User Acceptance

A computer system cannot be a true success without strong user support. Only if the user supports the system can a workable implementation plan be constructed. Total commitment to following the implementation plan must be present. The resources necessary to carry out the plan must also be available.

SETS is presently being used within the Operations Analysis and Planning (OA&P) Division of the Oak Ridge Gaseous Diffusion Plant to track significant events for several large software projects. The general concepts of SETS have been well accepted by the users. These concepts are:

- Free text interface allows externally controlled definition of data requirements.
- Text data are entered using an established editor with which the users are familiar.
- Ability to trace project history.
- Useful retrieval of data along pre-defined paths.
- Data available for trend or other analyses.

Some of the applications of SETS within OA&P are:

- Maintain a count of failure frequencies by type (Input/Output, syntax, etc.).
- Attempt to correlate errors by the symptoms of the exhibited failures.
- Monitor the reliability of software projects over time.
- Help users to decide when a module should be rewritten instead of modified.

B. SETS in the Data Base World

SETS is an effective method for collecting data on software events. SETS gives the user much freedom in selecting the data to be stored. Due to its indexed file structures, SETS allows retrieval of selected data without searching the entire data collection. The majority of data can be retrieved with one access.

The concepts used in SETS are very similar to the processes that would be needed to develop SETS using commercially available data base management systems. Many advantages are obtained from using a commercially available data base management system. A commercial package has associated with it a vendor support staff constantly devoted to upgrading the data base management package. Therefore, better storage methods are developed, better locking mechanisms exist leading to better data security, better data reorganization methods exist, and compatible on-line query and report systems can be developed.

SETS, however, was developed with the concept of hardware and software independence. SETS was written in the C langauge, which can be ported easily to

various hardware configurations. SETS is a stand-alone system that is not dependent on the presence of other software. SETS also provides a visually oriented, form-driven query interface that allows users to retrieve data easily from the data base without having to master a cryptic query language.

C. Summary

Collecting the data described in IEEE Draft Standard P1044 can provide valuable information to software developers. Collecting software and documentation failure data can lead to the identification of the particular life-cycle phase in a project in which most problems are introduced. This identification can be helpful in making decisions concerning which tools, techniques, or methodologies need implementing or changing. Failure data can also assist in the evaluation of reliability and productivity metrics.

Improved reliability, the end objective, may be achieved with an accurate history of errors and failures. An analysis of failures and the errors that cause them provides a basis for improving the development and support processes of computer software. At the same time, this history provides a performance projection of future systems to be developed with these processes. The accuracy of this projection and the success of process improvements is directly related to the accuracy of the record of errors and failures.

LIST OF REFERENCES

LIST OF REFERENCES

1. A. A. Abdel-Ghaly, P. Y. Chan, B. Littlewood, "Evaluation of Competing Software Reliability Predictions," *IEEE Transactions on Software Engineering* SE-12, No. 9, 950 (1986).

2. L. Binder, C. Cobb, E. Galbraith, and D. Russell, *Quality Analysis Report 3 by the Software Quality Seminar*, CS 6910 (1987).

3. B. W. Boehm, G. R. Brown, M. Lipow, "Quantitative Evaluation of Software Quality," *Proceedings Second International Conference on Software Engineering*, 592 (1976).

4. C. K. Cobb, "Selecting A Software Reliability Model Based Upon Failure Data Characteristics," Master's Thesis, University of Tennessee, Knoxville (1988).

5. P. A. Currit, M. Dyer, H. D. Mills, "Certifying the Reliability of Software," *IEEE Transactions on Software Engineering* SE-12, No. 1, 3 (1986).

6. C. G. Dale and L. N. Harris, "Approach to Software Reliability Prediction," *Proceedings of the 1982 Annual Reliability and Maintainability Symposium*, 167 (1982).

7. K. B. DeGreen, *Systems Psychology*, McGraw-Hill Book Co., New York (1970).

8. L. M. Duvall, "Baseline Software Data System, System Description," RADC-TR-79-185 (1979).

9. A. Endres, "An Analysis of Errors and Their Causes in Systems Programs," *Proceedings of International Conference on Reliable Software*, 327 (1975).

10. D. L. Fish, M. T. Matsumoto, "Software Data Baseline Analysis," RADC-TR-79-67 (1979).

11. R. L. Glass, *Software Reliability Guidebook*, Prentice-Hall, Inc., New York (1979).

12. A. L. Goel, "A Summary of Discussion on 'An Analysis of Competing Software Reliability Models,'" *IEEE Transactions on Software Engineering* SE-6, No. 5, 501 (1980).

13. A. L. Goel, "Software Reliability Models: Assumptions, Limitations, and Applicability," *IEEE Transactions on Software Engineering* Vol. SE-11, No. 12, 1411 (1985).

14. A. M. Goel, G. B. Soenjoto, "Hardware-Software Availability: A Cost Based Trade-off Study," *Proceedings of the 1983 Annual Reliability and Maintainability Symposium*, 303 (1983).

15. M. Halstead, *Elements of Software Science*, Elsevier–North Holland, Inc., New York (1977).

16. H. Hecht, "Measurement, Estimation and Prediction of Software Reliability," *Software Engineering Techniques*, Infotech, (1977).

17. IEEE Std P1044, "A Standard Classification for Software Errors, Faults, and Failures," (1987).

18. IEEE Std P982, "A Standard for Measures to Produce Reliable Software," (1987).

19. Z. Jelinski, P. B. Moranda, "Software Reliability Research," in *Statistical Computer Performance Evaluation*, W. Freiberger, Ed., Academic, New York (1972).

20. D. G. Kafura, S. M. Henry, and K. Harris, "On the Relationship Among Three Software Metrics," *Performance Evaluation Review* 10, No. 1, 81 (1981).

21. R. C. Linger, H. D. Mills, and B. J. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, Reading, MA (1979).

22. B. Littlewood, "Stochastic Reliability-Growth: A Model for Fault-Removal in Computer-Programs and Hardware-Designs," *IEEE Transactions on Reliability* R-30, No. 4, 313 (1981).

23. T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering* SE-2, No. 4, 308 (1976).

24. P. N. Misra, "Software Reliability Analysis," *IBM Systems Journal* 22, No. 3, 262 (1983).

25. G. D. Musa, "A Theory of Software Reliability and Its Application," *IEEE Transactions of Software Engineering* SE-1, No. 3, 312 (1975).

26. D. L. Parnas, "The Influence of Software Structure on Reliability," *Proceedings of the 1975 International Conference on Reliable Software*, 385 (1975).

27. R. Rubey, "Quantitative Aspects of Software Validation," *Proceedings of International Conference on Reliable Software*, 246 (1975).

28. V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support," *IEEE Transactions on Software Engineering* SE-9, No. 2, 155 (1983).

29. M. L. Shooman, "Software Reliability: A Historical Perspective," *IEEE Transactions on Reliability* R-33, No. 1, 48 (1984).

30. M. L. Shooman, "Software Reliability: Measurement and Models," *Proceedings of the 1975 Annual Reliability and Maintainability Symposium*, 485 (1975).

31. M. L. Shooman, A. K. Trivedi, "Error Data Collection in Software Systems," RADC-TR-169 (1975).

32. U. Sumita, Y. Masuda, "Analysis of Software Availability/ Reliability Under the Influence of Hardware Failures," *IEEE Transactions on Software Engineering* SE-12, No. 1, 32 (1986).

33. J. D. Ullman, *Principles of Database Systems*, Computer Science Press, Rockville, MD (1982).

34. VAX C User's Guide, Digital Equipment Corporation (1987).

35. VAX Text Processing Utility User's Guide, Digital Equipment Corporation (1987).

36. R. T. Yeh, "Software Engineering," *Computer* 7, No. 4, 15 (1975).

37. E. Yourdon and L. Constantine, *Structured Design*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1979).

38. E. Yourdon, *Structured Walkthroughs*, Yourdon Press, New York (1985).

APPENDICES

APPENDIX A

Overview of SETS Data Base Files

Name: PROJECT.DAT
Description: The non-text data for a software project.
Unique Key: Project_Number

Project_Number	Character	10	Project number
Title	Character	60	Project title
Version	Character	10	Project version number
Acronym	Character	10	Project acronym
Organization	Character	60	Organization for whom developed
Computer	Character	20	Computer identification
Operating_System	Character	20	Operating system identification

Name: CONTRIBUTORS.DAT
Description: The contributors to a software project.
Unique Key: Project_Number + Line_Number

Project_Number	Character	10	Project Number
Line_Number	Integer	2	Line Number
Contributor	Character	20	One contributor to the project

Name: TOOLS.DAT
Description: The text data describing the tools used in the software project.
Unique Key: Project_Number + Line_Number

Project_Number	Character	10	Project number
Line_Number	Integer	2	Line number
Tool	Character	76	One tool used in the project

Name: PROJECT_DESCRIPTION.DAT
Description: The text data describing the software project.
Unique Key: Project_Number + Line_Number

Project_Number	Character	10	Project number
Line_Number	Integer	2	Line number
Text	Character	76	One line of text.

Name: MODULE.DAT
Description: All non-text data for a software module.
Unique Key: Project_Number + Module_Name

Project_Number	Character	10	Project number
Module_Name	Character	10	Module name
Abstract	Character	70	Module abstract
Date_Created	Character	8	Date module was created
Language	Character	20	Language used
Compiler	Character	20	Compiler identification

Name: MODULE_DESCRIPTION.DAT
Description: The text data describing the software module.
Unique Key: Project_Number + Module_Name + Line_Number

Project_Number	Character	10	Project number
Module_Name	Character	10	Module name
Line_Number	Integer	2	Line number
Text	Character	76	One line of text.

Name: AUTHORS.DAT
Description: The authors of a software module.
Unique Key: Project_Number + Module_Name + Line_Number

Project_Number	Character	10	Project number
Module_Name	Character	31	Module name
Line_Number	Integer	2	Line number
Author	Character	20	One author of the module

Name: FAILURE.DAT
Description: All Non-Text Data for a Software Failure.
Unique Key: Project_Number + Failure_Number

Project_Number	Character	10	Project number
Failure_Number	Character	10	Failure number
Category	Character	1	Failure category
Originator	Character	40	Name of person creating failure report
Telephone_Number	Character	10	Telephone number of originator
Date_of_Failure	Character	6	Date failure occurred
Time_of_Failure	Character	5	Time failure occurred
Number_Runs	Integer	4	No. of successful runs before failure
Analyst_Name	Character	40	Name of person analyzing failure
Analysis_Time	Float	4	Time required to analyze failure
Disposition	Character	1	Disposition of failure report

Name: FAILURE_DESCRIPTION.DAT
Description: The text data describing the nature of the failure.
Unique Key: Project_Number + Failure_Number + Line_Number

Project_Number	Character	10	Project number
Failure_Number	Character	10	Failure number
Line_Number	Integer	2	Line number
Text	Character	76	One line of text.

Name: RESPONSIBLE_MODULES.DAT
Description: The modules responsible for the software failure.
Unique Key: Project_Number + Failure_Number + Line_Number

Project_Number	Character	10	Project number
Failure_Number	Character	10	Failure number
Line_Number	Integer	2	Line number
Module_Name	Character	30	Module responsible for failure

Name: ACTUAL_CAUSE_OF_FAILURE.DAT
Description: The text data describing the actual cause of the failure.
Unique Key: Project_Number + Failure_Number + Line_Number

Project_Number	Character	10	Project number
Failure_Number	Character	10	Failure number
Line_Number	Integer	2	Line number
Text	Character	76	One line of text.

Name: TESTING_TO_VERIFY_FIX.DAT
Description: The text data describing the testing required to verify a software fix.
Unique Key: Project_Number + Failure_Number + Line_Number

Project_Number	Character	10	Project number
Failure_Number	Character	10	Failure number
Line_Number	Integer	2	Line number
Text	Character	76	One line of text.

Name: SCP.DAT
Description: The non-text data for a software change proposal.
Unique Key: Project_Number + SCP_Number

Project_Number	Character	10	Project number
SCP_Number	Character	10	Software change proposal number
SCP_Title	Character	75	Title of software change proposal
Originator	Character	30	Name of person preparing SCP
Date_Prepared	Character	8	Date SCP was prepared

Name: NEED_FOR_SCP.DAT
Description: The text data describing the need for a software change or software enhancement.
Unique Key: Project_Number + SCP_Number + Line_Number

Project_Number	Character	10	Project number
SCP_Number	Character	10	Software change proposal number
Line_Number	Integer	2	Line number
Text	Character	76	One line of text.

Name: IMPLEMENTED_SCP.DAT
Description: The text data describing the software change or software enhancement that was implemented.
Unique Key: Project_Number + SCP_Number + Line_Number

Project_Number	Character	10	Project number
SCP_Number	Character	10	Software change proposal number
Line_Number	Integer	2	Line number
Text	Character	76	One line of text.

