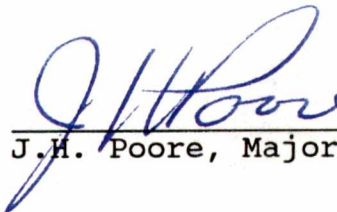


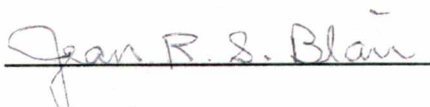
To the Graduate Council:

I am submitting herewith a thesis written by Charles C. Senn Jr. entitled "Box Structure BDL Generator: A Clerical Tool to Support Software Development." 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 the rules of the Library. Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgement of the source is made.

Permission for extensive quotation from or reproduction of this thesis may be granted by my major professor, or in his absence, by the Head of InterLibrary Services when, in the opinion of either, the proposed material is for scholarly purposes. Any copying or use of the material in this thesis for financial gain shall not be allowed without my written permission.

Signature Chad C. Allen

Date 7/28/89

BOX STRUCTURE BDL GENERATOR:

A Clerical Tool To Support Software Development

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Charles C. Senn Jr.

August 1989

ACKNOWLEDGEMENTS

The completion of the Master of Science Degree is a worthy goal, and one which has challenged not only my intellectual abilities, but also my physical endurance.

I could not have accomplished this task without the unfailing love and devotion of my wife, Sue, who worked two jobs in addition to maintaining a home while I was in school, and the patience and understanding of my daughters, Elizabeth, and Mary, who have had a dad that was "always studying" for two years.

I would like to recognize the other members of the programming team, each one of whom contributed hundreds of hours in the development of The Box Structure BDL Generator, the basic research for this thesis: Carmen Trammell, the encourager; Hialong Mao, "no problem"; Norman Sims, "what else can I do to help". And also, the members of the testing team, Jim Whittaker and Scott Hopkins, whose work I was not able to view first hand, but was vital for the successful completion of the project, and to Chris Jepeway for his time and 'C' language expertise given during the code review.

My special thanks is extended to Professor J.H. Poore, for his personal interest in me, and for excellent advise and counsel offered during the development of this thesis, to Professor David Straight, for extending to me the opportunity to become a graduate teaching assistant at the university, and serving as an advisor on the thesis committee, and also to my third advisor, Professor Jean Blair, for her constructive appraisal, and helpful comments.

Finally, I would like to thank Mr. Leon Binder. Mr. Binder first encouraged me to apply for the graduate teaching assistantship, later, he was instrumental in developing my enthusiasm for software quality, and recently, provided me with excellent counsel which led to my acceptance of a position with IBM.

ABSTRACT

The Cleanroom Methodology, properly applied, produces functionally verified statistically certifiable software that, when taken to testing, will contain an average of less than five errors per thousand lines of code (kloc).

When one considers that current industry data indicates average rates of approximately seventy-five errors per kloc, the claims of the Cleanroom Methodology on first inspection, do not appear credible.

This thesis details the design and development of a 7200 line program, The Box Structure BDL Generator, written in 'C' by a team of graduate students using the Cleanroom methodology. The Box Generator is a tool to facilitate top down design using the Box Structure method. The primary goal of this team was to learn and apply the Box Structure method in the development of the Box Generator, and meet the exacting standards set by the Cleanroom methodology.

TABLE OF CONTENTS

1.	INTRODUCTION	1
1.1.	TECHNOLOGY EXPLOSION	1
1.2.	SOFTWARE FRUSTRATIONS	1
1.3.	CLEANROOM AND STATISTICAL QUALITY CONTROL .	2
1.4.	THESIS GOAL	3
2.	THE CLEANROOM METHODOLOGY	5
2.1.	ERROR PREVENTION	5
2.2.	OVERVIEW	6
2.3.	EXPANSION PROCESS	7
2.4.	FUNCTIONAL VERIFICATION	8
2.5.	CERTIFICATION AND RELIABILITY	9
2.6.	CLEANROOM SOFTWARE DEVELOPMENT LIFE CYCLE .	11
2.7.	PROBLEM ANALYSIS AND SPECIFICATION	11
2.8.	BOX STRUCTURE ANALYSIS AND DESIGN	12
2.9.	TESTING TEAM ACTIVITY	13
2.10.	FEEDBACK	13
2.11.	TARGET LANGUAGE TRANSLITERATION	14
2.12.	TESTING ENVIRONMENT	14
2.13.	COMMON SERVICE TESTING	15
2.14.	INCREMENT FEEDBACK	15
2.15.	RELEASE FEEDBACK	17

3.	BOX STRUCTURE BDL GENERATOR	18
3.1.	MOTIVATION	18
3.1.1.	REINFORCE STEPWISE REFINEMENT	19
3.1.2.	BDL EDITING	26
3.1.3.	BDL PRINTING	26
3.1.4.	STIMULATE FURTHER RESEARCH	26
3.1.5.	LIFE CYCLE DATA	27
3.2.	GENERAL DESCRIPTION	27
3.2.1.	DESIGN SEQUENCE	28
3.2.2.	WELL DEFINED SPECIFICATION DEMANDED	29
3.2.3.	BOX GENERATOR DEVELOPMENT	30
3.2.3.1.	SPECIFICATION DEVELOPMENT	30
3.2.3.2.	TOP LEVEL BOX DEVELOPMENT	32
3.2.3.3.	"STORY BOOK" DEVELOPMENT	34
3.2.3.4.	SPECIFICATION REVIEW	37
3.2.3.5.	LOWER LEVEL BDL DEVELOPMENT	37
3.2.3.6.	PROBLEMS SURFACE	39
3.2.3.7.	"PROCESS INFORMATION"	
	REDESIGN	41
3.2.3.8.	FUNCTIONAL VERIFICATION	
	PROCEDURES	42
3.2.3.9.	CODING AND CODE REVIEW	43
4.	BOX GENERATOR DESIGN	44
4.1.	OVERVIEW	44

4.2.	DESIGN PHILOSOPHY	44
4.3.	DESIGN DECISION	46
4.4.	THE DESIGN ITSELF	47
5.	STATISTICAL TEST DEVELOPMENT	49
5.1.	PRELIMINARIES	49
5.2.	SIMULATE THE ENVIRONMENT	49
5.3.	TEST DEVELOPMENT	51
5.4.	TESTING PROCEDURES	52
6.	LIFE CYCLE DATA	54
6.1.	OVERVIEW	54
6.2.	TIME LOG	54
6.3.	MODIFICATION CLASSIFICATION	55
6.4.	TEST RESULTS	62
7.	TEAM COMPOSITION AND LESSONS LEARNED	67
7.1.	TEAM COMPOSITION	67
7.2.	LESSONS LEARNED	67
7.2.1.	CRITICIZE CORRECTLY	67
7.2.2.	GOOD SPECIFICATION HIGHLIGHTS "THORNY PROBLEMS"	68
7.2.3.	DON'T BE AFRAID TO START OVER	70
7.2.4.	THINK COMMON SERVICE	71
7.2.5.	KEEP IT SIMPLE	72

8. IN CONCLUSION	73
BIBLIOGRAPHY	76
APPENDICES	79
APPENDIX A	80
APPENDIX B	81
APPENDIX C	89
APPENDIX D	95
APPENDIX E	104
APPENDIX F	105
VITA	106

1. INTRODUCTION

1.1. TECHNOLOGY EXPLOSION

In 1946, ENIAC (Electronic Numerical Integrator and Calculator), a mammoth machine consisting of 18,000 vacuum tubes, occupying 1500 square feet of space, weighing thirty tons, and consuming 174,000 watts of power, began its calculations on the feasibility of a proposed design for the hydrogen bomb¹. This began what is commonly referred to as the computer age. At the time of this writing, less than fifty years later, children play with machines that are far more powerful and occupy a space of less than 2 square feet. One is justly awed by the explosive growth in computer technology, a growth unequalled by that of any other human endeavor since civilization began.²

1.2. SOFTWARE FRUSTRATIONS

The best hardware is worthless without instructions for its execution: Software engineers are relied upon to develop such instructions. The craft of software engineering, however, seems locked in a sequence of extremely expensive processes, whose outcomes are always in question. The frustrations of

¹ Steven L. Mandell, Computers and Information Processing (Saint Paul: West Publishing Company, 1987), p. 35.

² Frederick P. Brooks, Jr. "No Silver Bullet", Computer, (April 1987), p. 10.

software production are well known: uncontrolled development costs, more dollars are spent in debugging than in design, programs revised until they are no longer maintainable, and programs that cannot be relied upon.

Specifically, difficulties with uncontrolled development costs are rampant. The proliferation of case tools and fourth generation languages attests to the fact that the computer industry is searching for an answer, any answer, that can shed light on the problem of on-time and on-budget software development. One has only to follow the rise and fall of LOTUS Corporation to understand the negative impact on sales, reputation, and long term growth, caused by out of control software development³.

1.3. CLEANROOM AND STATISTICAL QUALITY CONTROL

IBM's Cleanroom process⁴ proposes to bring software development under statistical quality control by:

1. defect prevention rather than defect removal;
2. statistical certification of software quality;
3. requirement of concurrent fabrication and certification of product increments;

³ Keith H. Hammonds, "Jim Manzi", Business Week, Special Issue, (April 14, 1989), p. 147.

⁴ Harlan D. Mills, Michael Dyer, and Richard C. Linger, "Cleanroom Software Engineering", Quality Assurance, (September 1987), p. 19.

4. accumulation of increments into the system to be delivered.⁵

This methodology promises to produce high quality, statistically certifiable, software within scheduled delivery dates. While more research must be done before a definitive statement can be made with regard to the success of the methodology, Cleanroom appears to be raising the expectations of the industry:

Cleanroom performance is a tentative standard that has emerged from the collective experience with Cleanroom projects and its comparison with other development records. Although the collective experience with Cleanroom projects is not yet sufficiently extensive to be itself statistically analyzed, the anecdotal evidence is compelling. Clean first compiles are common. We see jobs done with one-third the number of lines of code required by other efforts. Where comparable data has been available, errors in code prior to first execution have been reduced by a factor of twenty-five. Errors in released code have been reduced by nearly two orders of magnitude. Project turnaround has been halved by experienced teams. As a rule of thumb, Cleanroom quality is achieved if the error rate in statistical testing is less than five errors per thousand lines of code the first time the code goes to the machine.⁶

1.4. THESIS GOAL

This thesis describes a tool which aids software developers in the production of Box Description Language (BDL).⁷ The

⁵ Ibid.

⁶ Harlan D. Mills and J.H. Poore, "Bringing Software Under Statistical Quality Control", Quality Progress, (November 1988), p. 22.

⁷ The pseudo code used in Cleanroom.

tool, the Box Structure BDL Generator, was developed as a Cleanroom project by a group of graduate students from the University of Tennessee, for which the present author served as chief programmer. The goal set for the team was to develop the prototype, or first increment, of the tool from specification through testing and certification. That process required 3100 person hours to complete and resulted in a model of the tool written in 'C'⁸ consisting of 7200 lines of code. Testing established that the team would achieve their goal of meeting the rigid Cleanroom methodology standards.

This thesis presents a description of the tool and chronicles its development. Chapter 2 is a discussion of Cleanroom methodology; Chapter 3, a description of the tool desired; Chapter 4, the tool design; Chapter 5, testing techniques used to certify the tool; Chapter 6, life cycle data captured during the project; and Chapter 7, an analysis of lessons learned from the project.

⁸Charles C. Senn, Carmen Trammell, Hialong Mao, Norman Sims, "Box Generator 'C' Code", Technical Report, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1989.

2. THE CLEANROOM METHODOLOGY

2.1. ERROR PREVENTION

IBM Federal Systems Division (FSD)⁹ presents the Cleanroom Methodology as a design system which can produce extremely high quality (approaching zero defect) software of statistically certifiable reliability.¹⁰ The concept behind the methodology is to prevent the introduction of errors into the design through:

1. non-execution based programming development;
2. verification of each design module (box);
3. step wise functional verification of the entire design;
4. statistically based independent testing.

The Cleanroom methodology seeks to eliminate the normal errors involved with the development of software by utilizing the design steps just outlined. Steps (1) and (2) are applied to every module developed within the program thus assuring that each module is correct. When the correctness of one module is confirmed, the verification process is applied to the next module, until all modules have been verified.

⁹ M.Dyer, "Cleanroom Software Development Method", IBM Federal Systems Division, Bethesda, Maryland, (October 14), 1982.

¹⁰ P. Allen Currit, Michael Dyer, and Harlan D. Mills, "Certifying the Reliability of Software.", IEEE Transactions on Software Engineering, (January 1986), p. 3.

2.2. OVERVIEW

The Box Structure methodology¹¹ takes the view that any system can be described at the highest level as a "Black Box" which analyzes a stimulus stream and produces correct responses based upon the stimulus history that it has received. Viewed at this level, the inner workings of the box are unknown; one simply observes the stimulus history and the responses produced by the box.