Name: FORMS.DAT
Description: The screen forms used by SETS.
Unique Key: Form_Name

Form_Name	Character	30	Form name
Record_Type	Integer	4	Type of form record
Start_Row	Integer	4	Start row of display
Start_Column	Integer	4	Start column of display
Video	Integer	4	Video attributes of display
End_Row	Integer	4	Start row of display
End_Column	Integer	4	Start column of display
Text	Character	80	Text to be displayed

Name: HELP.DAT
Description: The help text used by SETS.
Unique Key: Help_Name + Line_Number

Help_Name	Character	10	Help name
Line_Number	Integer	2	Line number
Text	Character	76	One line of help text.

APPENDIX B

SETS Screens

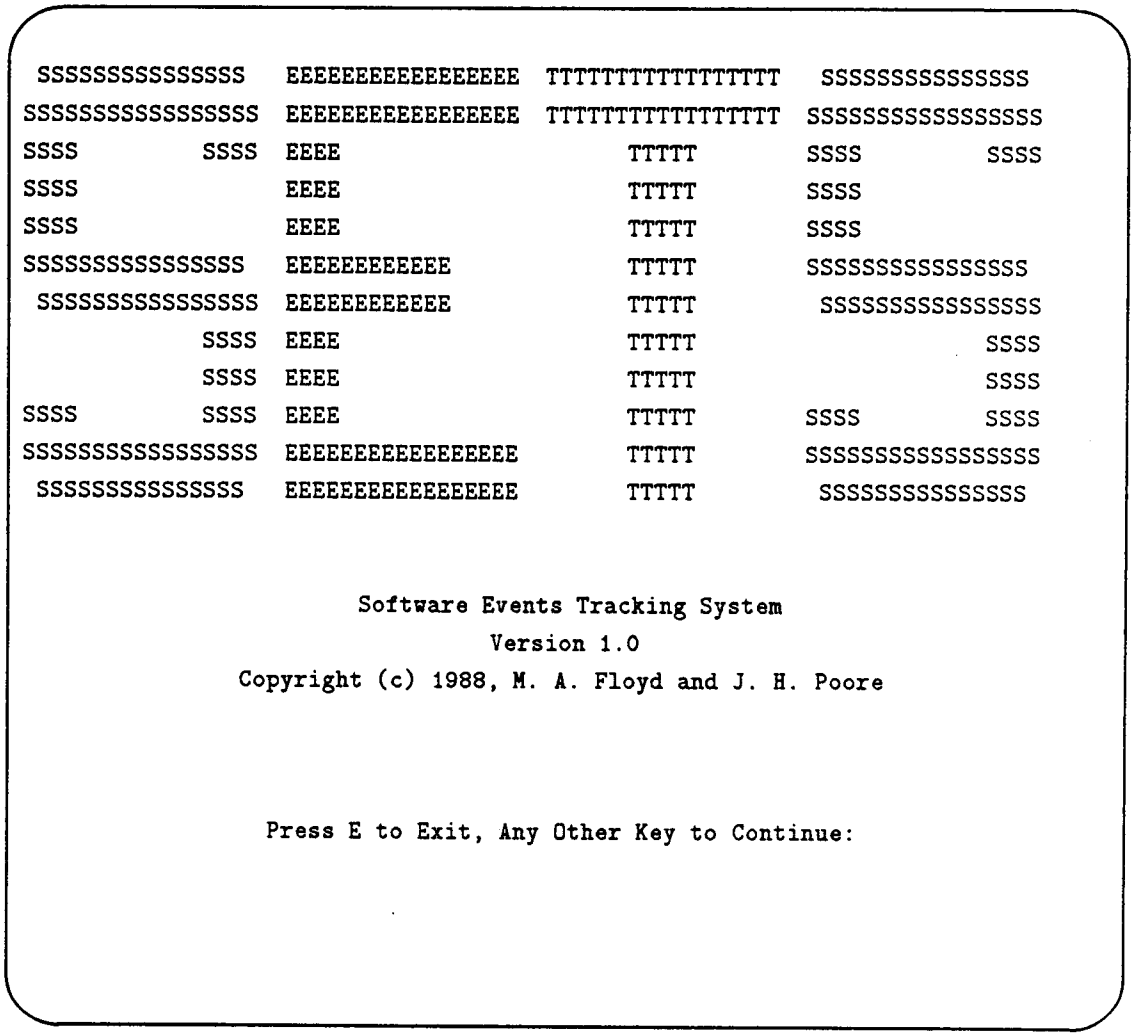

Figure 8. SETS Logo Screen

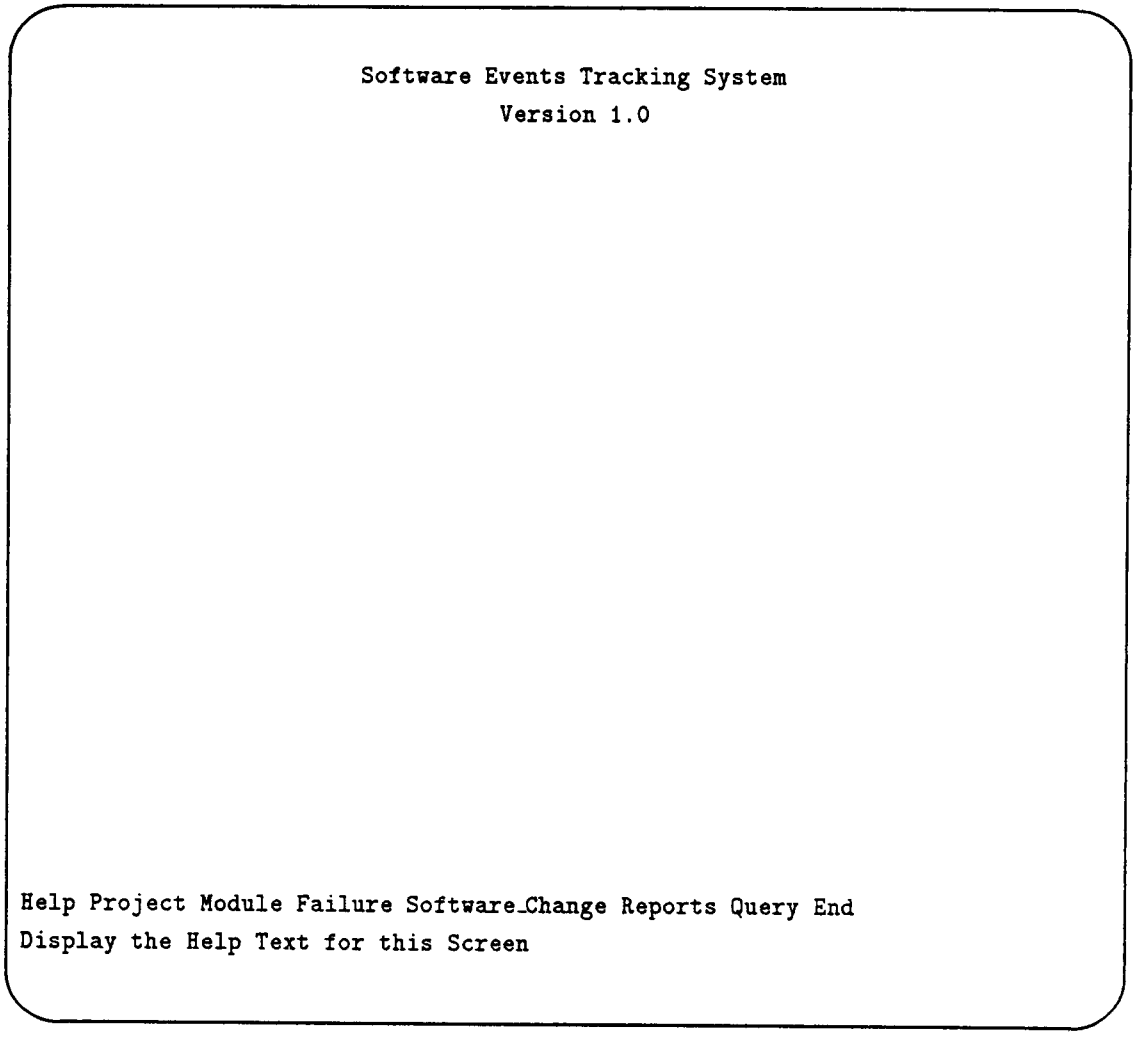

Figure 9. SETS Main Menu

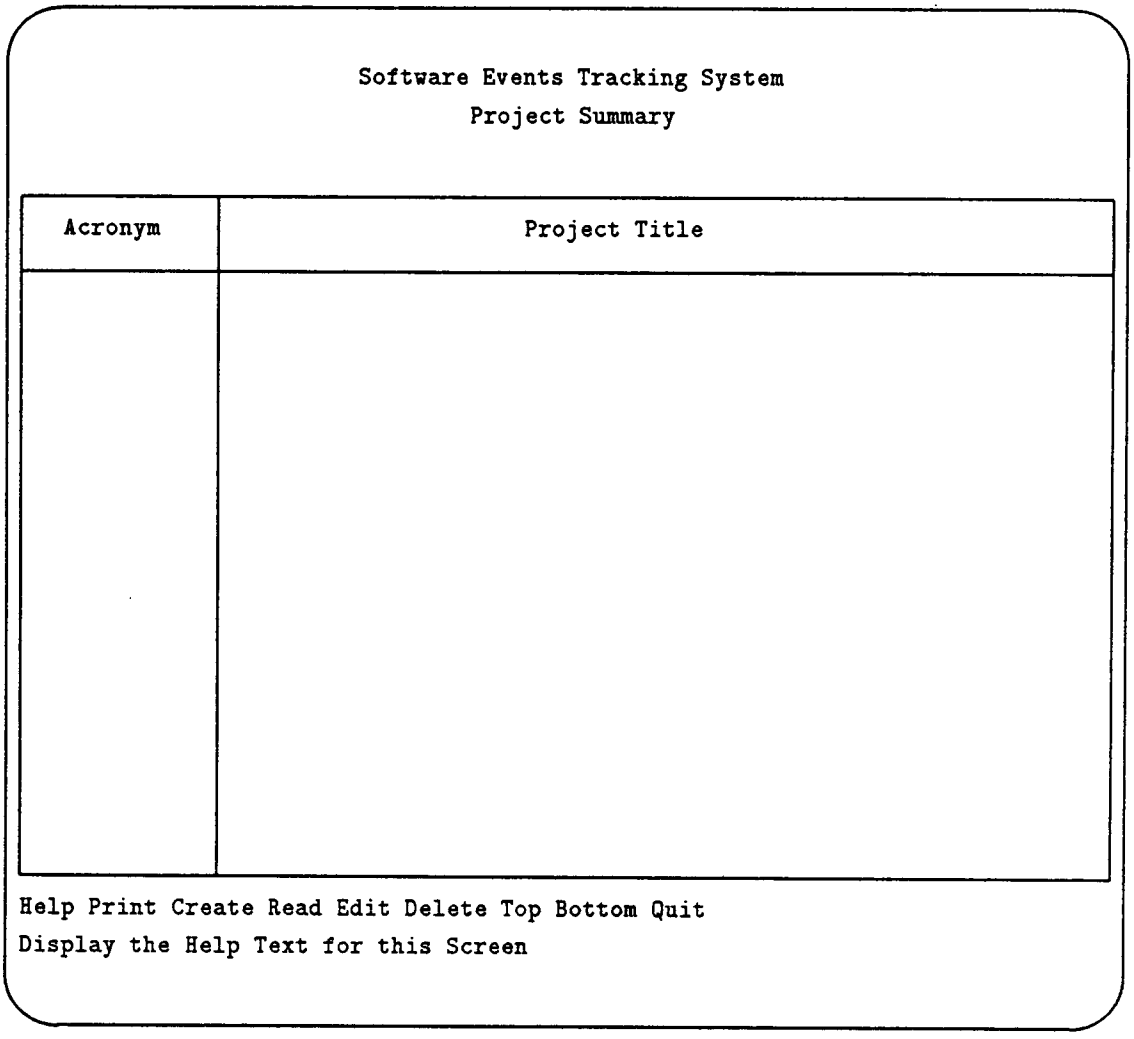

Figure 10. Project Summary Screen

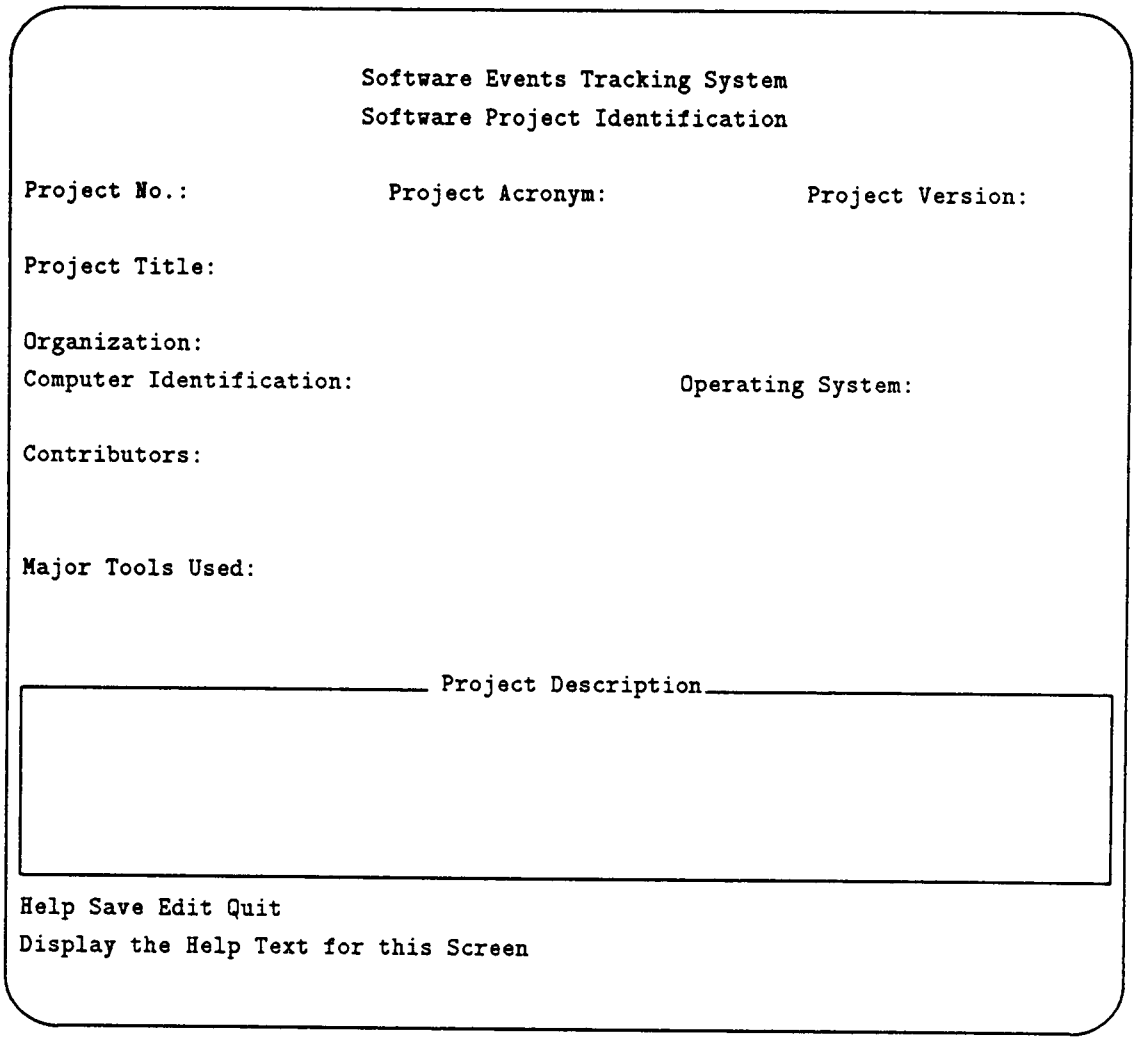

Figure 11. Project Screen

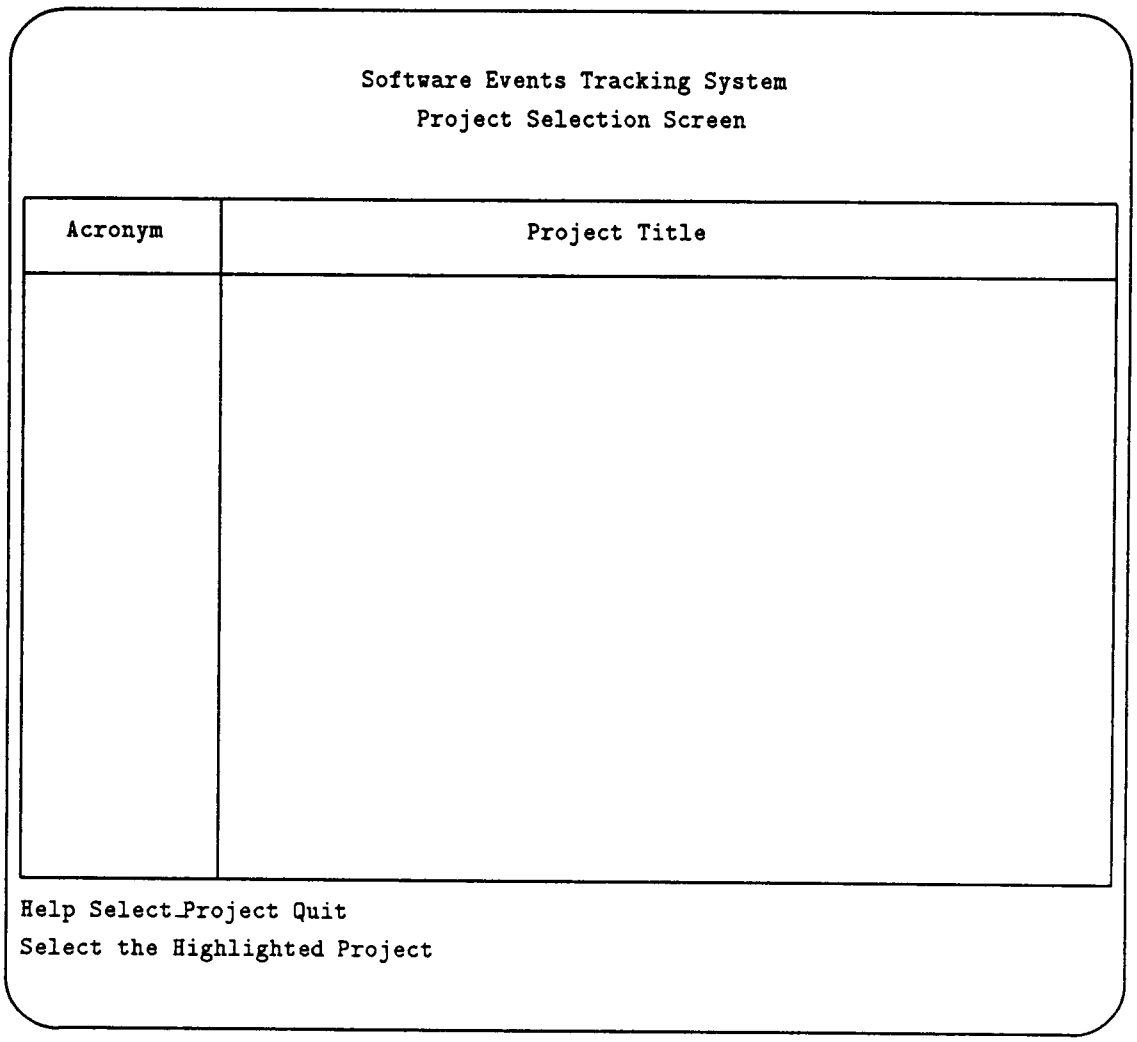

Figure 12. Project Selection Screen

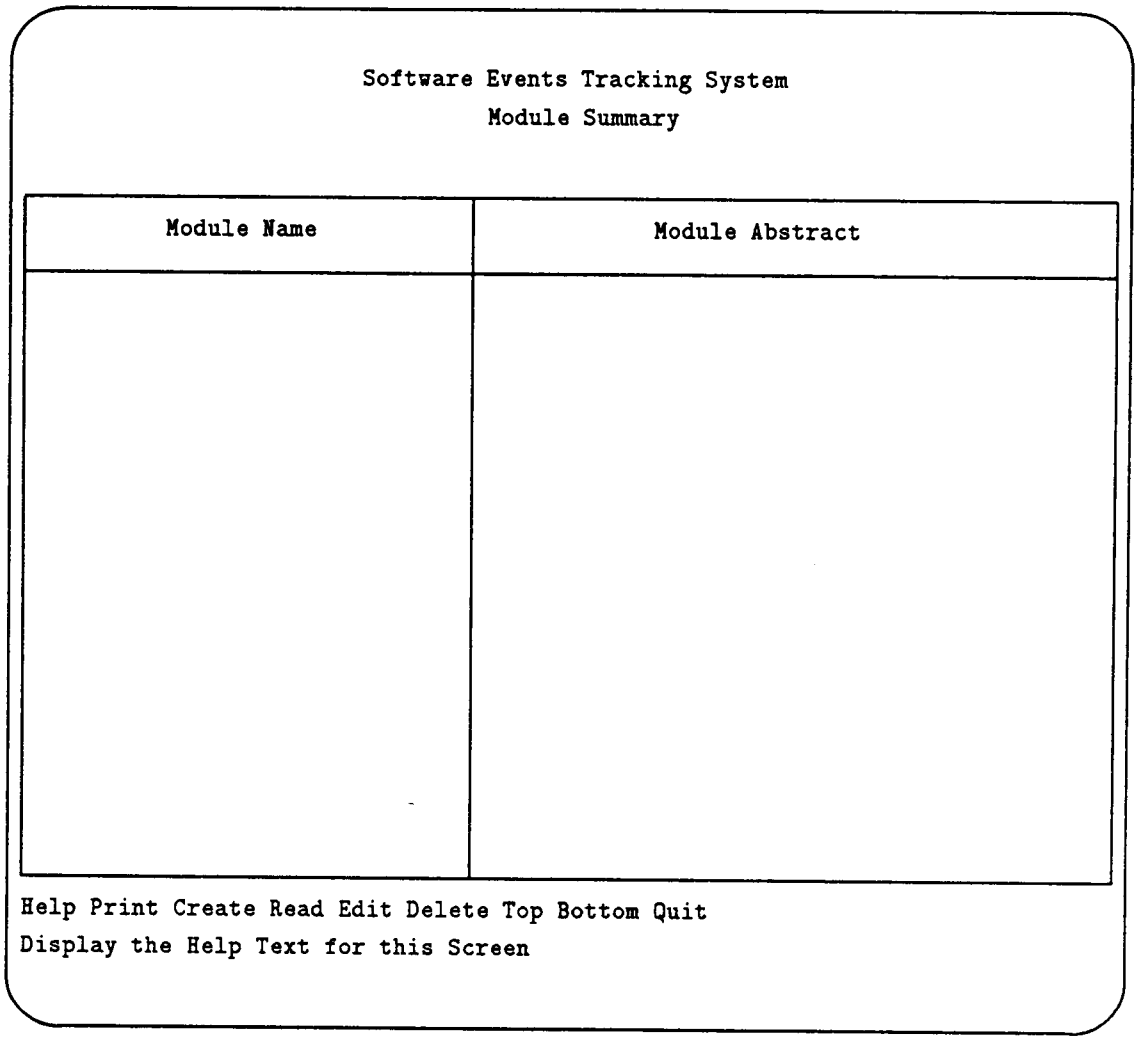

Figure 13. Module Summary Screen

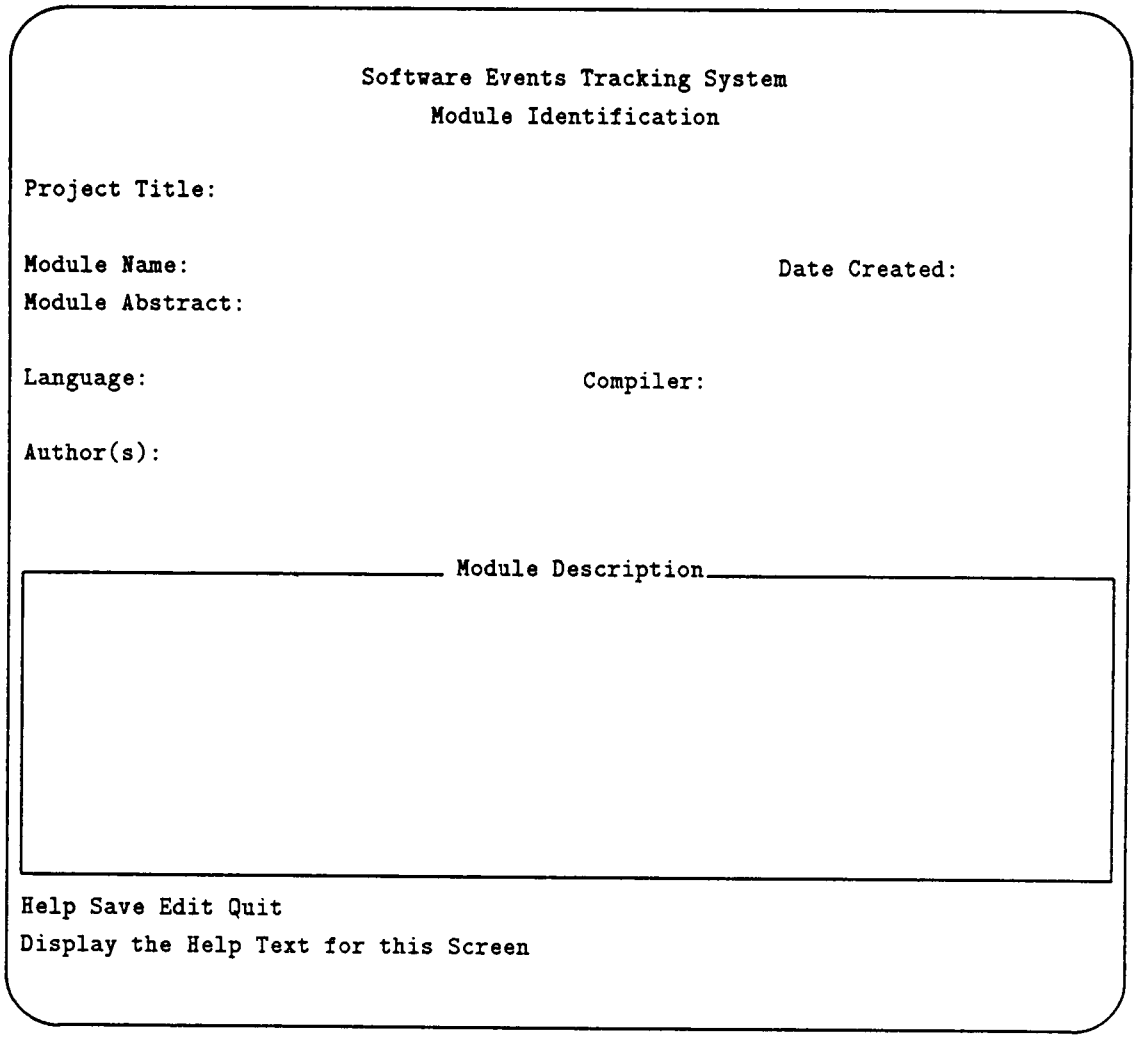

Figure 14. Module Screen

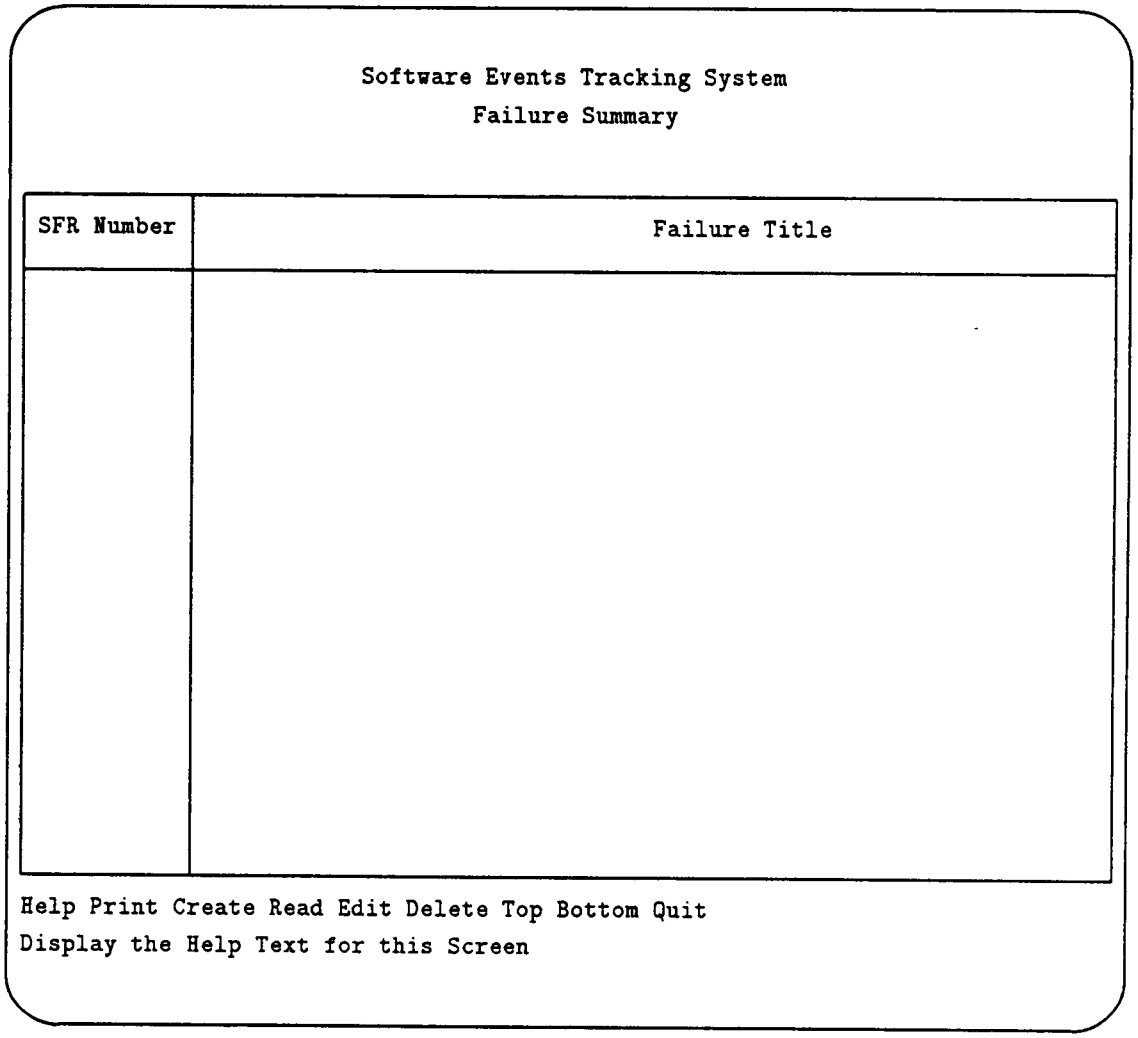

Figure 15. Failure Summary Screen

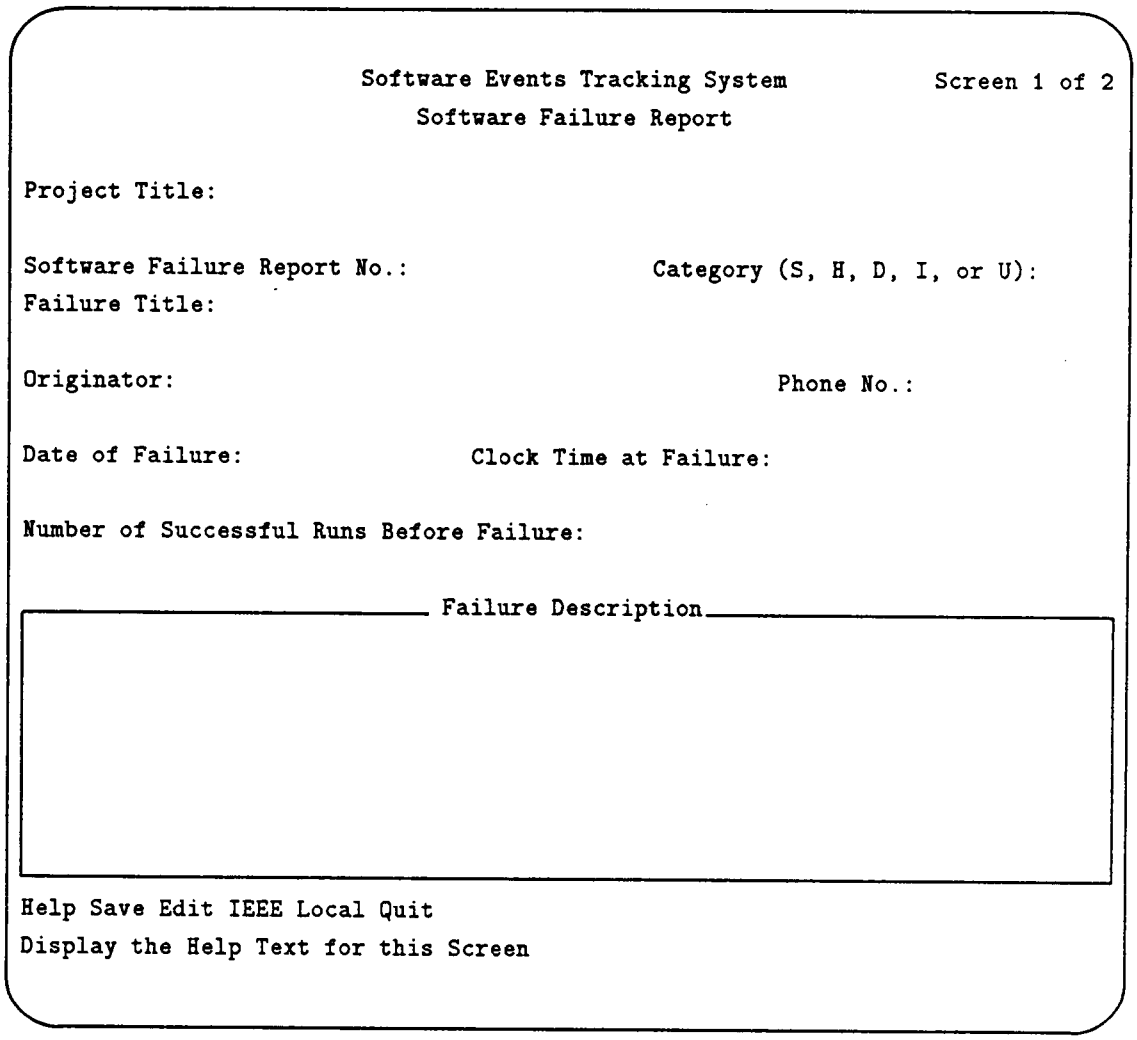

Figure 16. Failure Screen 1

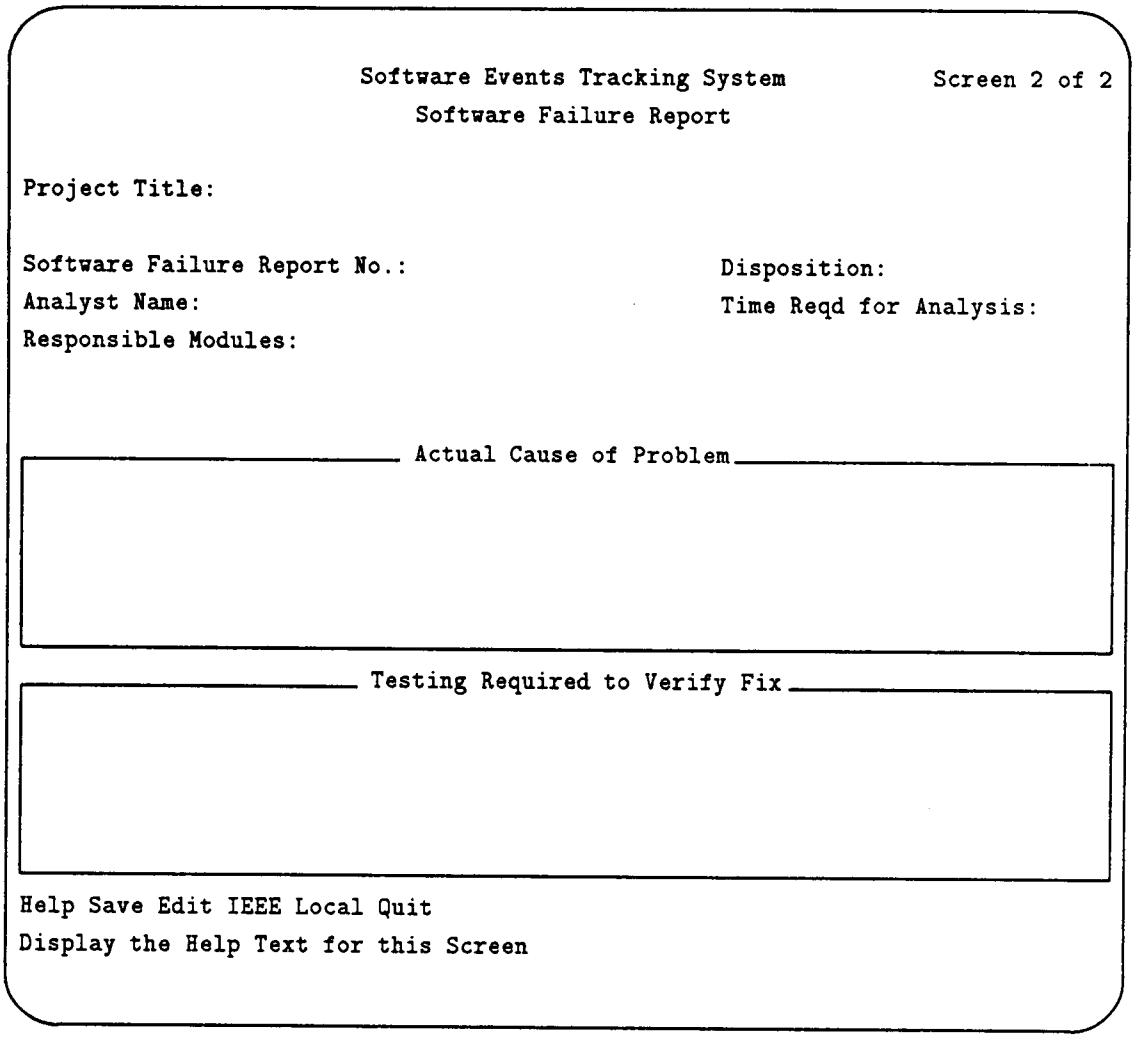

Figure 17. Failure Screen 2

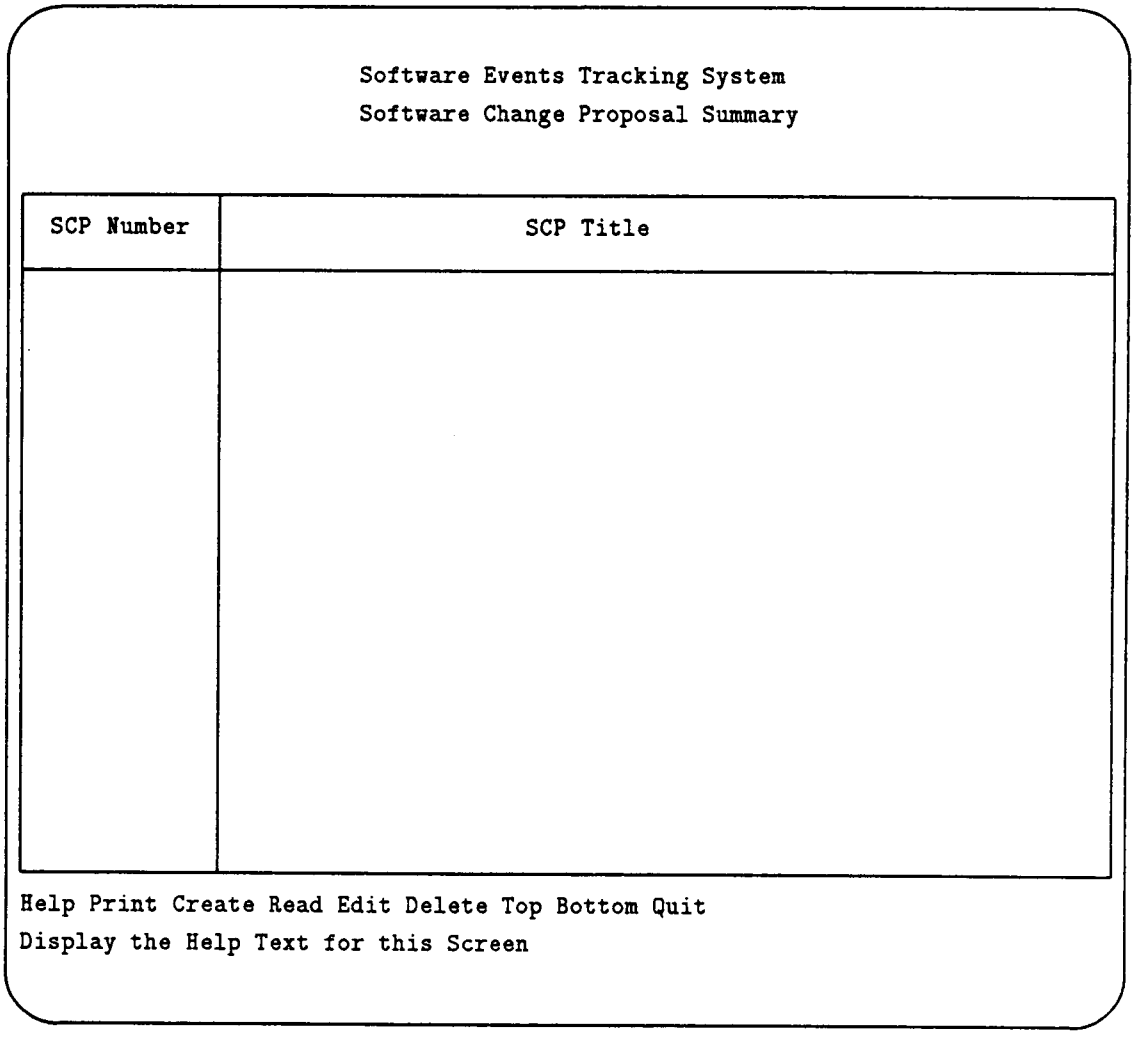

Figure 18. Software Change Summary Screen

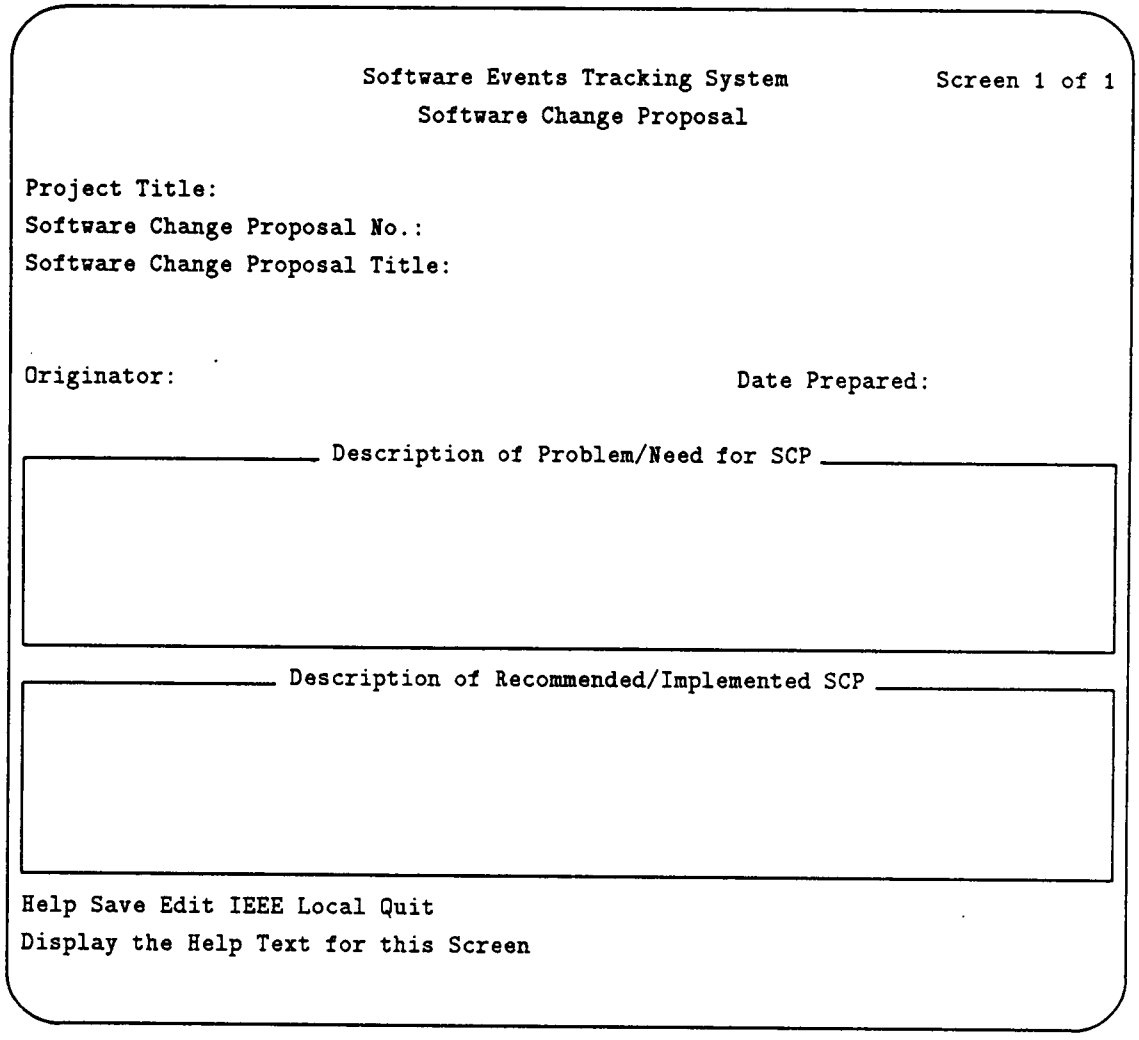

Figure 19. Software Change Screen

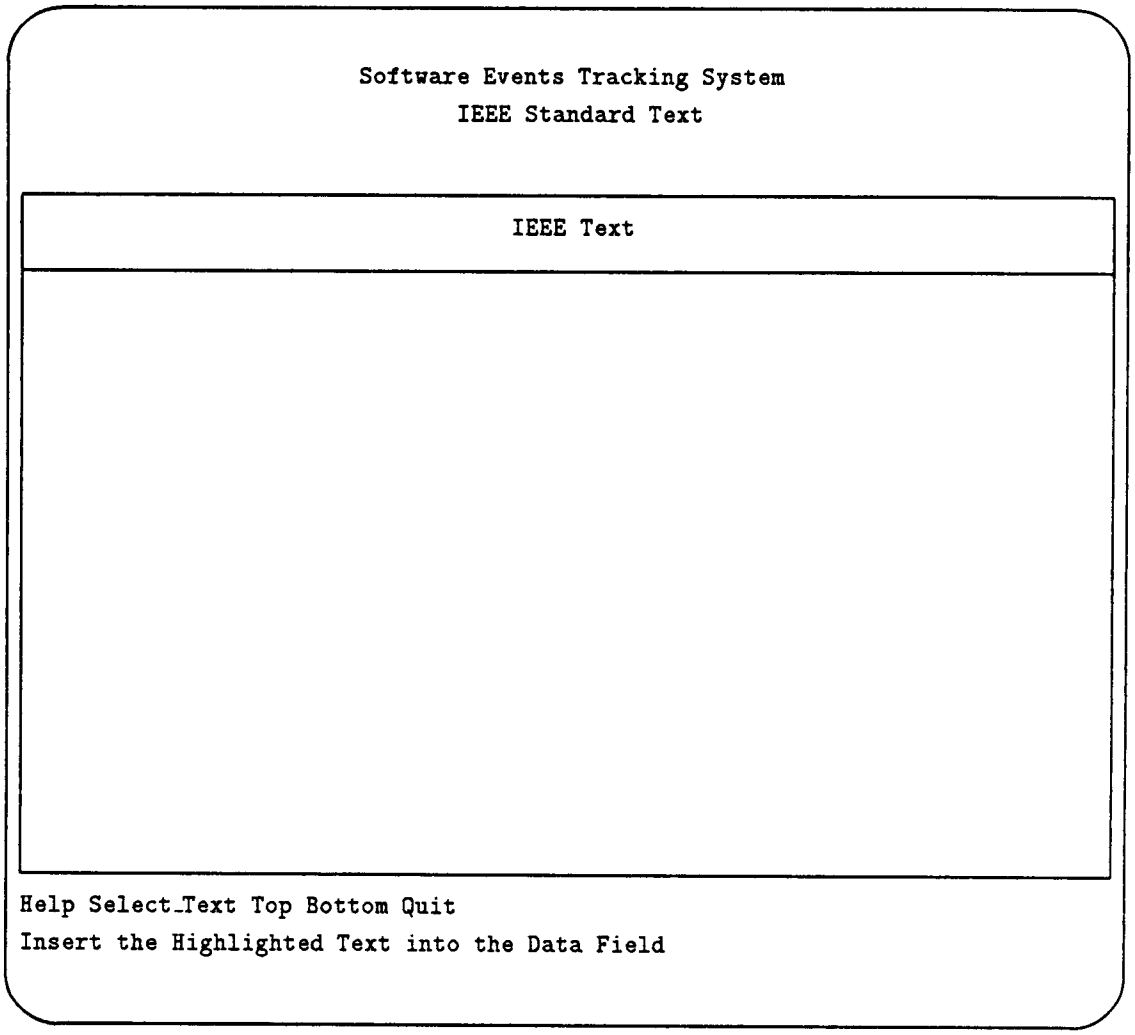

Figure 20. IEEE Standard Text Screen

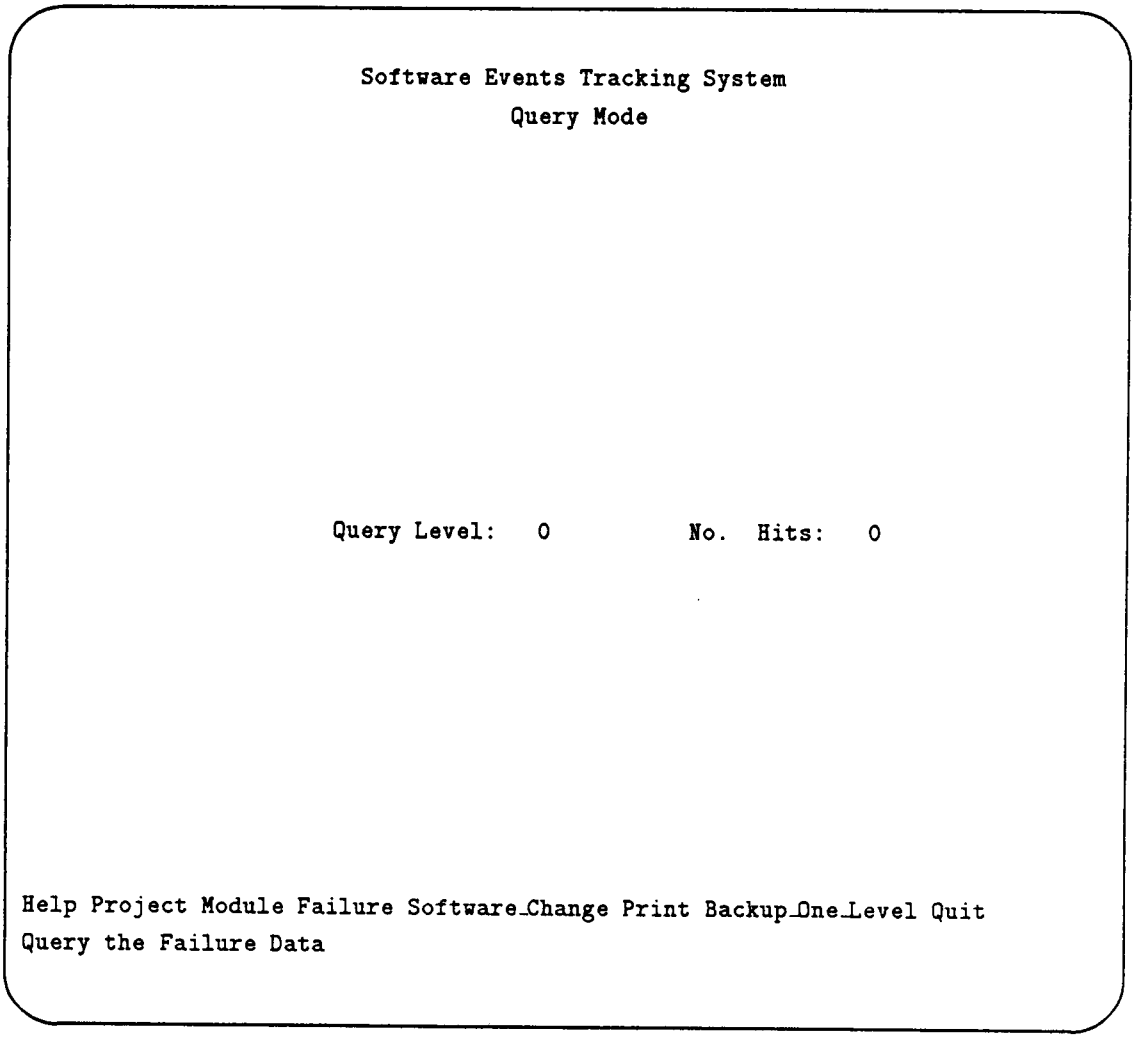

Figure 21. Query Specification Screen

APPENDIX C

Descriptions of Metrics Used in the Evaluation

This appendix borrows heavily from a report for CS6910 by Binder, Cobb, Galbraith and Russell [2]. The material is used by permission of the instructor.

A. McCabe's Cyclomatic Complexity

McCabe has described a complexity measure that is a function of the decision structure in a module. The cyclomatic complexity $V(G)$ of a module is defined as $M + 1$, where M is the number of decision points in the module. McCabe has demonstrated that $V(G)$ measures the number of paths through a module. An execution of the module may be expressed as a sequence of these paths [20].

McCabe suggests a maximum module complexity of 10. This is not, however, a magic number: a module with $V(G)$ greater than 10 may not be of poor quality, but it should be reviewed in order to determine if it is prone to errors. McCabe has pointed out that a module should probably not be rewritten simply for the sake of reducing its complexity. The cyclomatic complexity may be used, however, as an indicator that further testing is necessary. Since each path represents a fundamental element of the module, a reduction in the number of corresponding paths reduces the potential for errors occurring in the module [20].