The next step in the process involves the creation of the "State Box". The State Box represents the first unfolding of the inner workings of the Black Box. The program developer must first invent "state data", whose purpose is to capture the relevant parts of stimulus history that enable the box to perform its function. The functionality of the State Box is then described by the "State Box Transition Rule" and is verified in two ways. First a check is made to assure that all data required by the transition rule is available as either stimulus information or encapsulated in state data. Next a check is made to assure that the operation actually described by the State Box is the same as that described by its corresponding Black Box. When these two checks are complete, the Clear Box, which is the final unfolding of operational detail, can be described.

¹¹ H.D. Mills, R.C. Linger, and A.R. Hevner, "Box Structured Information Systems", IBM Systems Journal, V. 26, N. 4, 1987, pp. 1-15.

The purpose of the Clear Box is to use the constructs of structured programming to implement the State Box Transition Rule. Lower level Black Boxes will be created in the process. Machine data which can act as the facilitating mechanisms of the constructs, are introduced. The same checks that were carried out in the State Box transition are repeated in the Clear Box thus assuring the operational integrity of the box throughout the three layers of design. Upon satisfactory completion of the Clear Box, development is begun on the next Black Box.

2.3. EXPANSION PROCESS

Mills has outlined an eleven step process which details the design steps required in box expansion from Black Box to Clear Box.¹²

EXPANSION PROCESS

A. Define the Black Box.

1. Define black-box stimuli. Determine all possible stimuli for the Black Box.
2. Define black-box behavior. For each possible stimulus, determine its complete response in terms of its stimulus history.

B. Design the State Box.

3. Discover state data requirements. For each response to be calculated, encapsulate its stimulus history into a state data requirement.

¹² Harlan D. Mills, "Stepwise Refinement and Verification in Box-Structured Systems", Computer, (June 1988), p. 28.

4. Define the state. Select a subset of the required state data items to encapsulate stimulus histories.

5. Design the State Box. For the selected state, determine the internal Black Box required for the State Box.

6. Verify the State Box. Verify the correctness of the State Box with respect to the required black-box behavior.

C. Design the Clear Box.

7. Discover state data accesses. For each item of state data and each possible stimulus, determine all possible accesses of the item.

8. Define the data abstractions. Organize state data into data abstractions for effective access.

9. Design the Clear Box. Define sequential or concurrent uses of the data abstractions defined to replace the internal Black Box of the State Box.

10. Verify the Clear Box. Verify the correctness of the Clear Box with respect to the State Box behavior.

D. Continue the process.

11. Repeat the stepwise expansion until design completion. For each new data abstraction, repeat steps 1-10 until suitable data and program specifications are reached.

2.4. FUNCTIONAL VERIFICATION

In addition to Black Box, State Box, and Clear Box verification there is an alternative (or additional) phase of confirmation, functional verification¹³. The programming team uses the principles of predicate calculus and set theory, to

¹³ R.C. Linger, H.D. Mills, and B.I. Witt, Structured Programming, Theory and Practice (Philippines: Addison-Wesley Publishing Company, Inc., 1979), p. 214.

produce a mathematical proof of correctness for the program.

Poore describes the team interaction as

a mathematical thought-and-argument process which facilitates a detailed examination by the team in a way that will either convince the team members that the code is correct, or will expose mistakes. Mistakes are corrected and the code is again verified.¹⁴

Individual box verification is tantamount to piece-wise functional verification. Code inspections, and group walkthroughs round out the development process. The intention in Cleanroom is to "do it right the first time", to

impose discipline on software development so that system correctness results from a coherent, readable design rather than from a reliance on execution-based testing. The notion that "Well, the software should always be tested to find the faults" is eliminated.¹⁵

2.5. CERTIFICATION AND RELIABILITY

An independent testing team has as its goal the certification of the reliability of the increment, rather than traditional error detection. This change in perspective demands that the testers evaluate the user environment of the product, so as

¹⁴ J.H. Poore, "An Overview of the Cleanroom Software Development Process", Formal Methods Workshop, Halifax, Nova Scotia, 1989.

¹⁵ Richard W. Selby, Victor R. Basili, and F. Terry Baker, "Cleanroom Software Development: An Empirical Evaluation", IEEE Transactions on Software Engineering, Vol SE-13, No 9, September 1987, p. 1028.

to compose statistical tests which can be randomly selected. As a result of this type of testing, the concept of mean-time-to-failure becomes germane to the software development environment. This concept would allow the vendors of software to place written reliability estimates within their product warranties.¹⁶ Additionally, statistical testing tends to find errors in the order of their failure rates and provides a scientific basis for making statistical inferences.

A key feature of this methodology is the introduction of the "design increment". Instead of viewing software development as a sequence of design, implementation, and testing, the Cleanroom Methodology views the software project as a collection of individual increments, each one of which is fully developed and certified before it is incorporated into the overall project design. The complete collection of certified increments comprises the fully functional design project.¹⁷

¹⁶ Bonnie Collett Sexton, "Statistical Testing of Software", Masters Thesis, Department of Computer Science, The University of Tennessee, Knoxville, December 1988, p. 71

¹⁷ Selby, et al, p. 1027.

2.6. CLEANROOM SOFTWARE DEVELOPMENT LIFE CYCLE

The Cleanroom software development life cycle requires that the software program be composed of one or more Releases, ($R_1, R_2, \dots, R_i$) which are formed from one or more Increments¹⁸ ($I_1, I_2, \dots, I_i$). Appendix A¹⁹ depicts the life cycle as the following series of milestones:

1. Problem Analysis and Specification;
2. Box Structure Analysis and Design;
3. Functional Verification of Box Description Language (BDL);
4. Statistical Testing of Common Services; and
5. Statistical Testing for Certification.

Steps 2 through 5 are repeated for each increment in the release. When the complete collection of increments comprising the current release has been certified, the software is released to the field. If more releases are planned, the same cycle begins for the next release.

2.7. PROBLEM ANALYSIS AND SPECIFICATION

Problem analysis and specification requires communication with and feedback from the intended users of the system. Typically, the user does not have a full definition of the

¹⁸ A. Currit, M. Dyer, H.D. Mills, "Certifying the Reliability of Software", IEEE Transactions on Software Engineering, SE-12, No 1, January 1986, p. 3.

¹⁹ Poore, p. 13.

requirements for the system when the programming team is first tasked to provide a solution. It is only through interaction with the user that a clear definition of system requirements emerges. The specification, the document detailing the requirements, is "stable" when both the user and the programming team agree that the specification correctly interprets the user's requirements.

2.8. BOX STRUCTURE ANALYSIS AND DESIGN

When the specification is stable, the development team begins the process of Box Structure analysis and design. Here, the first task is to decompose the specification into a series of well-defined releases. A system which automates the accounting needs of a small retailer might, for example, be divided into five releases consisting of inventory control, accounts receivable, accounts payable, payroll, and general ledger. Inventory control, the first release, would be the subject of continued box analysis and design by being further decomposed into a series of increments: Inventory Ledger, Sales Interface, Purchasing Interface, and Managerial Reports. The focus now shifts to the Inventory Ledger Increment. The development team partitions this increment using the Cleanroom procedures in a continuing top down manner until the BDL produced by the process can be translated into the target language.

2.9. TESTING TEAM ACTIVITY

Simultaneously with and independently of the programmers, the testing team can begin its own discussions concerning the user environment in which this increment must function. Because the testing team is viewing the specification document from a different perspective than that of the development team, often potential discrepancies or ambiguities in the specification can be identified. The testing team communicates the discovery of such problems to the development team. Once informed, the developers can contact the user and discuss possible modifications to resolve the problem.

2.10. FEEDBACK

The feedback loop provided between the developers and the testers is a unique and important feature of the life cycle. As a result of this feedback loop, problem areas can be identified early in the design process. In other methodologies, such difficulties are frequently unnoticed until the entire project is coded and in the debugging phase. Furthermore, because the goal of debugging is to "get the code to work," the problem identified by the Cleanroom testing team would likely not surface in other methodologies until the product is delivered to the user. In other words, statistical certification requires the construction of tests that simulate the working environment of the increment. Development of such tests requires the testing team to focus on the "real world"

use of the product. It is this focus which provides the early recognition of specification error or ambiguity.

2.11. TARGET LANGUAGE transliteration

Upon completion of BDL functional verification, the increment is transliterated into the target language, and examined again to ensure that the transliteration is accurate. The increment is then delivered to testing for statistical certification.

2.12. TESTING ENVIRONMENT

The testers submit the increment to a statistically valid sample of tests until the increment passes the number of consecutive tests required for certification, until a satisfactory mean-time-to-failure is achieved, or until the increment fails a certain number of tests. Failure causes the increment to be returned to the developers for engineering changes, and the subsequently revised increment is tested again until certification or the next failure. Cleanroom standards define a maximum failure rate of five errors per thousand lines of code. For example, if an increment contains ten thousand lines of code, up to fifty errors are tolerated prior to certification of the increment. However, on the fifty-first failure, the increment no longer meets Cleanroom standards and is returned to the developers for complete redesign. Sexton calculates that to certify an increment as 98% reliable with 90% confidence, the increment must pass one

hundred consecutive tests.²⁰ By necessity, therefore, the testing team must maintain a sufficiently large pool of test cases so as to anticipate the worst testing scenario for the increment.

2.13. COMMON SERVICE TESTING

Depending upon the nature of the product and the desire of management, the testers could choose to extract common services (a Black Box called by two or more parents) from the increment and test them individually. For example, the certified common services could be cataloged into a repository of reusable modules, as required by the Department of Defense STARS project²¹ and would be available for the next programming project.

2.14. INCREMENT FEEDBACK

Design of increment($i + 1$) should not begin until increment(i) is certified. However increment($i + 1$) will benefit directly from the field experience provided by increment(i) as feedback into its specification. Referring to the above example, assume that the Inventory Ledger Increment has been in the field for several months while the user is awaiting the timely delivery of the second increment, the Sales Interface.

²⁰Sexton, p. 55

²¹Poore, p. 11.

Further, assume that the original specification called for a batch processing system in which inventory records would be updated daily after the close of business. As the proprietor becomes more comfortable with this new form of inventory control, he or she realizes that in order to "fine tune" the inventory, the batch process simply will not be adequate, because inventory figures are at best a day old. The proprietor concludes that a real time inventory control system will be required in order to achieve cost-effective inventory management.

This change in specifications brought about by the field use of increment one is easily handled by the development team. The BDL for the box "update transaction file" and for the box "update master file" is modified to support real time update of the master file. The temporary transaction file is eliminated from the design because it is no longer required. Increment 1 is then delivered to testing for recertification. BDL for the Sales Increment is restructured so that the quantity "inventory available for sale" can be obtained directly from the master file instead of being computed as the difference of "inventory on hand" from the master file and "inventory sold" from the transaction file. Delivery of the revised first increment is delayed until the increment can be certified in conjunction with the Sales Interface. The two increment package replaces the batch processing design of

increment one, and field use begins to provide feedback during the development of increment three, the Purchases Interface.

2.15. RELEASE FEEDBACK

The feedback loop (provided to the increments through field use) scales up and is also available between the current release in the field and the release under development. This loop ensures that increments are not developed within a vacuum; each increment delivered to the user reflects the cumulative clarification of the specification that only field use can provide.

3. BOX STRUCTURE BDL GENERATOR

3.1. MOTIVATION

The basic tools required by Cleanroom Methodology are, quite simply, pencil, paper, and text editor. However, the implementation of the methodology is up to the group that chooses to develop software under this discipline.

In an effort to provide a standard implementation, The University of Tennessee Cleanroom team has developed The Box Structure BDL Generator, a computer program which aids software design by providing a clerical tool and guidance for stepwise refinement. The rationale was: while developing a computer program which would support the methodology, the team should simultaneously develop a clear and precise understanding of stepwise refinement as it is used in the methodology. The motivation behind the development of the Box Generator, then, is five fold:

1. to provide a tool that reinforces stepwise refinement of Box Structure design by imposing the methodology on the users of the tool;
2. to provide an efficient means to write and edit design;
3. to provide an efficient means to print the BDL, either for the entire project, or for a single module;
4. to provide a working tool which can be maintained and enhanced by other graduate students as data from "field use" is gathered, interpreted and acted upon; and

5. to provide life cycle data.

3.1.1. REINFORCE STEPWISE REFINEMENT

The tool reinforces stepwise refinement by requiring the user to develop the system in top down fashion, beginning with the Top Level Box, and designing successive boxes in a breadth first order. Additionally, within each box, the Black Box must be developed before the State Box, and the State Box must be developed prior to the Clear Box. Release 1 of the Box Generator defines "developed" very loosely, requiring the user to visit each of the screens necessary to produce a complete box in a specific order before allowing the user free access to any screen in the system. It is the intention that subsequent versions will impose strict standards on the definition of "developed" and will require, for example, an explicit matching of stimuli to responses prior to the composition of the Black Box Transition Rule. The composition of the rule, then, would require explicit text describing the relationship of each stimulus-response pair before the user could progress to the State Box screen sequences.

The Top Level Box is unique within the system because it is the sole box that is not created within the Clear Box of its parent. Therefore a special screen (Appendix B, page 81) was developed to capture the Top Level Box Name. Once the name has been defined the system steps to the "Stimulus List"

screen (Appendix B, page 82) which is the beginning screen for the development of all subsequent boxes. It is here that the user defines all the stimuli associated with the box. The user also has the opportunity to explicitly associate a type (integer, real, structure, etc.) to the stimulus and, if necessary, can associate a "type description" with the type. The type description is simply a text block which could, for example, detail the elementary items of the type "structure".

The user indicates completion of the stimulus list by striking F4. This keystroke signals the Box Generator to display the "Response List" screen (Appendix B, page 83) and the user proceeds through an identical process of naming, typing, and providing type descriptions, for each of the responses required by the box under development.

When the response list is complete the user describes the "Black Box Transition Rule" (Appendix B, page 84) in a free text entry screen. The Black Box Transition Rule should describe the relationship of each stimulus to its associated response(s). The union of each of these statements would completely describe the function of the box. Upon full development of the Black Box Transition Rule, the Black Box definition is complete and the user is then allowed to begin development of the State Box.

"The role of the state machine is to open up the Black Box description of a system one step by making its data visible."²² In this phase of development, the user is examining the stimulus history described in the Black Box, and capturing the essential data items required by the box to perform its function. Mills cites an excellent example in his description of a machine that computes a twelve month running average. From the Black Box point of view, the machine has responded to the stimulus "monthly average" since the continuous operation of the machine began, possibly years ago, by calculating a new, current running average. Obviously, the current "monthly average", i.e., the current stimulus, and the history of the previous eleven monthly averages are required to compute a twelve month running average. The State Box provides the mechanism to capture the eleven monthly averages, to combine that history with the current stimulus to produce the current running average, and finally to trap the current stimulus and update the eleven monthly averages. Additionally, one can now turn the machine off, because the essential history is contained within the State Box.²³

To implement this feature of the methodology, the Box Generator provides the "State Data List" screen (Appendix B,

²²Harlan D. Mills, Richard C. Linger, and Alan R. Hevner, Principles of Information Systems Analysis and Design (Orlando: Academic Press, Inc., 1986) p. 8.

²³Mills, Linger, and Hevner, p. 11.

page 85). In a manner identical to defining stimuli and responses, the user defines the "state data" required for the successful operation of the State Box.

Once the state data have been defined, the user can construct the "State Box Transition Rule" (Appendix B, page 86). This rule describes the operation of the State Box which accepts the stimulus pair, (current external stimulus, current internal state), and produces a response pair (external response, new internal state) which replaces the old state.²⁴ Completion of the State Box Transition Rule completes the development of the State Box and the user is now able to define "machine data" for the "Clear Box".

Machine data are those elements required by the Clear Box for its successful procedural operation. Machine data include, but are not limited to, loop counters, array indices, temporary variables, and return flags. It is important to remember that machine data are completely hidden from and have no relation to the stimulus history associated with the Black Box and captured within the State Box. Those histories concern themselves with the external operations of the system, while machine data are used by the Clear Box exclusively as internal control mechanisms governing its procedure.

²⁴ In the generator, Release 1 makes no check on the quality of the rule, simply allowing the user to describe the rule in a free text format.

The mechanism used to define stimuli, responses, and state data, is now employed in the creation of machine data (Appendix B, page 87). Each item of data can have associated with it a type and a type description as required by the user. After all machine data are defined, the user is ready to describe the "Clear Box Procedure".

The Clear Box, as the name suggest, opens up the state machine [box] description of a system one more step in an internal view that describes the system processing of the stimulus and state [stored data]. The processing is described in terms of the three control constructs of structured programming, namely, sequence, alternation, and iteration, and a concurrent structure.²⁵

A concurrent structure differs from a sequence construct in the following manner: a sequence construct requires that procedure "a" be accomplished prior to the execution of procedure "b", while a concurrent structure requires that the order of execution of the procedures "a" and "b" will not affect the procedural correctness of the Clear Box.

Cleanroom Methodology has developmental conventions for the Black Box and State Box Transition Rules, but these conventions are not enforced by the Box Generator. However, the Box Generator does enforce rigid restrictions within the development of the Clear Box Procedure. Only the three control constructs and the concurrent structure are permitted as outer

²⁵Mills, Linger, and Hevner, p. 8.

syntax and they are made available through a "pick list" window. When the program developer requires an "if/then/else/fi" construct, for example, the developer selects it from the pick list and that structure is displayed in its entirety in the procedure window (Appendix B, page 88). The developer can then define the inner syntax associated with the selected construct. Deletion of a construct can only be accomplished through the use of a function key, ensuring that the entire contents of the construct are deleted. Control of the outer syntax in this fashion safeguards its integrity. Therefore, the restrictions of the Clear Box Procedure prevent the possibility that the program developer will construct flawed outer syntax by omitting an integral portion of a syntax construct such as a closing fi in an if/then/else/fi structure.

Additionally, Clear Box Procedure provides checks on the use of Black Box names, requiring that each name be unique in the system. This check is desirable because a duplicate name implies that the user desires to change a Black Box to a common service or that the user has made an error. If a user does enter a duplicate box name, the Clear Box Procedure queries the user's intentions and acts accordingly.

Upon completion of the Clear Box Procedure, the program developer is presented with the "State Migration Information

Screen". Mills introduces the concept of "state migration" as follows:

A Clear Box expansion of an original Black Box and its state machine into several other Black Boxes and their state machines already implies a distribution of state parts across the hierarchy. The state of the entire box structure is made up of the states of the state machines defined throughout the hierarchy. However first conceived, further analysis and study may reveal possible improvements to the distribution of state parts across the hierarchy of the box structure.²⁶

The Box Generator reminds the user to examine the procedure just developed and determine the need for state migration.²⁷ Once the user is informed that the possibility of state migration should be investigated, the design of the Clear Box is complete.

When the design of the Clear Box is complete, the Box Generator does a breadth first search for incomplete boxes. If an incomplete box is found, the program developer is returned to the "Stimulus List" for that box. The design process is complete when all Black Boxes created through the Clear Box are fully defined.

²⁶ Mills, Linger, and Hevner, p. 234.

²⁷ Later versions could possibly do some analysis of the structures developed thus far and automatically perform state migration.

3.1.2. BDL EDITING

Efficient editing of the design modules, the second motivation suggested for the tool, is achieved through the Box Menu (a popup utility menu) selection, "EDIT". This feature will allow the user to access any design screen previously visited by highlighting the screen title with the light bar and then pressing the carriage return. The screen selected is presented to the user together with the germane information from the current node in memory. The user is then free to edit the information as needed.

3.1.3. BDL PRINTING

Printed BDL can be obtained for a single node through the Box Menu selection, "PRINT", or if the user desires, the entire project can be printed through the Project Menu selection "PRINT". The printed BDL is formatted to enhance the group activities of design review and functional verification, by printing the node number which identifies the node position relative to the project hierarchical tree, by printing line numbers for every line of BDL, and also through the use of indentation of outer syntax which provides a visual clue to the proper nesting of control structures.

3.1.4. STIMULATE FURTHER RESEARCH

The fourth motivation, to provide a base from which other students can continue design work and research, is already a

reality. A Cleanroom team of graduate students will begin work in the fall of 1989 on Release 2 of the Box Generator, and another graduate student is beginning research into the special properties of statistical testing for single-user keyboard-screen interaction software.

3.1.5. LIFE CYCLE DATA

The final motivation, to gather and analyze life cycle data, will be discussed in chapter 6.

3.2. GENERAL DESCRIPTION

The Box Generator is an interactive software development tool designed for the IBM PC and compatibles using MSDOS version 3.3 and higher, or the PS/2 operating systems. The Box Generator uses eight separate interactive screens which control the design sequence of functional modules, called boxes. Additionally, several utilities are provided which accomplish file updates on disk, BDL printing, editing of previous work, selection of specific boxes for continued development, renaming of boxes, deletion of boxes, naming of new types, composition of type descriptions, and normal program exit.

3.2.1. DESIGN SEQUENCE

The Box Generator imposes the Mills design sequence (previously outlined) in the following manner:

EXPANSION PROCESS	BOX GENERATOR SCREEN
A. Define the Black Box. 1. Define black-box stimuli. Determine all possible stimuli for the Black Box. 2. Define black-box behavior. For each possible stimulus, determine its complete response in terms of its stimulus history.	Stimulus List Stimulus list is composed. A Type may be assigned to each stimulus. A type description may be assigned to each type. Response List Response list is composed. A Type may be assigned to each response. A type description may be assigned to each type. Black Box Transition Rule A free text entry screen in which the program developer describes the relationship of stimulus to response.
B. Design the State Box. 3. Discover state data requirements. For each response to be calculated, encapsulate its stimulus history into a state data requirement. 4. Define the state. Select a subset of the required state data items to encapsulate stimulus histories. 5. Design the State Box. For the selected state, determine the internal Black Box required for the State Box. 6. Verify the State Box. Verify the correctness of the State Box with respect to the required black-box behavior.	State Data List State data list is composed. A Type may be assigned to each state datum. A type description may be assigned to each type. (invention process) (invention process) State Box Transition Rule A free text entry screen in which the program developer describes the operation of the box making reference to state data rather than stimulus history.

C. Design the Clear Box.

7. Discover state data accesses.
For each item of state data and each possible stimulus, determine all possible accesses of the item.

State Migration

An informational screen reminding the developer to examine the design to discover opportunity for state migration.

8. Define the data abstractions.
Organize state data into data abstractions for effective access.

(invention process)

9. Design the Clear Box. Define sequential or concurrent uses of the data abstractions defined to replace the internal Black Box of the State Box.

(invention process)

10. Verify the Clear Box. Verify the correctness of the Clear Box with respect to the State Box behavior.

Machine Data List

Machine data list is composed. A type may be assigned to each machine datum. A type description may be assigned to each type.

Clear Box Procedure

Full unfolding of the box internal operation. Outer syntax is controlled. Text format is automatically nested as required. Inner syntax is free text.

D. Continue the process.

11. Repeat the stepwise expansion until design completion. For each new data abstraction, repeat steps 1-10 until suitable data and program specifications are reached.

3.2.2. WELL DEFINED SPECIFICATION DEMANDED

The methodology demands a well defined specification as a foundation. This specification must provide both the developers of the software and the testers of the system enough information so that the two tasks can proceed independently. The Box Generator's specification met that expectation by providing a written description of requirements characterizing the performance of the tool through three design releases. Additionally, a "story book" was created

which described both the appearance of the eighteen operational screens and five informational screens which compose the Box Generator, and the correct response to every keystroke on each screen.²⁸

3.2.3. BOX GENERATOR DEVELOPMENT

Sections 3.2.3.1 through 3.2.3.10 chronicle the development of the Box Generator itself. The process is described from specification through functional verification and is presented as an example of how the methodology imposes a structure on the design process which, when followed correctly, produces the high quality product.

3.2.3.1. SPECIFICATION DEVELOPMENT

The development of the specification was an arduous task, requiring almost 800 person hours to complete. The first phase of the specification development considered only the written description of the three releases. The design team quickly grasped the basic requirement of the system, to produce printed BDL. On preliminary inspection, however, that task seemed too mundane, so the team began to enhance the system by adding more function to the prototype. For example, it seemed perfectly reasonable to expect that if BDL could be

²⁸ Charles C. Senn, Carmen Trammell, Hialong Mao, Randy Baker, Norman Sims, "Box Generator Specification", Technical Report, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1989.

printed on paper, then it should also be available on the screen. Also, Mills had, throughout his text, used extensive diagrams called Box Description Graphics²⁹ to further describe the BDL. Some members of the group believed that the Box Generator should be able to analyze the BDL for any box contained in the system and produce the correct Box Description Graphic.

Additionally, some members wanted a "box score" should be maintained on each box, tracking the status of certain well defined events and attributes of the box:

1. State Box verification status
2. Clear Box verification status
3. referential transparency status
4. transaction closure
5. common service indicator
6. state migration status
7. functional verification status.

Some thought that the system should also extract metrics directly from the Clear Box Procedure. A display with the appearance of a thermometer then could be developed which would indicate the status of Clear Box complexity. As the

²⁹ Mills, Linger, Bevner, p. 14.

Clear Box Procedure became more complicated the "temperature" in the thermometer would begin to rise, giving a visual clue as to whether the program developer should redesign this particular Clear Box Procedure.

These and other suggestions were debated and what emerged was an outline for a tool which could be developed in three releases: the first would be primarily clerical in nature imposing upon the user a relatively primitive understanding of the methodology; the next release would capture management and research data during the development of a project; and finally, the third release would impose a more comprehensive understanding of the methodology. Four-hundred-fifty person hours were required to develop the specification from the idea that "we need a tool to print BDL" to the orderly progression of these releases.

3.2.3.2. TOP LEVEL BOX DEVELOPMENT

With the release requirements profiled, the team changed its orientation and began to focus on the development of the Top Level Box for Release 1 of the system.³⁰ It was during this time that the team began to get comfortable with the concepts of Box Structure design, stimulus-response mapping, transition rules, and other abstractions associated with the

³⁰ Charles C. Senn, Carmen Trammell, Hialong Mao, Norman Sims, "BDL for the Box Generator", Technical Report, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1989.

methodology. A crucial decision reached during this phase of the Box Generator's development was the realization that at the top level, the team could completely describe the response of the tool in a single phrase, "to produce printed box description language". This was remarkable, because all of the complexity associated with the user interface, in whatever form that interface took on, could now be deferred to a lower level. In other words, the team had achieved referential transparency with regard to the user interface, a concept "that to reference an object by name gives the entire effect of the object itself"³¹.