B. Halstead's Software Science

Halstead's Software Science metrics are based on the number of unique operators and operands ($\eta_1$and η_2, respectively) and the total number of operators and operands ($N_1$and N_2, respectively) in a module. Generally, any symbol or

keyword in a module that specifies an algorithmic action is considered an operator, and a symbol used to represent data is considered an operand [28]. The vocabulary of a module is then defined as the sum of the operators and operands:

$$\eta = \eta_1 + \eta_2$$

and the length of a module is defined as the cumulative occurrence of operators and operands,

$$N = N_1 + N_2$$

A basic hypothesis of Software Science is that the length, estimated by $\hat{N}$, of a well-structured module is a function only of $\eta_1$and η_2[28]. The length estimator is defined as

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

Halstead defines the volume V of a module as

$$V = N \log_2 \eta$$

V is the total number of operators and operands multiplied by the number of bits necessary to represent a vocabulary of size η [20]. The program level L of a module is defined as

$$L = \frac{V}{v^*}$$

where v^* is the minimum possible representation of the module. L ranges between zero and one. If L is one, then the module is written at the highest possible level and has the minimum possible size. The program level estimator $\hat{L}$ is defined as

$$\hat{L} = \frac{2\eta_2}{\eta_1 N_2}$$

$\hat{L}$ provides an estimation to L using only operator and operand counts and was postulated because v^* is not always readily available. The effort E of a module is defined as

$$E = \frac{V}{\hat{L}}$$

Therefore, the effort required to implement a module increases as the size of the module increases. The difficulty D is the inverse of the program level and is defined as

$$D = \frac{1}{L}$$

D is approximated by $\hat{D}$ where

$$\hat{D} = \frac{\eta_1 N_2}{2\eta_2}$$

$\hat{D}$, like $\hat{L}$, is not based on v^*. As the program level decreases, the difficulty increases. Therefore, redundant usage of operands and the failure to use higher level control constructs increase the volume as well as the difficulty of a module. Studies have shown that $\hat{D}$ is a good measure of the relative error-proneness of a module [28].

C. Coupling

Coupling refers to the interconnection between modules. It is important to know how modules in a program are coupled in order to maintain the program properly. If modules are strongly coupled, any modification or correction to one module may require that other modules be modified. Coupling is also an indicator of complexity, i.e., the difficulty in understanding a program; this complexity