At this point the Top Level Box was far from complete. The next major task was to invent the state data, which at the top level is the design of all global data structures required for system operation. The team settled upon an intricate structure of dynamically allocated linked lists whose entire contents would fully describe a single box of the design and would compose a single variable length record which would be stored on disk in a "master file".

With the state data fully defined, the team focused on the Clear Box Procedure. This procedure went through fifteen major design revisions prior to its acceptance. The primary

³¹Mills, Linger, and Hevner, p. 14.

design difficulty was unnecessary complexity at the top level. Because the stimulus to this box had been described as two vectors consisting of "Project Information" and "Box Information", the Clear Box Procedure initially attempted to deal with the vectors separately. Furthermore, this two-vectored design required the addition of a loop controller in the top level Clear Box Procedure. Intuitively, the team felt that the top level should consist of a single sequence construct, precluding the necessity of a loop controller. The dilemma was resolved when the group realized that, although the stimuli had been abstractly represented as project and box information, in reality there was a single stimulus to the user interface, namely the keystroke. The team could then create the functionally cohesive Black Box "Process Information" which had the responsibility of producing the correct response for each keystroke stimulus. Furthermore, the loop control would reside at a lower level within this box's Clear Box, thus allowing the top level Clear Box Procedure to be described in a single sequence construct.

3.2.3.3. "STORY BOOK" DEVELOPMENT

Satisfied with the top level, the team attempted development of the lower level Black Boxes created within the top level Clear Box. However, it was determined that further design would be futile without a "story book" describing the user

interface. The team therefore turned its efforts towards that task.

Creation of the story book first required determining the complete configuration of all screens necessary for the a developer to effectively use the Box Generator in designing programs. A system composed of twenty-one operational screens and eight informational screens emerged. The screens are grouped in the following task orientation categories given in Table I. After the sequence of screens was determined, keystrokes were mapped to screen responses. Each screen represented a visual finite automaton in a specific state, and when the automaton detected a new stimulus it would transition to the next correct state. Transaction closure (a condition satisfied when the transactions are sufficient to generate all state data, and the state data are sufficient to generate all the transactions)³² was achieved on each screen by providing a NULL response to any keystroke that was not explicitly mapped to the screen. The value of using this technique to define interactive screens surfaced during the development of the lower level Black Boxes, where each keystroke became a functionally cohesive module whose transition rule was completely described by the specification.

³²Mills, Linger, and Hevner, p. 105.

Table I. Box Generator Screen Task Assignment

BLACK BOX	STATE BOX	CLEAR BOX	UTILITIES
Top Level Box Name	State Data List	Machine Data List	Welcome
Stimulus List	State Box	Clear Box Procedure	New Type Name
Response List	Transition Rule	IN List Variable Selection	Type Detail
Stimulus/Response Mapping		OUT List Variable Selection	Select Common Service
Black Box		State Migration	Project Menu
Transition Rule		Clear Box BDL	Saving Work
			Printing in Progress
			Save?
			Exit
			Box Menu
			Select
			Edit
			Rename
			Deleting
			Undeleting
			Display BDL

3.2.3.4. SPECIFICATION REVIEW

The 350 person hours required to develop the specification culminated in a review of the eighty page document. During the review a decision was made to study the specification once again with the goal of simplifying Release 1. The release had acquired several features which enhanced the tool but were extraneous to the central purposes. For example, two modes of editing, insert and overstrike, were proposed. Because the details of the editor had not been specified for the release, the insert mode could be eliminated from the system design. Several screen operations were also removed without detracting from the specification. By the end of this "pruning session" three of the operational screens had been simplified, and five screens as well as six utility-function key operations were deferred without compromise to the Release 1 specification. With the specification now stable, the design team turned its attention to the development of the lower level Black Boxes.

3.2.3.5. LOWER LEVEL BDL DEVELOPMENT

The "keep it simple" spirit that took hold during the specification review invaded the Top Level Boxas well. The team decided to convert the data structures from linked lists and variable length records to arrays and fixed length records. This eliminated the need for a b-tree file manager which is required for fast access of variable length records. Simple pointer arithmetic was all that would be required for

record retrieval from disk. This change was not without consequence. The user is now limited to a fixed length text buffer in which to describe the box. Although this buffer is adequate for Release 1, linked lists and variable length records might appear in a later release.

Recall that the stimuli project information and box information were the major components of the stimulus vector for the Black Box, Process Information. Project information and box information were abstract concepts whose real counterparts were keystrokes provided by the user. The specifications began to drive the development of Process Information because it was readily apparent that the response to a keystroke was dependent upon the screen in which the keystroke was actuated. For example, a carriage return on the Top Level Box Name screen has a different response than a carriage return on the Stimulus List screen. The design of Process Information became simplicity in itself. It would be composed of a single case statement whose parts would represent each of the operational screens of the system. Within each part, three additional Black Boxes would be required to fully operate the screen: a box to draw the screen; a box to load existing information which could be displayed on the screen; and a controller which would analyze the incoming keystroke and produce the correct response. The case statement would be enclosed in a "while loop" which would

terminate upon the user's request to exit the program, or upon a system error. Termination of the while loop would return control to the top level where a graceful exit of the program would be executed.

During the design review of the BDL, it became clear that the team had made a correct decision in electing to expend the effort required to produce the detailed specification. The specification was forcing the design and the resulting design was simple. A complete Box Structure development of the first operational screen was produced. Its purpose was to serve as a model which could be used by the other team members in the development of the screen designs for which each was responsible. The team felt confident enough in the plan to move from meticulous design review of the high level Box Structures to a BDL production phase. In that phase the individual members of the team could develop the specialized screen displays, loaders, and controllers, while still maintaining contact with each other at a much more informal level. This contact was crucial in the management and coordination of the interface between the various modules being developed.

3.2.3.6. PROBLEMS SURFACE

It was during this phase of development that the sophistication required by some of the screen controllers

began to surface. The Clear Box Procedure, which provides, in part, for the selection and deletion of BDL control constructs, proved complicated. Implementation of Clear Box features required that each line of text in the Clear Box Procedure be flagged with an appropriate symbol which could be used to determine:

1. whether insertions or deletions of text were permissible;
2. whether the line was a reserved word which would allow deletion of the control construct; or
3. whether the line contained a Black Box name, that, when changed, would require a test to determine its uniqueness.

The interface, or calling sequence between modules, was also becoming intricate because the team had elected not to use any global variables. The attributes, screen display, Project Information, Box Information and pointers to files were included in argument lists in addition to the stimuli and responses that would actually be used in the target language function call. These additional stimuli and responses made the examination of associated calling sequences between modules long, involved and error-prone.

Another major problem was the attempt to place the screen display, screen loaders, and screen controllers (for functions invoked through either the Project Menu or the Box Menu)

within the Process Information case statement. Unlike operational screens, which are visited in a particular order, the Project Menu and Box Menu functions are available on almost all screens. The original concept was that selection of a Box Menu or Project Menu item would simply change the value of "scrn_number" (the machine data variable that was evaluated upon entry to the case statement within Process Information). The new scrn_number would then pass control to the appropriate part of the case statement. The problem was that the specifications required the program developer to be returned to the same screen from which the menu item was selected. But, if this were done, information would have been lost since scrn_number, which previously held the correct value, now holds the value of the actual menu item screen. An attempted solution required three additional machine data variables, prev_scrn, return_row, and return_col, to hold the history necessary to return to the correct screen when the menu operation was completed. However, this proposal compromised the simplicity designed in Process Information.

3.2.3.7. "PROCESS INFORMATION" REDESIGN

The design team conducted a formal review of Process Information in an effort to retain its simplicity. An excellent solution was quickly drafted. Developing the Project Menu and the Box Menu as common service routines and placing those routines at a lower level than the operational

screens solved the problem. Return to the correct screen was guaranteed through the normal interactions of calling subroutines.

Eleven hundred person hours were required to mass produce the remaining BDL and thus complete the Box Generator's design. The team prepared for functional verification, the next phase of the methodology, by participating in a review of the formal methods for proving program correctness.

3.2.3.8. FUNCTIONAL VERIFICATION PROCEDURES

Much of the BDL had been prepared individually by members of the team without the customary group review. Consequently, the team decided a BDL "familiarization" review was in order prior to functional verification. In the 67 modules sampled for which data were kept a total of 136 errors were found. The relatively poor quality of the BDL could be attributed directly to the decision to mass produce the remaining BDL, without peer evaluation.

Functional verification found an additional 45 errors distributed in 40 boxes. Additionally, as a result of both the group review and functional verification, 62 boxes were optimized, resulting in a code reduction of approximately 300 lines or almost 5% of total code. Chapter 6 categorizes the errors found in both design review and functional

verification. Functional verification was now complete and the team began the transliteration of the BDL into the target language, 'C'.

3.2.3.9. CODING AND CODE REVIEW

Transliteration was uneventful, requiring 120 person hours to produce the 7200 lines of C-code. Once transliteration is complete, the expectation is that the code review should simply be a verification that (1) the logic of the BDL is not altered, and (2) the target language is syntactically correct. What, in fact, happened, was that the rigidity of the target language, 'C', made clear many logic errors that had escaped the team's attention in the functional verification. The pseudo language had not conveyed the intricate pointer relationships inherent in 'C' and this lack of rigidity accounted for many logic problems within the transliterated code. The code review resulted in many more changes to the code than were expected. C-code review showed that a lack of attention to detail had permeated the entire functional verification process. Had time permitted, another code review would have been desirable, to verify that no more changes were required.

4. BOX GENERATOR DESIGN

4.1. OVERVIEW

This chapter highlights three major design aspects of the Box Generator. Design Philosophy concerns the role of "user friendliness" in the degree of sophistication incorporated in the tool. Design Decision discusses the difficulty involved in internal data structures. And, lastly, Design Itself provides statistics on the design hierarchy.

4.2. DESIGN PHILOSOPHY

From discussions which focused on the nature and function for each of the Box Generator's three releases, a design philosophy began to emerge for Release 1. That the tool should be a clerical one was clear, but the level of refinement was a matter of hot debate. Three areas of the design received the most attention. First the sophistication of the editor, second, the role that lexical analysis would play in the Clear Box Procedure, and third, the role "user friendliness " would play in the tool development.

The team desired an editor that would incorporate features such as:

1. insert or overstrike mode selection;
2. word wrapping;
3. word and line deletion;
4. page up or page down cursor movement.

The team wanted the program developer to have a tool which would not only aid in the production of Box Structures, but would also feel comfortable and natural. "User friendliness" dominated the text editor discussions. No member of the team could cite an example of a commercial editor that did not have two edit modes, therefore, it was assumed that for the Box Generator to be acceptable it must have an insert-overstrike capability. Similarly, for each feature contemplated for the editor, a strong opinion was voiced that the tool would be unacceptable without the feature.

User friendliness was also the impetus behind the desire for lexical analysis of the BDL inner syntax. The rationale was that for the tool to truly manage the outer syntax control constructs, lexical analysis must be used on the inner syntax to ensure that the developer had not placed control constructs in the free text inner syntax. Additional valid uses for lexical analysis occur in the Clear Box Procedure. For example, Black Box argument lists (IN: OUT: lists) could be validated against information contained in the stimulus, response, state data, and machine data lists associated with the function.

Information screens and pick-list screens were designed explicitly to enhance the tool's friendliness. For example, when the developer, while working on the Clear Box Procedure

screen, placed the cursor within a particular control construct which would require access to the stimulus list or response list, the list would be made available in the form of a pick-list-window.

The minority opinion held that this version of the tool was a prototype and there was no need set by the specification to implement the more advanced features. What the specification required and more importantly what the Cleanroom Increment concept demanded was the tool be built from a basic, well defined, functioning core. With the core in place additional increments implementing user friendly features could be incorporated into later releases.

The debate over sophistication of features lasted throughout the development of the "story book". It was not until the review of the completed "story book" in light of the specification that team members realized a reevaluation was in order. As one team member observed, "We have been engaging in software development out of control."

4.3. DESIGN DECISION

The most elusive design decision concerned the data structure. The structure must contain all information for a box under development. Initially, a design of linked lists was thought to be best. The advantage of linked lists was the

information concerning any particular box would be efficiently stored in memory. Complementing efficient use of memory would be the efficient use of disk space through variable length records. This design was discarded because of a growing concern that the project was expanding beyond the team's ability to complete on schedule. Elimination of the linked lists and variable length record strategy eliminated the requirement to develop a b-tree file manager, saving approximately 1000 lines of code.

Fixed length records are simple to manage. The size of each fixed length record, initially calculated at 13,000 bytes, was large, but encountered no hardware limitation because of the PC's 64,000 byte stack-heap capacity. During the design review, a need arose to recalculate the size of the data structure. The true record size proved to be 125,000 bytes. In other words, a single record could not be contained in memory. A redesign of the problem area in the data structure and modification of an existing working file resolved the problem. The final data structure size of 11,000 bytes was well within the design parameters.