affects other aspects such as testability and reliability. As the coupling between modules becomes stronger, the complexity of the relationships between the modules increases, and the program thus becomes more complex.

The following factors affect coupling:

- The type of connection between modules
- The complexity of the interface
- The type of information flow along the connection
- The binding time of the connection [37]

The type of connection may be minimal, normal or pathological. Minimally connected modules are fully parameterized modules with the smallest number of interfaces possible (e.g., a subroutine that communicates data through the use of parameters). The fewer parameters that are passed, the less complex the interface between modules will be. Normally connected modules are minimally connected modules with at least one of two exceptions:

1. There may be multiple entry points in a module.
2. Return from a module may be to a point other than the next statement following the call to that module.

Modules are pathologically connected if they transfer control to a point inside another module (i.e., a point that is not the normal entry point) or if they make references to data elements outside their own module boundaries [37]. Ideally, modules should be minimally or normally connected in order to reduce the dependencies between them.

The complexity of an interface may be measured by counting the number of items involved in the interface. For example, one could count the number of parameters passed between subroutines or the number of data elements shared between modules. Obviously, an interface involving 100 parameters is more complex than one involving only five parameters. In general, the larger the number of items involved in an interface, the more complex the interface will be.

The types of information that may flow between modules are data and control. If data are passed between modules, the data are data-coupled. This form of coupling is necessary if modules are to communicate with each other; it is also the weakest and most desirable of the four types of coupling. If information that influences the execution of a module is passed to the module from a superordinate or subordinate module, the modules are control-coupled. A module that is control-coupled requires directions from another module in order to perform its function; this situation obviously strengthens the dependencies between the modules. Control coupling may occur when data affecting the execution of a module (e.g., a flag that determines the next course of operations) are passed to the module.

Another type of coupling is common environment coupling. This form occurs when data are shared in a data base or other common areas such as the *extern* variables in C. Data may be accessible to several modules, and a modification in one module may impact other modules. If many modules share data in the common environment, the relationships between modules may become quite complex. The nature of the relationships also depend on the number of elements in the environment that is shared by the modules.