4.4. THE DESIGN ITSELF

When viewed as an hierarchical tree, the Box Structure design is of height (maximum level of the tree) 10, and span (maximum

number of children of any parent) 22. Appendix C contains the Box Generator hierarchical tree.

5. STATISTICAL TEST DEVELOPMENT

5.1. PRELIMINARIES

Statistical testing leads to an independent certification of the correctness of a software product. The certification goal established for the Box Generator was 0.90 reliability with 90% confidence.

The testing team could not discover documentation on statistical testing of interactive software. Therefore, no precedent for the development of test cases or conducting a test was available. An oracle, human or mechanical, decides if the outcome for a test is in agreement with the specification (pass) or not (failure). The team elected to use a human oracle to determine from the specification what would be considered a correct response to each input during the Box Generator testing.

5.2. SIMULATE THE ENVIRONMENT

The specification led the testers to a point of view similar to that held by the developers, namely, that the Box Generator was a finite automaton with transitions from a current state to a new state based on the current stimulus. The usage environment forces order on the relationship between the current stimulus and the previous stimulus. In other words, the selection of a stimulus (keystroke) is determined by two

factors: (1) the specific screen on which that keystroke took place, and (2) the history of previous keystrokes on that screen. The Box Generator's usage environment would depend upon the skill level of the operator. A novice user of the system would produce a keystroke history very different from an experienced user, even if both were entering or editing the same BDL document. Since all prospective users for the tool will be students becoming familiar with the concepts of stepwise refinement in Box Structures, the decision was made to simulate a novice environment for the Box Generator and to develop statistical tests accordingly.

Future releases will incorporate a keystroke log which will be used to accurately model the user history. Such a history will be quite useful in the determination of accurate statistical tests. Currently, the availability of BDL samples is limited. As a result, the testers elected to use the BDL that was being written by the design team for the Box Generator as its source for certification testing. By studying the changes in the BDL from one revision to the next, the testing team developed a "novice usage profile" which would be the model for statistical test development.

The testing team determined that the usage environment could be correctly modeled in three dimensions: screen, previous keystroke, and current keystroke. The question could be

posed: "Given that the previous keystroke is x, what is the probability that the current keystroke is y? But the focus on "keystroke" seemed myopic, so the testing team coined "event" to mean a significant activity on the current screen. For example, if the program developer defined a stimulus name, "start_pulse", that definition would be termed a single event, as opposed to a series of eleven keystroke stimuli. The concept of event was substituted for keystroke in the model, making the model more nearly an expression of the user environment. Screen transition matrices were developed for each of the Box Generator's operational screens. Appendix E contains the matrix developed for the Stimulus List Screen found in Appendix B, page 82.

5.3. TEST DEVELOPMENT

The test team determined the number of test cases needed to certify the Box Generator. The Box Generator code contained 7200 lines. Since the standard set for the Box Generator was three errors per 1000 lines of code, the Box Generator could meet the standard while containing twenty-two errors. Furthermore, in order to be certified 0.90 reliable with 90% confidence, the Box Generator must pass twenty-one consecutive tests. Therefore, the number of test cases required to demonstrate success could be as many as twenty-one (tests) times twenty-two (errors) or 462 test cases having an average of 12 events per test case. This volume of test cases led the

testers to devise an Ada program (also developed using Cleanroom) to generate test cases. This Ada program is completely independent of the Box Generator code (not automated testing) and as a result could not confound the experiment. A member of the testing team designated as the human oracle estimated the test cases and determined the proper response for each event based upon the specification. Test case events were randomly generated and manually keyed to the Box Generator. Appendix F contains a sample test case.

5.4. TESTING PROCEDURES

The Box Generator testing began with the processing of the randomly generated sequence of 462 test cases. Each event was executed on the Box Generator and the resulting response was classified as either pass or fail by the human oracle. Event execution continued until an event (and therefore a test) failed, or until the Box Generator passed twenty-one consecutive tests (approximately 250 events). A failure caused testing to stop and the Box Generator code was returned to the design team. The design team made engineering changes required to fix the fault, and returned the code to the testers. The testers began the administration of tests from the original event and repeated all events in order through the event which caused the failure. The effect of repeating the previous events was to bring the Box Generator to the exact state in which the failure occurred and pass the event

which before had caused failure. Testing proceeded with the number of consecutive tests passed reset to zero. Throughout this process, the test results were entered into a program which calculates the mean-time-to-failure estimates based on the Certification Model. In general, it is this mean-time-to-failure estimate which is the basis for bringing the overall software development process under statistical process control.

6. LIFE CYCLE DATA

6.1. OVERVIEW

Several sets of data were maintained by the team during the Box Generator project:

1. time required for each development activity
2. classification of modifications made in the design review
3. classification of changes made in functional verification classification
4. classification of changes made in code review
5. testing results.

6.2. TIME LOG

The time log presented below details the hours required in the evolution of the Box Generator from specification through code review.

ACTIVITY	NUMBER OF HOURS	PERCENTAGE OF TOTAL HOURS
Specification	755	24%
Design	1350	43%
Design Review	530	17%
Functional Verification	220	7%
Code	125	4%
Code Review	155	5%

Figure 1 presents a pie chart depicting the same information as the time log. In general, the total time required for Cleanroom project completion does not exceed that required by other methodologies. Note, however, the percentage of time required by Cleanroom for project design and verification (67% for the Box Generator excluding specification). It is clear management must expect a redistribution of time and effort with the implementation of Cleanroom methodology.

6.3. MODIFICATION CLASSIFICATION

Data on modifications was grouped into three basic classes:

1. Commission
2. Omission
3. Optimization.

Optimization proved to be a powerful byproduct of the group review process. Modification data on optimization indicates the possibility of significant reduction in module length and complexity. See Appendix C for an itemized listing of optimization modifications.

The commission and omission classifications (as well as the subcategories: initialization, computation, control, interface, data, and cosmetic) were established by Basili and

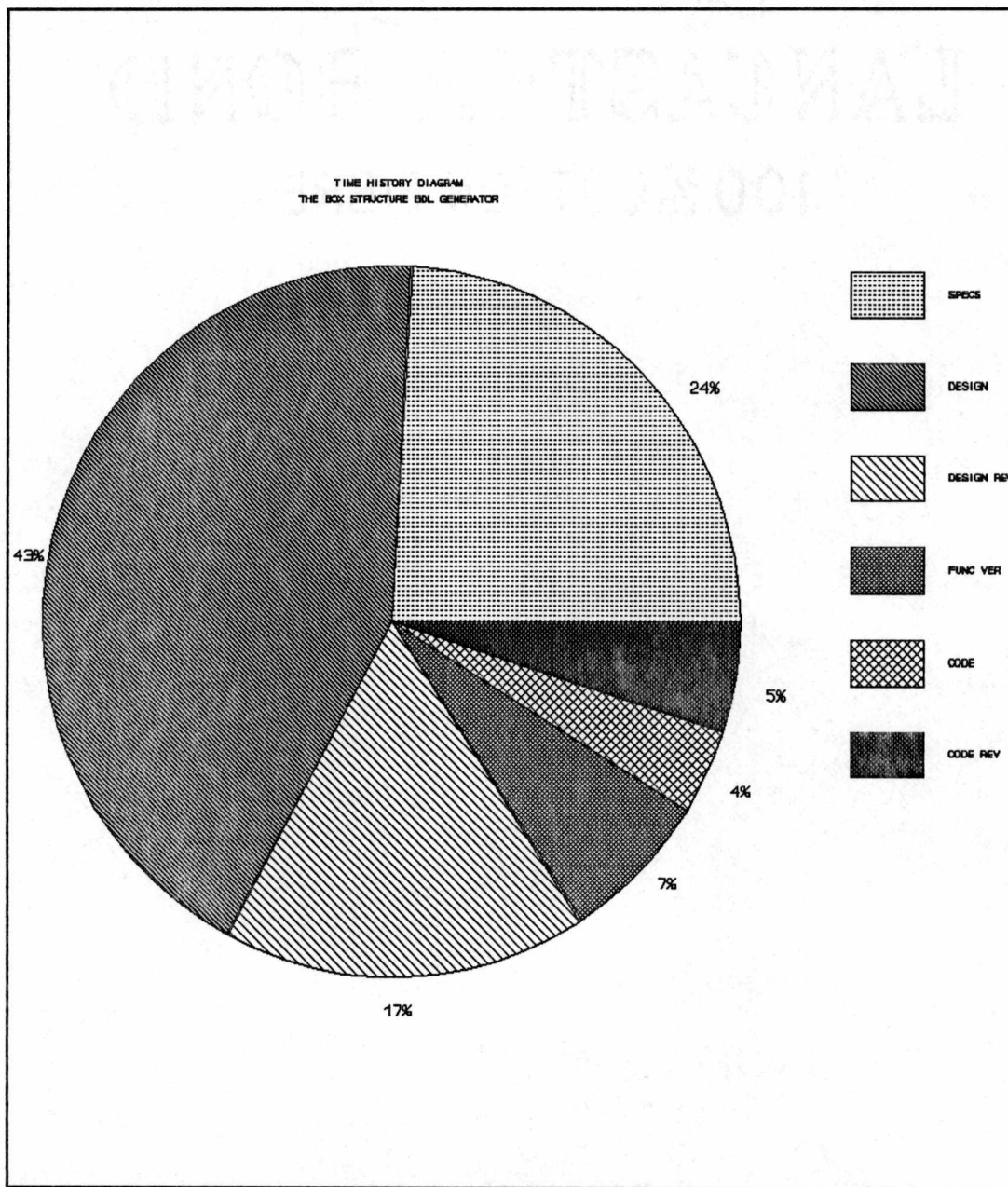


Figure 1 Box Generator Time History

Selby³³ in their work on software testing strategies. Commission is an error resulting from a mistake in the written BDL. Omission, is an error resulting from an oversight in the BDL.

Below are box modification classification summaries for commission and omission for design review, functional verification, and code review (see Appendix C for complete data of the 67 module sample for which data was kept).

As indicated by Table II data, the most difficult problem area encountered with the design review was module interface. Recall that the team had elected not to use any global variables. This decision meant that the parameters "screen display," "master_tmp_ptr," "diskloc_tmp_ptr," "type_file_ptr" (pointers to i/o channels), "master.bdl," "diskloc.bdl," "deftype.bdl", "master.tmp." diskloc.tmp," and "deftype.tmp" (system files) were carried on argument lists in addition to the arguments actually needed by the module. Twelve item argument lists were not uncommon. The sequence of the parameter listings is important and proved to be very difficult to maintain correctly when composing the BDL. Incorporating limited use of global variables would have been

³³V.R. Basili and R.W. Selby, "Comparing the Effectiveness of Software Testing Strategies", IEEE Transactions on Software Engineering, Vol. SE_13, No. 12, (December 1987), p. 1281.

Table II. Design Review Modifications

TREE LEVEL	INITIALIZATION	COMPUTATION	CONTROL	INTERFACE	DATA	COSMETIC
Commission						
1				1		1
2	2		1	1		
3	2		1	3	1	
4	11		4	4	1	
5	9		6	3		
6	1	1	1		2	
7	2					
Omission						
1			1	1	1	
2				2	2	
3	1		6	14	1	
4	3		5	15	4	2
5	2		5	4	2	
6			3	3	1	1
7						

extremely cost effective in terms of time saved in BDL review and could have potentially reduced the errors encountered in the design review by over one third.

Table III shows that during functional verification, interface coordination was still presenting problems, but even more revealing was the number of control errors found. All control errors were the result of incorrect predicates within while/do/od, if/then/fi, or if/then/else/fi constructs. These errors were logic problems resulting from either a misunderstanding of system requirements or a lack of attention to detail by the program designer. In either case, study by the programming team of these logic problems should result in improved programming skills.

Interface and control problems continued to dominate the code review as established in Table IV. The number of logic errors remained troublesome. Additionally, a new circumstance produced further interface difficulties. The target language, 'C', makes excellent use of call by value (copying the value of a variable and sending the copy to the subroutine) and call by reference (sending the machine address of the variable to the subroutine). In the BDL transliteration, if a variable appeared on the stimulus list, it was considered a call by value; if the variable appeared on the response list, it was considered a call by reference. Correct BDL also allowed a

Table III. Functional Verification Modifications

TREE LEVEL	INITIALIZATION	COMPUTATION	CONTROL	INTERFACE	DATA	COSMETIC
Commission						
1				1		
2	1			1	1	1
3			2	1		
4	2		5			
5			5			
6						1
7						
Omission						
1						
2	1		1			1
3			1	3		1
4	2		5	10	1	1
5			1	3		
6			1	1	1	
7						

Table IV. Code Review Modifications

TREE LEVEL	INITIALIZATION	COMPUTATION	CONTROL	INTERFACE	DATA	COSMETIC
Commission						
1						
2			4	2		
3	6		2	16	1	6
4	3	1	4	7	1	
5				2		
6		2	3	17		
7						
Omission						
1				1		
2				2		
3	1		3	7		1
4	5		3	2		
5			1			
6			2	2	1	
7						

single variable to appear on both the stimulus and response lists. However, since 'C' allows only one reference to that single variable, inconsistent transliteration of those variables from BDL to 'C' resulted in interface conflicts.