The most severe type of coupling is hybrid coupling. This form occurs when one module passes information that will actually modify the contents of the receiving module. In summary, the various types of coupling may be ranked from weakest to strongest as follows:

- Data coupling
- Common environment coupling
- Control coupling
- Hybrid coupling

Binding refers to "the process of resolving or fixing the values of identifiers used within a system." [37] The time at which this resolution occurs is referred to as the binding time. For example, if values are "hard-coded" in a program, the binding occurs when the program is written. Binding may also occur when a program is assembled or compiled, when it is linked or loaded, or when it is actually executed. The earlier a value is assigned to a variable, the more difficult it becomes to change that value. The program will have to be recompiled (and linked and loaded again) for any changes in hard-coded values. Thus, it is desirable to have binding occur only during the execution of the program.

D. Cohesion

Cohesion refers to the relationship between elements in a module. An element may be a line of code or a group of lines that together perform a "small" function.

Ideally, every element in a module should be there because it is related in some way to other elements in the module. Generally, the greater the cohesion of individual modules, the weaker the coupling between the modules [37]. The following are the seven levels of cohesion (from lowest to highest) that have been described by Yourdon [37]:

- Coincidental — No elements are related. The module performs a variety of functions that have nothing to do with each other.
- Logical — Elements fall into the same logical class of similar or related functions. For example, a module that performs all calculations needed by a program is logically cohesive.
- Temporal — Elements are related by time. For example, initialization modules that contain all elements having to do with 'start-up" are temporally cohesive.
- Procedural — Elements are part of a data structure or an algorithm. For example, modules whose purpose is to perform subordinate modules are procedurally cohesive.
- Communicational — All elements operate upon the same input data or produce the same output data. A module that assembles items from various places to produce a print line is communicationally cohesive.
- Sequential — Output data from one processing element serve as input data for the next processing element.
- Functional — Every element in a module is part of a single function.

The following guidelines can be used to help distinguish non-functional cohesive modules.

- If the only reasonable way of describing the module's operation is with a compound sentence, then the module cohesion is probably sequential, communicational, or logical in terms of cohesion.
- If the descriptive sentence contains such time-oriented words as "first," "next," "after,", "then," "start,", "step,", "when," "until," or "for all," then the module cohesion is probably has temporal or procedural cohesion.
- If the predicate of the descriptive sentence does not contain a single specific object following the verb, the module cohesion is probably logically cohesive.
- Words such as "initialize," "clean-up," and "housekeeping" in the descriptive sentence imply temporal cohesion.

APPENDIX D

Abbreviated Module Names

Table VII. Abbreviations for Module Names

Module Number	Module Abbreviation	Module Name
1	MAIN	SETS_MAIN
2	P01	BROWSE_PROJECT
3	P02	CLOSE_PROJECT_FILES
4	P03	COLLECT_PROJECT_DATA
5	P04	DEFINE_PROJECT
6	P05	DELETE_PROJECT
7	P06	DELETE_PROJECT_INTERNAL
8	P07	DESTROY_PROJECT_LISTS
9	P08	DISPLAY_PROJECT_LOG
10	P09	DISPLAY_PROJECT_SCREEN_1
11	P10	FORMAT_PROJECT_DATA
12	P11	GET_DEFINED_PROJECTS
13	P12	GET_NEW_PROJECT
14	P13	INITIALIZE_PROJECT
15	P14	INSERT_PROJECT
16	P15	MOVE_PROJECT_DATA
17	P16	OPEN_PROJECT_FILES
18	P17	PRINT_PROJECT
19	P18	PROJECT_SCREENS
20	P19	REMOVE_PROJECT_SCREEN_1
21	P20	RETRIEVE_PROJECT_DATA
22	P21	SELECT_PROJECT
23	P22	STORE_PROJECT_DATA
24	P23	UPDATE_PROJECT

Table VII. (Continued)

Module Number	Module Abbreviation	Module Name
25	M01	BROWSE_MODULE
26	M02	CLOSE_MODULE_FILES
27	M03	COLLECT_MODULE_DATA
28	M04	DEFINE_MODULE
29	M05	DELETE_MODULE
30	M06	DELETE_MODULE_INTERNAL
31	M07	DESTROY_MODULE_LISTS
32	M08	DISPLAY_MODULE_LOG
33	M09	DISPLAY_MODULE_SCREEN_1
34	M10	FORMAT_MODULE_DATA
35	M11	GET_DEFINED_MODULES
36	M12	GET_NEW_MODULE
37	M13	INITIALIZE_MODULE
38	M14	INSERT_MODULE
39	M15	MODULE_SCREENS
40	M16	MOVE_MODULE_DATA
41	M17	OPEN_MODULE_FILES
42	M18	PRINT_MODULE
43	M19	REMOVE_MODULE_SCREEN_1
44	M20	RETRIEVE_MODULE_DATA
45	M21	STORE_MODULE_DATA
46	M22	UPDATE_MODULE

Table VII. (Continued)

Module Number	Module Abbreviation	Module Name
47	F01	BROWSE_FAILURE
48	F02	BROWSE_FAILURE_SCREEN_1
49	F03	BROWSE_FAILURE_SCREEN_2
50	F04	CLOSE_FAILURE_FILES
51	F05	COLLECT_FAILURE_DATA_SCREEN_1
52	F06	COLLECT_FAILURE_DATA_SCREEN_2
53	F07	DEFINE_FAILURE
54	F08	DELETE_FAILURE
55	F09	DELETE_FAILURE_INTERNAL
56	F10	DESTROY_FAILURE_LISTS
57	F11	DISPLAY_FAILURE_LOG
58	F12	DISPLAY_FAILURE_SCREEN_1
59	F13	DISPLAY_FAILURE_SCREEN_2
60	F14	FAILURE_SCREENS
61	F15	FORMAT_FAILURE_DATA
62	F16	GET_DEFINED_FAILURES
63	F17	GET_NEW_FAILURE
64	F18	INITIALIZE_FAILURE
65	F19	INSERT_FAILURE
66	F20	MOVE_FAILURE_DATA
67	F21	OPEN_FAILURE_FILES
68	F22	PRINT_FAILURE
69	F23	REMOVE_FAILURE_SCREEN_1
70	F24	REMOVE_FAILURE_SCREEN_2
71	F25	RETRIEVE_FAILURE_DATA
72	F26	STORE_FAILURE_DATA
73	F27	UPDATE_FAILURE
74	F28	UPDATE_FAILURE_SCREEN_1
75	F29	UPDATE_FAILURE_SCREEN_2