6.4. TEST RESULTS

Testing is expected to determine that the software can be certified 90% reliable with 90% confidence. Below follows a synopsis of the testing activity.

Error Log

Test Number	Event Number	Pass/Fail	Error Number	Description	Certified MTF
1	1	F	1	unable to write to bdl file, deftype.bdl	0
1	3	F	2	garbage on input line, Top Level Box Name screen	3
1	3	F	3	cursor in wrong position, Stimulus List screen	4
1	3	F	4	no information displayed in window 3, Stimulus List screen	4
1	10	F	5	character 'd' not accepted as input window 2, Stimulus List Screen	9

Test Number	Event Number	Pass/Fail	Error Number	Description	Certified MTTF
1	12	F	6	characters draw across window 2, Stimulus List screen	15
1	12	F	7	input boundary exceeded in window 2, Stimulus List screen	20
2	20	P			38
3	21	F	8	exit from State Migration screen displays incorrect box in Stimulus List screen	47
3	21	F	9	box name not displayed on path bar, Stimulus List screen	50
3	21	F	10	no cursor displayed on Stimulus List screen	53
3	21	F	11	newly declared type does not appear in window 3, Stimulus List screen	56
4	37	F	12	variable "location.no de_id" not initialized	68
4	37	F	13	BDL printing does not terminate	78
5	49	P			111
6	51	F	14	function key "F4" does not execute on Black Box Transition Rule screen	131

Test Number	Event Number	Pass/Fail	Error Number	Description	Certified MTTF
6	62	F	15	Message "Error in append common service" appears	147
6	71	F	16	"use BB" appears twice in window 3, Clear Box Procedure	165
7	80	P			216
8	81	F	17	up arrow produces the error message "box name should not contain blanks" Clear Box Procedure screen	253
8	84	F	18	message "error in print common service" received during BDL printing	263
9	89	P			316
10	141	P			450
11	147	P			593
12	164	P			763
13	170	P			965
14	179	F	19	message "error in update deftype.tmp" received in Type Detail screen	970

The data collected during certification was used as input to the Certification Model. The program requires that a

meaningful time measurement be used when describing the sequence of failures. The test team chose the "event number" as the time measurement because the number of successful events passed between each failure serves as the time log needed by the program. A mean-time-to-failure of 970 events was calculated for the Box Generator at the nineteenth error. When the program passes the 0.90 reliability with 90% confidence certification goal, the mean-time-to-failure will be approximately 25,566 events. Figure 2 charts mean-time-to-failure.

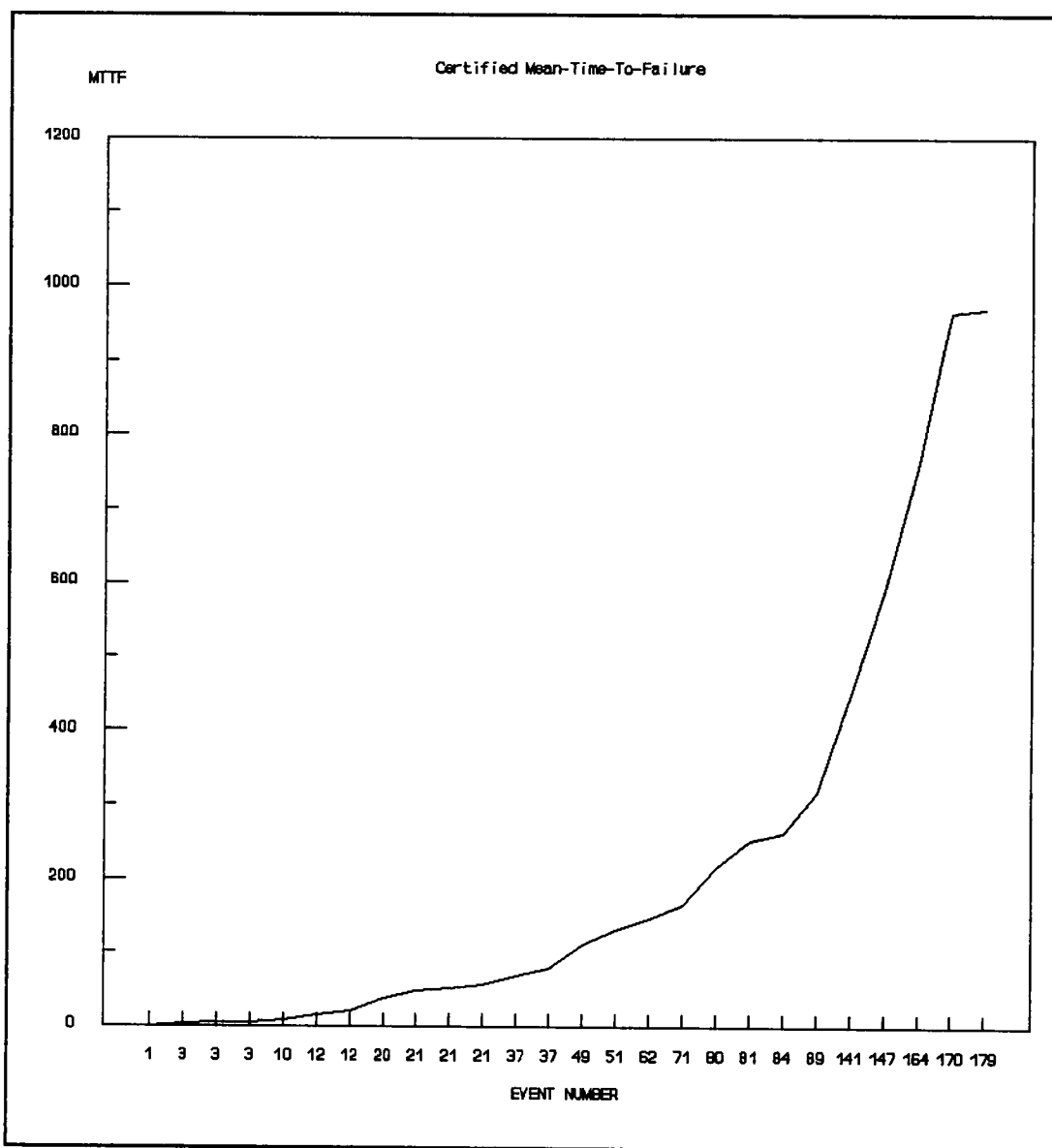


Figure 2. Certified Mean-Time-To-Failure through 19 errors

7. TEAM COMPOSITION AND LESSONS LEARNED

7.1. TEAM COMPOSITION

The chief programmer³⁴ concept was used throughout the development of the Box Generator. The team varied in size from a maximum of six to a minimum of four graduate students. The author served as the chief programmer for the duration of the project, and as such, had the responsibility of ensuring the smooth operation of the team and cooperation among its members. Each team member was self motivated, and professional, with a genuine desire to test his or her ability against the goals established by the methodology. Leadership by example and the public praise of the members' efforts proved the only motivation necessary to keep the team on task and working harmoniously.

7.2. LESSONS LEARNED

Through the course of this project the team learned several lessons. Some of the more significant ones are outlined below.

7.2.1. CRITICIZE CORRECTLY

In projects of this type, the team leader and all team members must exercise care that fragile egos are kept intact during the review processes. It must be recognized that all work

³⁴Harlan D. Mills, Software Productivity (Canada: Little, Brown, & Company, 1983) pp. 65-70.

presented for review represents the creative effort of one or more individuals. The focus must be on the analysis and criticism of the work itself. Any comment, however incidental, which transfers the criticism from the work to the author of that work cannot be tolerated. The reward for this sensitivity is trust among the team members. Trust invites more critical review, and critical review produces better design. The team's own experience was that the individual members felt comfortable enough with the review process that the goal of the verification became two fold: first, to judge the correctness of the BDL; and second, if the BDL presented seemed correct, to optimize the procedure. The long-term value of group review is that the process serves as an open forum for discussion on how best to analyze a problem and code the appropriate solution. The developers learn from each other and becoming stronger programmers in the process.

7.2.2. GOOD SPECIFICATION HIGHLIGHTS "THORNY PROBLEMS"

The major lesson learned from development of the specification was that a proper specification requires as much attention to detail as programming itself. Forcing detail into the specification causes an inspection of the system in a way that "thorny problems" are addressed well before the team is released for design. Discussion will lead to problem identification and solution before erroneous BDL is

constructed and presented for review. For example, the programming team was able to identify the following problem and solution during specification development.

The first operational screen, Top Level Box Name, has the function key F1 available to it so that the Project Menu may be invoked by the user. One of the selections on the Project Menu is "Save", which updates information in the "working" files. Now suppose a user starts the system, comes to this screen and immediately selects "Save" from the Project Menu. Without any checks, the worst case initially appears to be that the system will write to disk a record containing no useful information. However, an additional decision by the user to select the "Bye" option leads to system failure. An exit to DOS occurs, and everything seems normal. But, when the system is started for the next session, a problem appears.

During system start up, a check is made to see if the three files required for system operation exist on the disk. If so (and this is the case here) the system makes a sequential search of "diskloc.tmp," the working file which contains a flag designating the "last worked box." This record is made available to the user as an initial starting point. The problem, in this example, is that the empty record written to disk was the "box last worked," although the flag so designating that box was erroneously never set. Successful

system operation under these circumstances is not possible, since one of the axioms of the system is that exactly one record in "diskloc.tmp" will contain the "last worked flag" set to the appropriate value.

The solution to this problem, required two steps: first, disallow a write to disk under these circumstances and second, set the "last work flag" to TRUE immediately after a record is retrieved from disk.

It must be emphasized that the identification and solution of this problem, and many others like it, evolved during the development of the specification--and well before actual BDL design began.

7.2.3. DON'T BE AFRAID TO START OVER

"Don't be afraid to start over" should be the banner of every design team. The exemplar for this programming assignment was the redesign of the Process Information box discussed previously. The key indicator that redesign was in order was the proliferation of flags and variables used to capture history sequences. On inspection, there seemed to be almost as much code devoted to the overhead as there was to the primary mission of the box, the orderly transfer of control to individual screen boxes. The redesign required

approximately thirty person hours and probably saved at least that many person hours in the functional verification.

7.2.4. THINK COMMON SERVICE

Another lesson learned during the design phase was the value and importance of common services. Process Information was the box that benefited most from the discipline of converting two or more boxes with similar function to a single box that could serve without loss of function in place of the others. Recall that the original design of Process Information required a case statement which would transfer control to individual screen handlers based upon the "screen number" evaluated within the case statement. This first design of Process Information correctly implemented the specification as outlined. However, through the use of common services, it was discovered that the twenty part case statement could be reduced to six parts. Almost every part contained three lower level boxes, so the cumulative effect of the design change was to reduce the fifty-seven boxes required to implement the first design to fourteen boxes in the final design. Although common service designs are generally more complex than the Black Boxes they are replacing, each common service was designed as a functionally cohesive module requiring little more code than any one of the original Black Boxes.

7.2.5. KEEP IT SIMPLE

The old saw "keep it simple" is as relevant to programming as to any other task. Simplicity of design doesn't just happen, it requires hard thought and hard work. However, Cleanroom demands nothing less than simplicity. In addition, with a little more effort, simplicity becomes elegance.

8. IN CONCLUSION

Evidence supporting Cleanroom effectiveness towards the goal of "zero defect software" is slowly mounting. Mills, for example, cites three major projects: IBM language product (eighty-thousand lines of code), Air Force helicopter flight program (thirty-five thousand lines), and a NASA contract (forty-five thousand lines). He summarizes their results as:

1. human verification, even though fallible, can replace debugging in software development
2. verification takes no more time than debugging (although it takes place earlier in the cycle)
3. more than ninety percent of total product defects were found before first execution
4. a drop in total defect count by as much as half is expected
5. for the IBM project, productivity averaged more than 400 lines of code per man-month, largely as a result of sharply reduced testing time and effort compared to conventional developments
6. the IBM project saw a first phase of development (twenty thousand lines) which had just fifty-three errors during testing (2.5 errors per thousand lines of code).³⁵

³⁵ Harlan D. Mills, Michael Dyer, and Richard C. Linger, "Cleanroom Software Engineering", pp. 19-20.

The standard retort to this anecdotal evidence is that the people involved in these projects represent the best the industry had to offer, and that these impressive goals were achieved because of the people, rather than the methodology.³⁶ The retort does not hold for this project however, because of the graduate students involved, only one had commercial programming experience in a team environment. In an effort to add more data to the discussions surrounding Cleanroom, a programming team composed of graduate students from the University of Tennessee produced a 7200 line C-program using Cleanroom. Their results tend to confirm the observations by Mills and others in the following ways:

1. two-hundred-eight errors were detected in the 67 module samples prior to the transliteration of BDL into the target language.
2. one-hundred-nine errors were detected during target language review.
3. group review of design resulted in code reduction of almost 5%.
4. a testing error rate of 3.5 errors per thousand lines of code on first execution was documented;
5. a production rate of 530 lines of code per person-month was achieved.
6. it is noted that the people who comprised the programming team were graduate students, not experienced professionals, and therefore hard work and Cleanroom must receive credit for the team's accomplishment.