Table VII. (Continued)

Module Number	Module Abbreviation	Module Name
76	S01	BROWSE_SCP
77	S02	CLOSE_SCP_FILES
78	S03	COLLECT_SCP_DATA
79	S04	DEFINE_SCP
80	S05	DELETE_SCP
81	S06	DELETE_SCP_INTERNAL
82	S07	DESTROY_SCP_LISTS
83	S08	DISPLAY_SCP_LOG
84	S09	DISPLAY_SCP_SCREEN_1
85	S10	FORMAT_SCP_DATA
86	S11	GET_DEFINED_SCPS
87	S12	GET_NEW_SCP
88	S13	INITIALIZE_SCP
89	S14	INSERT_SCP
90	S15	MOVE_SCP_DATA
91	S16	OPEN_SCP_FILES
92	S17	PRINT_SCP
93	S18	REMOVE_SCP_SCREEN_1
94	S19	RETRIEVE_SCP_DATA
95	S20	SCP_SCREENS
96	S21	STORE_SCP_DATA
97	S22	UPDATE_SCP

Table VII. (Continued)

Module Number	Module Abbreviation	Module Name
98	Q01	QUERY_DATA
99	Q02	COLLECT_QUERY_DATA
100	Q03	PARSE_QUERY
101	Q04	PRINT_KEYS
102	Q05	QUERY_PROJECT
103	Q06	GET_PROJECT_KEYS
104	Q07	PRINT_PROJECT_QUERY
105	Q08	QUERY_MODULE
106	Q09	GET_MODULE_KEYS
107	Q10	PRINT_MODULE_QUERY
108	Q11	QUERY_FAILURE
109	Q12	GET_FAILURE_KEYS
110	Q13	COLLECT_FAILURE_QUERY_DATA
111	Q14	PRINT_FAILURE_QUERY
112	Q15	QUERY_SCP
113	Q16	GET_SCP_KEYS
114	Q17	PRINT_SCP_QUERY
115	T01	READ_CHARACTER
116	T02	WREAD_STRING
117	T03	PROCESS_TERMINATOR
118	T04	INSERT_CHARACTER
119	T05	OVERSTRIKE_CHARACTER
120	IO1	CLOSE_FILE
121	IO2	DELETE
122	IO3	IREAD
123	IO4	IWRITE
124	IO5	OPEN_EXISTING
125	IO6	OPEN_INDEXED_FILE
126	IO7	SREAD
127	IO8	UNLOCK
128	IO9	UPDATE

Table VII. (Continued)

Module Number	Module Abbreviation	Module Name
129	D01	INITSCR
130	D02	INIT_FORMS
131	D03	DELWIN
132	D04	WDISPLAY_FORM
133	D05	DISPLAY_HELP
134	D06	DISPLAY_WINDOW
135	D07	ECHO
136	D08	ENDWIN
137	D09	MVWADDSTR
138	D10	NEWWIN
139	D11	NOCRMODE
140	D12	NOECHO
141	D13	NONL
142	D14	REVERSE_OFF
143	D15	REVERSE_ON
144	D16	WSETCENTER
145	D17	WSET_CURSOR
146	D18	STANDEND
147	D19	STANDOUT
148	D20	WCLRTOEOL
149	D21	WDRAW_LINE
150	D22	WDRAW_RECTANGLE
151	D23	WERASE
152	D24	WMOVE
153	D25	WREFRESH

Table VII. (Continued)

Module Number	Module Abbreviation	Module Name
154	C01	ADD_IEEE_TEXT
155	C02	CHECK_FOR_VALID_CATEGORY
156	C03	CHECK_FOR_VALID_DISPOSITION
157	C04	CHECK_MENU_ANSWER
158	C05	CHECK_PICTURE_STRING
159	C06	CHECK_RECORD
160	C07	CONVERT_DATE
161	C08	CREATE_TEXT_FILE
162	C09	CREATE_TPUINI
163	C10	DEFINE_MENU
164	C11	DELETE_DATA
165	C12	DELETE_HEAD
166	C13	DELETE_PREVIOUS_CHARACTER
167	C14	DELETE_PREVIOUS_WORD
168	C15	DELETE_TAIL
169	C16	DESTROY_LIST
170	C17	DISPLAY_IEEE_TEXT
171	C18	EDIT_SCROLLED_FIELD
172	C19	EDIT_TEXT
173	C20	FORMAT_FIELD
174	C21	FORMAT_TEXT
175	C22	GET_DATE
176	C23	GET_IEEE_TEXT
177	C24	GET_OPERATOR
178	C25	GET_TIME
179	C26	HEADER
180	C27	INDEX
181	C28	INSERT_TAIL
182	C29	INSERT_TEXT
183	C30	MAIN_MENU

Table VII. (Continued)

Module Number	Module Abbreviation	Module Name
184	C31	MENU
185	C32	MESSAGE
186	C33	MOVE_TO_BOTTOM_TEXT
187	C34	MOVE_TO_NEXT_TEXT
188	C35	MOVE_TO_PREVIOUS_TEXT
189	C36	NO_MORE_DATA
190	C37	PAD
191	C38	PRINT_DATA
192	C39	PRINT_FORM
193	C40	PRINT_LIST
194	C41	PRINT_MENU
195	C42	PRINT_OPTIONS
196	C43	PRINT_TEXT
197	C44	READ_TEXT_FILE
198	C45	REMOVE_BLANK_LINES
199	C46	REMOVE_FROM_LIST
200	C47	RETRIEVE_TEXT
201	C48	SCROLL_HELP_TEXT
202	C49	SCROLL_TEXT
203	C50	SET_NEXT_FIELD
204	C51	SPAWN_IMAGE
205	C52	STORE_TEXT
206	C53	STRIPCPY
207	C54	STRIPLEN
208	C55	UNCONVERT_DATE
209	C56	UPPERCASE
210	C57	CONVERT_PROJECT_NUMBER_TO_TITLE
211	C58	DELETE_RECORD

APPENDIX E

McCabe and Halstead Metric Analysis

Table VIII. McCabe and Halstead Analysis of SETS Modules

Module	$V(G)$	N	$\hat{N}$	V	$\hat{L}$	E	$\hat{D}$
MAIN	2	40	77	178	0.138	1288	7
P01	5	188	253	1076	0.018	60411	56
P02	2	27	35	96	0.067	1452	15
P03	4	266	447	1695	0.027	62996	37
P04	11	389	392	2423	0.013	182432	75
P05	3	110	174	585	0.045	13068	22
P06	1	41	43	151	0.017	9103	60
P07	2	37	35	132	0.036	3648	27
P08	4	130	203	713	0.033	21864	30
P09	1	143	238	811	0.038	21445	26
P10	2	294	343	1783	0.027	65286	36
P11	2	63	97	296	0.064	4644	15
P12	2	51	74	224	0.053	4200	18
P13	1	218	218	1204	0.007	160916	133
P14	2	66	70	285	0.033	8557	30
P15	2	90	133	453	0.071	6409	14
P16	2	47	63	199	0.083	2396	12
P17	2	46	54	188	0.064	2930	15
P18	3	52	74	228	0.057	3997	17
P19	1	28	40	103	0.073	1425	13
P20	3	100	151	512	0.022	22968	44
P21	14	309	341	1874	0.013	148816	79
P22	2	220	327	1324	0.022	60033	45
P23	9	214	290	1258	0.018	69763	55

Table VIII. (Continued)

Module	$V(G)$	N	$\hat{N}$	V	$\hat{L}$	E	$\hat{D}$
M01	7	153	210	845	0.021	39645	46
M02	2	22	31	76	0.089	856	11
M03	4	217	372	1338	0.029	46720	34
M04	12	405	426	2560	0.012	214242	83
M05	3	108	173	574	0.052	11117	19
M06	1	46	61	188	0.024	7756	41
M07	2	27	31	93	0.056	1681	18
M08	4	144	238	816	0.032	25271	30
M09	1	143	238	811	0.038	21445	26
M10	2	253	303	1500	0.031	47680	31
M11	3	93	128	465	0.044	10463	22
M12	1	13	20	41	0.250	165	4
M13	1	188	179	1000	0.011	94959	94
M14	2	66	70	285	0.033	8557	30
M15	3	52	74	228	0.057	3997	17
M16	2	79	117	387	0.077	5039	13
M17	2	37	54	151	0.099	1525	10
M18	2	46	54	188	0.064	2930	15
M19	1	23	35	82	0.100	825	10
M20	3	107	171	565	0.025	22268	39
M21	3	224	340	1358	0.022	62845	46
M22	9	214	290	1258	0.018	69763	55

Table VIII. (Continued)

Module	$V(G)$	N	$\hat{N}$	V	$\hat{L}$	E	$\hat{D}$
F01	4	42	126	206	0.042	4946	24
F02	7	139	169	734	0.022	33454	45
F03	7	210	253	1202	0.016	77404	64
F04	2	32	40	118	0.052	2279	19
F05	6	276	453	1764	0.023	76452	43
F06	6	241	412	1514	0.023	66969	44
F07	12	405	426	2560	0.012	214242	83
F08	3	100	156	520	0.059	8798	16
F09	1	60	71	254	0.016	16248	63
F10	2	47	40	173	0.026	6696	38
F11	4	125	191	678	0.034	20134	29
F12	2	161	275	939	0.036	26261	27
F13	2	171	295	1010	0.031	32323	32
F14	3	52	74	228	0.057	3997	17
F15	2	551	550	3636	0.017	217511	59
F16	3	93	128	465	0.044	10463	22
F17	2	51	74	224	0.053	4200	18
F18	1	322	331	1932	0.006	331168	171
F19	2	66	70	285	0.033	8557	30
F20	2	153	221	854	0.051	16615	19
F21	2	57	72	250	0.072	3458	13
F22	2	46	54	188	0.064	2930	15
F23	1	18	31	62	0.148	420	6
F24	1	28	40	103	0.073	1425	13
F25	3	147	233	825	0.017	48490	58
F26	2	316	487	2046	0.018	116022	56
F27	3	42	64	178	0.077	2319	13
F28	9	280	374	1727	0.015	118512	68
F29	9	280	367	1721	0.014	122198	70

Table VIII. (Continued)

Module	$V(G)$	N	$\hat{N}$	V	$\hat{L}$	E	$\hat{D}$
S01	7	158	210	872	0.021	42430	48
S02	2	22	31	76	0.089	856	11
S03	4	165	282	966	0.031	31628	32
S04	12	405	426	2560	0.012	214242	83
S05	3	100	156	520	0.059	8798	16
S06	1	46	61	188	0.024	7756	41
S07	2	27	31	93	0.056	1681	18
S08	4	137	220	765	0.033	23511	30
S09	1	132	220	737	0.039	18732	25
S10	2	294	291	1729	0.023	74555	43
S11	3	93	128	465	0.044	10463	22
S12	2	51	74	224	0.053	4200	18
S13	1	172	173	909	0.012	76363	84
S14	2	66	70	285	0.033	8557	30
S15	2	67	101	318	0.084	3774	11
S16	2	37	54	151	0.099	1525	10
S17	2	46	54	188	0.064	2930	15
S18	3	107	171	565	0.025	22268	39
S19	3	107	171	565	0.025	22268	39
S20	3	52	74	228	0.057	3997	17
S21	2	200	295	1181	0.024	49093	41
S22	9	257	355	1569	0.015	107450	68

Table VIII. (Continued)

Module	$V(G)$	N	$\hat{N}$	V	$\hat{L}$	E	$\hat{D}$
Q01	11	341	393	2124	0.014	153680	72
Q02	4	69	110	331	0.033	9951	30
Q03	12	315	361	1930	0.012	160733	83
Q04	2	226	270	1312	0.031	41835	31
Q05	7	209	276	1219	0.023	52149	42
Q06	2	60	91	278	0.068	4078	14
Q07	1	73	108	350	0.048	7264	20
Q08	8	226	314	1350	0.021	65727	48
Q09	2	60	91	278	0.068	4078	14
Q10	1	87	135	438	0.041	10752	24
Q11	8	217	295	1281	0.022	59329	46
Q12	2	60	91	278	0.068	4078	14
Q13	9	117	192	634	0.033	19420	30
Q14	1	87	135	438	0.041	10752	24
Q15	8	226	314	1350	0.021	65727	48
Q16	2	60	91	278	0.068	4078	14
Q17	1	87	135	438	0.041	10752	24
T01	5	271	349	1649	0.032	52273	31
T02	15	426	379	2636	0.012	219519	83
T03	4	188	214	1044	0.031	33525	32
T04	9	186	150	961	0.025	39228	40
T05	8	124	140	630	0.033	18925	30
IO1	2	36	77	160	0.099	1623	10
IO2	3	33	44	128	0.135	958	7
IO3	4	98	145	502	0.043	11729	23
IO4	3	63	91	292	0.068	4282	14
IO5	1	19	36	70	0.208	337	4
IO6	3	138	271	801	0.029	27597	34
IO7	4	76	112	369	0.058	6362	17
IO8	3	33	44	128	0.135	958	7
IO9	3	58	81	262	0.073	3581	13

Table VIII. (Continued)

Module	$V(G)$	N	$\hat{N}$	V	$\hat{L}$	E	$\hat{D}$
D01	1	45	62	191	0.112	1708	8
D03	1	6	12	15	0.400	39	2
D04	3	102	150	527	0.046	11535	21
D05	1	102	128	510	0.066	7735	15
D06	7	147	153	759	0.017	44079	58
D07	1	5	7	11	0.667	17	1
D08	1	11	18	33	0.333	99	3
D09	1	41	63	174	0.090	1941	11
D10	1	35	40	133	0.150	888	6
D12	1	5	7	11	0.667	17	1
D14	1	5	7	11	0.667	17	1
D15	1	5	7	11	0.667	17	1
D16	1	25	48	100	0.173	579	5
D17	5	49	63	208	0.097	2141	10
D18	1	5	7	11	0.667	17	1
D19	1	5	7	11	0.667	17	1
D20	1	6	12	15	0.400	39	2
D21	1	30	31	107	0.159	678	6
D22	1	30	31	107	0.159	678	6
D23	1	18	24	59	0.171	349	5
D24	1	18	24	59	0.171	349	5

Table VIII. (Continued)

Module	$V(G)$	N	$\hat{N}$	V	$\hat{L}$	E	$\hat{D}$
C01	5	319	328	1921	0.023	81901	42
C02	2	54	76	240	0.126	1906	7
C03	2	39	62	165	0.138	1196	7
C04	8	144	174	766	0.026	29313	38
C05	12	261	221	1457	0.032	45813	31
C06	23	417	251	2388	0.015	162156	67
C07	2	79	77	352	0.095	3724	10
C08	2	47	77	209	0.071	2934	14
C09	1	129	109	620	0.096	6463	10
C10	2	82	100	385	0.029	13077	33
C11	3	59	77	263	0.060	4370	16
C12	2	28	38	103	0.067	1554	15
C13	6	162	157	843	0.021	39609	46
C14	7	168	151	868	0.020	42993	49
C15	3	48	50	192	0.061	3168	16
C16	2	16	26	53	0.125	425	8
C17	5	151	208	834	0.029	28298	33
C18	9	246	245	1402	0.016	87963	62
C19	2	131	208	723	0.047	15557	21
C20	5	105	145	538	0.044	12152	22
C21	6	174	196	949	0.029	32773	34
C22	8	187	245	1065	0.031	34264	32
C23	5	68	76	303	0.096	3159	10
C24	5	147	156	765	0.036	21017	27
C25	9	140	226	786	0.045	17614	22
C26	4	253	288	7732	0.008	17614	129
C27	3	46	78	205	0.058	3517	17
C28	2	63	70	272	0.035	7760	28
C29	3	108	161	566	0.054	10485	18
C30	4	170	271	987	0.026	38144	38

Table VIII. (Continued)

Module	$V(G)$	N	$\hat{N}$	V	$\hat{L}$	E	$\hat{D}$
C31	7	163	203	895	0.022	40283	45
C32	2	64	103	304	0.056	5432	17
C33	7	106	104	504	0.033	15121	30
C34	12	180	108	865	0.022	38940	45
C35	12	168	105	798	0.019	41738	52
C36	1	5	8	11	0.500	23	2
C37	2	27	40	102	0.101	1018	9
C38	3	200	214	1110	0.026	42922	38
C39	2	110	115	534	0.027	19594	36
C40	2	41	63	174	0.083	2090	12
C41	3	94	135	474	0.032	14936	31
C42	5	206	320	1236	0.026	46927	37
C43	1	54	62	229	0.086	2676	11
C44	2	90	128	450	0.046	9836	21
C45	3	45	81	200	0.042	4776	23
C46	4	85	68	367	0.040	9210	25
C47	3	80	117	392	0.056	6954	17
C48	9	233	314	1392	0.021	65313	46
C49	14	404	447	2575	0.017	153329	59
C50	5	57	68	246	0.060	4118	16
C51	1	25	32	89	0.111	807	9
C52	2	75	109	360	0.039	9134	25
C53	2	39	54	159	0.073	2192	13
C54	5	63	73	276	0.040	6918	25
C55	2	86	91	399	0.067	5962	14
C56	2	31	40	118	0.085	1381	11
C57	2	67	107	322	0.069	4646	14
C58	3	33	44	128	0.135	958	7

APPENDIX F

Coupled Modules

MAIN — D02, C26, D16, T01, C30, D08
P01 — P09, C49, P19
P02 — IO1
P03 — T02, C18, C19, C50
P04 — C10, P11, P08, C31, D05, P18, P12, P05, C36, D17, C16
P05 — C54, T02, D17, D20, P06, M06, F09, S06, C46
P06 — P16, C11, P02
P07 — C16
P08 — D04, C53, D17, C41
P09 — D04, D10, D17, D06
P10 — D17, D20, C20, C21
P11 — IO6, IO7, P14, IO1
P12 — P14
P13 — C37
P14 — P13, P15
P16 — IO6
P17 — P10, C42, C16
P18 — P16, P20, P23, P01, P17, P02, P07
P19 — D23, D03
P20 — IO3, P15, P07, C47
P21 — C10, P11, C32, P08, C31, D05, C36, C16
P22 — D23, D17, IO3, IO9, IO4, C52, C53
P23 — C10, P09, C41, P03, C31, D05, P22, T02, P19, C16
M01 — M09, C49, M19
M02 — IO1
M03 — T02, C55, C22, C07, C18, C19, C50
M04 — C10, P21, M11, M08, C31, D05, M15, M12, M05, C36, D17, C16
M05 — C54, T02, D17, D20, M06, C46
M06 — M13, M17, C11, M02
M07 — C16
M08 — D04, C53, D17, C41
M09 — D04, D10, D17, C55, D06
M10 — D17, D20, C20, C55, C21
M11 — IO6, IO3, M14, IO7, IO1
M12 — M14
M13 — C37
M14 — M13, M16
M15 — M17, M20, M22, M01, M18, M02, M07
M17 — IO6
M18 — M10, C42, C16
M19 — D23, D03
M20 — IO3, M16, M07, C47, C57
M21 — D23, D17, C54, C32, IO3, IO9, IO4, C52
M22 — C10, M09, C41, M03, C31, D05, M21, T02, M19, C16
F01 — F02, F03, C32
F02 — F12, C49, F23
F03 — F13, C49, F24
F04 — IO1
F05 — T02, C02, C55, C22, C07, C25, C19, C50
F06 — T02, C03, C18, C19, C50
F07 — C10, P21, F16, F11, C31, D05, F14, F17, F08, C36, D17, C16

F08 — T02, D17, D20, F09, C46
F09 — F18, F21, C11, F04
F10 — C16
F11 — D04, D17, C41
F12 — D04, D10, D17, C55, D06
F13 — D04, D10, D17, D06
F14 — F21, F25, F27, F01, F22, F04, F10
F15 — D17, D20, C20, C55, C21
F16 — IO6, IO3, F19, IO7, IO1
F17 — F19
F18 — C37
F19 — F18, F20
F21 — IO6
F22 — F15, C42, C16
F23 — D23, D03
F24 — D23, D03
F25 — IO3, F20, F10, C47, C57
F26 — D23, D17, IO3, IO9, IO4, C52, C53
F27 — F28, F29, C32
F28 — C10, F12, C41, F05, C31, D05, F26, C01, D06, T02, F23, C16
F29 — C10, F13, C41, F06, C31, D05, F26, C01, D06, T02, F24, C16
S01 — S09, C49, S18
S02 — IO1
S03 — T02, C55, C22, C07, C19, C50
S04 — C10, P21, S11, S08, C31, D05, S20, S12, S05, C36, D17, C16
S05 — T02, D17, D20, S06, C46
S06 — S13, S16, C11, S02
S07 — C16
S08 — D04, D17, C41
S09 — D04, D10, D17, C55, D06
S10 — D17, D20, C20, C55, C21
S11 — IO6, IO3, S14, IO7, IO1
S12 — S14
S13 — C37
S14 — S13, S15
S16 — IO6
S17 — S10, C42, C16
S18 — D23, D03
S19 — IO3, S15, S07, C47, C57
S20 — S16, S19, S22, S01, S17, S02, S07
S21 — D23, D17, IO3, IO9, IO4, C52, C53
S22 — C10, S09, C41, S03, C31, D05, S21, C01, D06, T02, S18, C16
Q01 — C10, D23, D17, C31, D05, Q06, Q05, Q09, Q08, Q12, Q11, Q16, Q15, C38
Q02 — T02, C50
Q03 — C54, C24, C07, IO6, IO3, C06, IO7, C28, IO1
Q04 — IO6, C28, IO3, IO1, D10, D22, C48, D03, C16
Q05 — C10, D04, C41, Q02, C31, D05, Q03, C16
Q06 — IO6, IO7, C28, IO1
Q07 — P13, P16, P20, P02, P10, C20, P07
Q08 — C10, D04, C41, Q02, C31, D05, Q03, C16
Q09 — IO6, IO7, C28, IO1

Q10 — M13, M17, M20, M02, M10, C20, M07
Q11 — C10, Q13, C31, D05, Q03, C16
Q12 — IO6, IO7, C28, IO1
Q13 — D04, C41, T02, C50
Q14 — F18, F21, F25, F04, F15, C20, F10
Q15 — C10, D04, C41, Q02, C31, D05, Q03, C16
Q16 — IO6, IO7, C28, IO1
Q17 — S13, S16, S19, S02, S10, C20, S07
T01 — D24, D25
T02 — C37, C54, D17, D20, D09, T01, C05, T05, T04, T03
T03 — C54, C13, C14, C37, D09
T04 — D09
T05 — D09
D02 — D01, D07, D11, D13
D04 — IO6, C37, IO3, C39, IO7, IO1
D05 — D10, D22, IO6, C47, IO1, C48, D03, C16
D06 — D23, D17
D16 — D17, D17, D19, D15, D09, D18, D14
C01 — C10, C23, D10, C17, C31, D05, C29, C33, C35, C34, D17, D03, C16
C02 — C32
C03 — C32
C04 — D17, C56
C05 — D17
C06 — C54, C56, C27
C08 — C53
C10 — C54
C11 — IO3, C58, IO7
C13 — D09
C14 — D09, C37
C16 — C16
C17 — D04, C54, D17, C54, C41
C18 — D06, T02, C28, C54, C45
C19 — D23, T02, C09, C08, C16, C51, C44, D06, IO2
C20 — C28, C54
C21 — C28, C20, C54
C22 — T02, C32
C23 — C44
C25 — C37, T02, C32
C26 — C56, D17, D16
C29 — C54, C28, D17
C30 — C10, D04, C41, C31, D05, P04, M04, F07, S04, Q01, D17, C16, D23
C31 — C41, T01, C04
C32 — D12, T02, D07, D23
C33 — C54
C34 — C54, C36
C35 — C54, C36
C36 — C32
C38 — D17, D20, Q07, Q10, Q14, Q17, D10, D22, C48, D03, C16
C39 — D21, D22, D23, D17, D16
C40 — C54
C41 — D23, D17

C42 — C10, C31, D05, C40, T02, C32, D17, C16
C43 — C21, C42, C16
C44 — C37, C28
C45 — C46
C46 — C15, C12
C47 — IO3, C28, IO7, IO8
C48 — C10, D06, C31, D05, C36, C43, D17, C16
C49 — C10, C54, D17, D16, D06, C31, D05, C36, C16
C51 — C54
C52 — C11, IO4
C53 — C54
C55 — C54
C57 — IO6, IO3, C37, IO1

VITA

Mark A. Floyd was born in Gadsden, Alabama on February 14, 1955. Mr. Floyd was graduated from Athens High School, Athens, Alabama, in May 1973. The following September he entered the University of Tennessee, and in December 1976 he received a Bachelor of Science degree in Chemistry. In February 1977 he accepted a position with Ames Laboratory of the Iowa State University as an assistant chemist. He began study toward a Master's degree in chemistry shortly after. This degree was awarded in March 1980.

Mr. Floyd accepted a position with the Oak Ridge Gaseous Diffusion Plant in March 1980. In April 1983 he entered the Graduate School of the University of Tennessee and began study toward a Master's degree in Computer Science. He received the Master of Science degree with a major in Computer Science in June 1988.

Mr. Floyd is currently employed as a development staff member by Martin Marietta Energy Systems, Inc., operators of the Department of Energy installations in Oak Ridge since 1984.