³⁶ A paraphrased remark by Richard C. Linger during an informal gathering of the graduate students involved in this project. March 1989.

This project represents additional support for Cleanroom. Definitive scientific verification of the anecdotal evidence will probably never become available; the cost of shadow projects seems prohibitive. Further study is necessary; additional projects need to be developed. But as Lincoln Steffens has said, "I have been over into the future, and it works"³⁷.

³⁷ Lincoln Steffens, The Autobiography of Lincoln Steffens (New York: Harcourt, Blake and Company, 1931), p. 799.

BIBLIOGRAPHY

BIBLIOGRAPHY

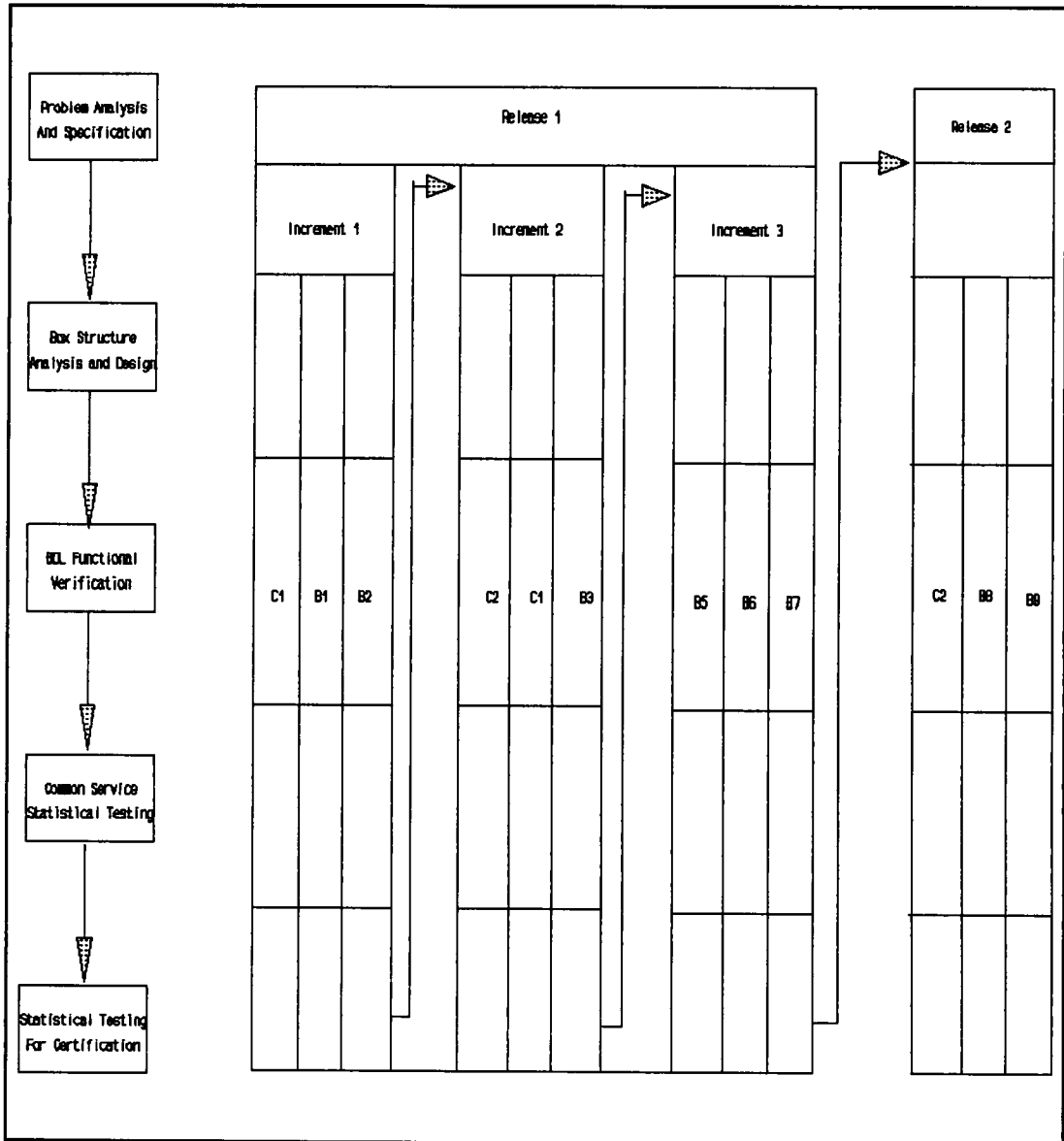
- Baker, F. Terry. "Chief-Programmer Team Management of Production Programming." IBM Systems Journal. V. 1, N. 1, 1972, pp. 56-73.
- Basili, V.R. and R.W. Selby. "Comparing the Effectiveness of Software Testing Strategies." IEEE Transactions on Software Engineering. Vol. SE-13, No.12, December 1987, pp. 1278-1296.
- Brooks, Frederick P. Jr. "No Silver Bullet." Computer April 1987, pp.10-19.
- Currit, Allen, Michael Dyer, and Harlan D. Mills. "Certifying the Reliability of Software." IEEE Transactions on Software Engineering January 1986, pp. 3-11.
- Hammonds, Keith H. "Jim Manzi." Business Week April 14, 1989, pp. 147-153.
- Linger, R.C., H.D. Mills, and B.I. Witt. Structured Programming, Theory and Practice. Reading: Addison-Wesley Publishing Company, Inc., 1979.
- Mandell, Steven L. Computers and Information Processing. Saint Paul: West Publishing Company, 1987.
- Mills, Harlan D. Software Productivity. Canada: Little, Brown & Company, 1983.
- Mills, Harlan D. "Stepwise Refinement and Verification in Box-Structured Systems." Computer June 1986, pp. 23-36.
- Mills, H.D., R.C. Linger, and A.R. Hevner. "Box Structured Information Systems." IBM Systems Journal V. 26, N. 4, 1987, pp. 1-15.
- Mills, Harlan D., Richard C. Linger, and Alan R. Hevner. Principles of Information Systems Analysis and Design. Orlando: Academic Press, Inc., 1986.
- Mills, Harlan D., Michael Dyer, and Richard C. Linger. "Cleanroom Software Engineering." IEEE Transactions on Software Engineering September 1987, pp. 19-25.

- Mills, Harlan D. and J.H. Poore. "Bringing Software Under Statistical Quality Control." Quality Progress November 1988, pp. 52-65.
- Poore, J.H. "An Overview of the Cleanroom Software Development Process." Formal Methods Workshop, Halifax, Nova Scotia, 1989.
- Selby, Richard W., Victor R. Basili, and F. Terry Baker. "Cleanroom Software Development: An Empirical Evaluation." IEEE Transactions on Software Engineering. Vol SE-13, No 9, September 1987, pp. 1027-1037.
- Senn, Charles C., Carmen Trammell, Hialong Mao, Norman Sims. "BDL for the Box Generator." Technical Report, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1989.
- Senn, Charles C., Carmen Trammell, Hialong Mao, Norman Sims. "Box Generator 'C' Code." Technical Report, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1989.
- Senn, Charles C., Carmen Trammell, Hialong Mao, Randy Baker, Norman Sims. "Box Generator Specification." Technical Report, Department of Computer Science, University of Tennessee, Knoxville, Tennessee, 1989.
- Sexton, Bonnie Collet. "Statistical Testing of Software." Master's Thesis, University of Tennessee, Knoxville, Tennessee, 1988.
- Steffens, Lincoln. The Autobiography of Lincoln Steffens. New York: Harcourt, Blake and Company, 1931.

APPENDICES

APPENDIX A

CLEANROOM SOFTWARE DEVELOPMENT LIFE CYCLE



APPENDIX B

Top Level Box Name: _____

F1-Project Menu
F6-

F2-
F7-

F3-
F8-

F4-Next Screen
F9-

F5-
F10-Help

Project Name		::BB Box Name	:::Stimulus List
Name	Stimulus	Type	Defined Types
			Array Boolean Character File Integer Pointer Real Structure
scroll			
F1-Project Menu	F2-Box Menu	F3-	F4-Next Screen
F6-New Type	F7-Type Detail	F8-	F5-Change Window
			F9-
			F10-Help

Project Name	::BB Box Name	:::Response List
Name	Type	Defined Types
Response: _____		Array Boolean Character File Integer Pointer Real Structure
scroll		
F1-Project Menu	F2-Box Menu	F3-Prev Screen
F4-Next Screen	F5-Change Window	F6-New Type
F7-Type Detail	F8-	F9-
F10-Help		

Name

Defined Types

Response: _____

```

Array
Boolean
Character
File
Integer
Pointer
Real
Structure

```

scroll

F1-Project Menu

F2-Box Menu

F3-Prev Screen

F4-Next Screen

F5-Change Window

F6-New Type

F7-Type Detail

F8-

F9-

F10-Help

Project Name	::BB Box Name	:::BB Transition Rule
Black Box Transition Rule		
F1-Project Menu F6-	F2-Box Menu F7-	F3-Prev Screen F8-
		F4-Next Screen F9-
		F5- F10-Help

Project Name	::BB Box Name	:::State Data List
Name	Type	Defined Types
Variable: _____		Array Boolean Character File Integer Pointer Real Structure
scroll		
F1-Project Menu	F2-Box Menu	F3-Prev Screen
F4-Next Screen	F5-Change Window	F6-New Type
F7-Type Detail	F8-	F9-
F10-Help		

Project Name	::BB Box Name	:::SB Transition Rule
State Box Transition Rule		
F1-Project Menu F6-	F2-Box Menu F7-	F3-Prev Screen F8- F4-Next Screen F9- F5- F10-Help

Project Name		::BB Box Name	:::Machine Data List	
Machine Data				
Name		Type		Defined Types
				Array Boolean Character File Integer Pointer Real Structure
scroll				
F1-Project Menu	F2-Box Menu	F3-Prev Screen	F4-Next Screen	F5-Change Window
F6-New Type	F7-Type Detail	F8-	F9-	F10-Help

Project Name :: CB Box Name ::: Clear Box Procedure

Clear Box Procedure

```
proc
corp
```

BDL Outer Syntax

```
use Bb
use cS
Do/od
If/then/fi
if/then/Else/fi
Case/part/esac
Part
While/do/od
cOn/noc
```

scroll

F1-Project Menu F2-Box Menu F3-Previous Screen F4-Next Screen F5-Change Window
F6- F7-Select CS F8- F9-Delete Structure F10-Help

APPENDIX C

Box Generator Hierarchical Tree

- 1. box_generator
 - 1.1. make_windows
 - 1.2. init_structures
 - 1.2.1. ct31_init_cbproc
 - 1.3. scrn_1
 - 1.4. locate_files
 - 1.5. create_project_file
 - 1.6. create_disk_loc_file
 - 1.7. create_def_type_file
 - 1.7.1. blanks
 - 1.8. create_working_project_file
 - 1.9. create_working_disk_loc_file
 - 1.10. create_working_type_def_file
 - 1.11. create_def_types_list
 - 1.12. load_current_node
 - 1.12.1. get_current_index
 - 1.12.2. load_record
 - 1.13. process_information
 - 1.13.1. get_tlb_name
 - 1.13.2. draw_2_22_tlb_new_type
 - 1.13.2.1. blanks
 - 1.13.2.2. draw_path
 - 1.13.2.3. draw_fkeys
 - 1.13.3. ctrl_2_11_22_tlb_new_type_
 - 1.13.3.1. get_col_row
 - 1.13.3.2. update_max_scrn
 - 1.13.3.3. blanks
 - 1.13.3.4. ct_2_set_cursor
 - 1.13.3.5. clr_m_w_cursor
 - 1.13.3.6. ct_2_pc
 - 1.13.3.7. ct_2_bs
 - 1.13.3.8. ct_2_cd
 - 1.13.3.9. ct_f1
 - 1.13.3.9.1. popup_menu
 - 1.13.3.9.1.1. draw_menu_outline
 - 1.13.3.9.1.2. draw_menu_outline
 - 1.13.3.9.1.3. fill_menu_text
 - 1.13.3.9.1.4. get_menu_choice
 - 1.13.3.9.2. draw_4_5_6_14_15_saving_printing
 - 1.13.3.9.2.1. draw_fkeys
 - 1.13.3.9.2.2. draw_one_window
 - 1.13.3.9.2.2.1. draw_path
 - 1.13.3.9.2.2.2. draw_fkeys
 - 1.13.3.9.3. ctrl_4_saving
 - 1.13.3.9.3.1. update_tmp_files
 - 1.13.3.9.3.2. update_master_files
 - 1.13.3.9.4. printer_check
 - 1.13.3.9.4.1. clr_m_no_cursor
 - 1.13.3.9.4.2. clr_m_no_cursor
 - 1.13.3.9.5. draw_4_5_6_14_15_saving_printing
 - 1.13.3.9.6. update_tmp_files
 - 1.13.3.9.7. ctrl_5_print_project
 - 1.13.3.9.7.1. load_diskloc_record
 - 1.13.3.9.7.2. breadth_first_print
 - 1.13.3.9.7.2.1. load_record
 - 1.13.3.9.7.2.2. print_box

```

1.13.3.9.7.2.2.1. print_keyword_and_line
1.13.3.9.7.2.2.1.1. print_line
1.13.3.9.7.2.2.1.2. check_line
1.13.3.9.7.2.2.1.2.1.
    print_page
1.13.3.9.7.2.2.2. print_line
1.13.3.9.7.2.2.3. print_line
1.13.3.9.7.2.2.4.
    print_key_word_and_line
1.13.3.9.7.2.2.5. print_none
1.13.3.9.7.2.2.5.1. print_line
1.13.3.9.7.2.2.5.2. check_line
1.13.3.9.7.2.2.6.
    print_stim_resp_state_mach
1.13.3.9.7.2.2.6.1. blanks
1.13.3.9.7.2.2.6.2. print_line
1.13.3.9.7.2.2.6.3. check_line
1.13.3.9.7.2.2.7. print_keyword_and_line
1.13.3.9.7.2.2.8. print_none
1.13.3.9.7.2.2.9.
    print_stim_resp_state_mach
1.13.3.9.7.2.2.10.
    print_keyword_and_line
1.13.3.9.7.2.2.11. print_none
1.13.3.9.7.2.2.12.
    print_stim_resp_state_mach
1.13.3.9.7.2.2.13.
    print_keyword_and_line
1.13.3.9.7.2.2.14.
    print_keyword_and_line
1.13.3.9.7.2.2.15. print_none
1.13.3.9.7.2.2.16.
    print_stim_resp_state_mach
1.13.3.9.7.2.2.17.
    print_keyword_and_line
1.13.3.9.7.2.2.18. print_trans_proc
1.13.3.9.7.2.2.18.1. print_line
1.13.3.9.7.2.2.18.2. blanks
1.13.3.9.7.2.2.18.3. get_name
1.13.3.9.7.2.2.18.4. check_line
1.13.3.9.7.2.2.19. print_trans_proc
1.13.3.9.7.2.3. load_diskloc_record
1.13.3.9.7.2.4. load_record
1.13.3.9.8. draw_4_5_6_14_15_saving_printing
1.13.3.9.9. ctrl_6_ask_save
1.13.3.9.9.1. stub
1.13.3.9.9.2. clear_m_n_cursor
1.13.3.9.9.3. draw_4_5_6_14_15_saving_printing
1.13.3.9.9.4. ctrl_4_saving
1.13.3.9.10. draw_7_exit
1.13.3.10. ct_f3_f10
1.13.3.10.1. stub
1.13.3.10.2. clr_m_w_cursor
1.13.3.11. ct_2_f4
1.13.3.12. ct_f3_f10
1.13.3.13. ct_2_la
1.13.3.14. ct_2_ra
1.13.3.15. ct_2_del
1.13.4. draw_9_10_11_select_edit_rename_migrate
1.13.4.1. draw_one_window
1.13.4.2. draw_path
1.13.4.3. draw_fkeys
1.13.5. ctrl_9_select
1.13.5.1. update_max_screen
1.13.5.2. pre_order

```

- 1.13.5.2.1. instantiate_node
 - 1.13.5.2.2. pre_order
 - 1.13.5.2.3. pre_order
- 1.13.5.3. append_common_service
 - 1.13.5.3.1. pre_order
- 1.13.5.4. ld_9
 - 1.13.5.4.1. ct_9_fill
 - 1.13.5.4.2. get_9_choice
 - 1.13.5.4.2.1. update_tmp_files
 - 1.13.5.4.2.2. load_record
 - 1.13.5.4.2.3. load_diskloc_record
- 1.13.6. draw_21_24_27_30_31
 - 1.13.6.1. draw_path
 - 1.13.6.2. draw_windows
 - 1.13.6.3. draw_fkeys
- 1.13.7. load_21_24_27_30_var_list
 - 1.13.7.1. load_define_types
- 1.13.8. ctrl_21_24_27_30_var_list
 - 1.13.8.1. update_max_screen
 - 1.13.8.2. get_max_rows
 - 1.13.8.3. blanks
 - 1.13.8.4. ct_21_set_cursor
 - 1.13.8.5. clr_m_w_cursor
 - 1.13.8.6. ct_f1
 - 1.13.8.7. ct_f2
 - 1.13.8.7.1. popup_menu
 - 1.13.8.7.2. stub
 - 1.13.8.7.3. clr_m_n_cursor
 - 1.13.8.7.4. ctrl_9_select
 - 1.13.8.7.5. draw_9_10_11_37_select_edit_rename_migrate
 - 1.13.8.7.6. ctrl_2_11_22_tlb_n_rename_new_type
 - 1.13.8.7.7. find_name
 - 1.13.8.7.8. re_name
 - 1.13.8.7.9. draw_4_5_6_14_15_saving_printing
 - 1.13.8.7.10. ctrl_14_ask_delete
 - 1.13.8.7.10.1. stub
 - 1.13.8.7.10.2. clr_m_no_cursor
 - 1.13.8.7.10.3. draw_4_5_6_14_15_saving_printing
 - 1.13.8.7.10.4. invalidate_boxes
 - 1.13.8.7.10.5. get_parent
 - 1.13.8.7.10.6. find_name
 - 1.13.8.7.10.7. reorder_ptrs
 - 1.13.8.7.10.8. redo_node_ids
 - 1.13.8.7.10.9. clear_tmp_files
 - 1.13.8.7.11. printer_check
 - 1.13.8.7.12. draw_4_5_6_14_15_printing_saving
 - 1.13.8.7.13. print_box
 - 1.13.8.7.14. draw_path
 - 1.13.8.8. ct_f3_f10
 - 1.13.8.9. ct_21_f5
 - 1.13.8.9.1. draw_fkeys
 - 1.13.8.9.2. fill_w3_text
 - 1.13.8.9.3. get_21_menu_choice
 - 1.13.8.9.3.1. display_scroll
 - 1.13.8.9.3.2. ct_f3_f10
 - 1.13.8.9.4. draw_fkeys
 - 1.13.8.10. draw_path
 - 1.13.8.11. ct_21_f6
 - 1.13.8.11.1. draw_2_22_tlb_n_new_type
 - 1.13.8.11.2. ctrl_2_11_22_tlb_n_rename_new_type
 - 1.13.8.11.3. fill_w3_text
 - 1.13.8.12. ct_21_f7
 - 1.13.8.12.1. draw_23_26_29_new_type_bbtr_sbtr
 - 1.13.8.12.1.1. draw_path
 - 1.13.8.12.1.2. draw_fkeys

```

1.13.8.12.2. read_description
1.13.8.12.3. load_23_type_description
1.13.8.12.4. ctrl_23_26_29_new_type_bbtr_sbtr
    1.13.8.12.4.1. blanks
    1.13.8.12.4.2. ct_23_set_cursor
    1.13.8.12.4.3. clr_m_w_cursor
    1.13.8.12.4.4. ct_23_pc
        1.13.8.12.4.4.1. get_max_rows
    1.13.8.12.4.5. ct_23_bs
    1.13.8.12.4.6. ct_23_cr
        1.13.8.12.4.6.1. get_max_rows
    1.13.8.12.4.7. ct_23_cd
        1.13.8.12.4.7.1. get_max_rows
    1.13.8.12.4.8. ct_f1
    1.13.8.12.4.9. ct_f2
    1.13.8.12.4.10. ct_f3_f10
    1.13.8.12.4.11. ct_f3_f10
    1.13.8.12.4.12. ct_23_la
    1.13.8.12.4.13. ct_23_ra
        1.13.8.12.4.13.1. get_max_rows
    1.13.8.12.4.14. ct_23_ua
    1.13.8.12.4.15. ct_23_da
        1.13.8.12.4.15.1. get_max_rows
    1.13.8.12.4.16. ct_23_del
1.13.8.12.5. update_deftype_tmp
1.13.8.13. ct_f3_f10
1.13.8.14. ct_21_la
1.13.8.15. ct_21_ra
    1.13.8.15.1. get_max_rows
1.13.8.16. ct_21_ua
1.13.8.17. ct_21_da
    1.13.8.17.1. get_max_rows
1.13.8.18. ct_21_del
1.13.8.19. ct_21_bs
1.13.8.20. ct_21_cr
1.13.8.21. ct_21_cd
    1.13.8.21.1. get_max_rows
1.13.8.22. ct_21_pc
1.13.9. draw_23_26_29_new_type_bbtr_sbtr
1.13.10. load_26_29_bbtr_sbtr
1.13.11. ctrl_23_26_29_new_type_bbtr_sbtr
1.13.12. ctrl_23_26_29_new_type_bbtr_sbtr
1.13.13. draw_21_24_27_30_31
1.13.14. ctrl_31_clear_box_proc
    1.13.14.1. update_max_scrn
    1.13.14.2. ct31_load
        1.13.14.2.1. ct_31_displayproc
            1.13.14.2.1.1. ct31_getstr
    1.13.14.3. clr_m_w_cursor
    1.13.14.4. ct_f1
    1.13.14.5. ct_f2
    1.13.14.6. ct_f3_f10
    1.13.14.7. ct31_f4
    1.13.14.8. ct_31_f5
        1.13.14.8.1. display_scroll
        1.13.14.8.2. draw_fkeys
        1.13.14.8.3. popup_menu
        1.13.14.8.4. display_scroll
1.13.14.9. ct31_newstruct
    1.13.14.9.1. ct31_ins_lines
        1.13.14.9.1.1. ct31_str_initial
    1.13.14.9.2. ct31_setstruct
    1.13.14.9.3. ct31_set_line_mode
    1.13.14.9.4. ct31_displayproc
1.13.14.10. ct31_f9

```

```

1.13.14.10.1. ct31_paring
1.13.14.10.2. ct31_paring
1.13.14.10.3. ct31_paring
1.13.14.10.4. ct31_paring
1.13.14.10.5. ct31_paring
1.13.14.10.6. ct31_paring
1.13.14.10.7. ct31_paring
1.13.14.10.8. ct31_paring
1.13.14.10.9. ct31_paring
1.13.14.10.10. ct31_paring
1.13.14.10.11. ct31_paring
1.13.14.10.12. ct31_del_bbs
    1.13.14.10.12.1. get_name
    1.13.14.10.12.2. clr_m_w_cursor
    1.13.14.10.12.3. delete_name
        1.13.14.10.12.3.1. reorder_ptrs
        1.13.14.10.12.3.2. invalidate_boxes
        1.13.14.10.12.3.3. redo_node_ids
1.13.14.10.13. ct31_del_lines
1.13.14.10.14. ct31_displayproc
1.13.14.11. ct_f3_f10
1.13.14.12. ct31_del
1.13.14.13. ct31_up_down_arrow
    1.13.14.13.1. ct31_maintain_bb_name
        1.13.14.13.1.1. ct31_getstr
        1.13.14.13.1.2. ct31_insert_name
            1.13.14.13.1.2.1. find_name
            1.13.14.13.1.2.2. insert_name
                1.13.14.13.1.2.2.1. max_level_test
                1.13.14.13.1.2.2.2. get_last_right
                1.13.14.13.1.2.2.3. init_structures
                1.13.14.13.1.2.2.4. ct31_init_cbproc
            1.13.14.13.1.2.3. add_alias
                1.13.14.13.1.2.3.1. max_level_test
                1.13.14.13.1.2.3.2. get_last_right
                1.13.14.13.1.2.3.3. convert_BB_to_CS
                1.13.14.13.1.2.3.4.
                    convert_first_rec_to_CS_info
        1.13.14.13.1.3. get_name
        1.13.14.13.1.4. ct31_rename
            1.13.14.13.1.4.1. find_name
            1.13.14.13.1.4.2. re_name
1.13.14.14. ct_31_up_down_arrow
1.13.14.15. ct_31_left_right_arrow
1.13.14.16. ct_31_left_right_arrow
1.13.14.17. ct_31_ctrl_del
    1.13.14.17.1. ct31_del_lines
    1.13.14.17.2. ct31_displayproc
    1.13.14.17.3. ct31_set_line_mode
1.13.14.18. ct31_alt_key
    1.13.14.18.1. ct31_newstruct
1.13.14.19. ct31_character
1.13.14.20. ct31_bs
    1.13.14.20.1. ct31_del
1.13.14.21. ct31_cr
    1.13.14.21.1. ct31_up_down_arrow
    1.13.14.21.2. ct31_maintain_bb_name
    1.13.14.21.3. ct31_set_line_mode
    1.13.14.21.4. ct31_ins_lines
    1.13.14.21.5. ct31_displayproc
1.13.15. draw_9_10_11_37_select_edit_rename_migrate
1.13.16. ctrl_37_state_mig
    1.13.16.1. ct_f1
    1.13.16.2. ct_f2
    1.13.16.3. ct_f3_f10

```

- 1.13.16.4. update_tmp_files
- 1.13.16.5. get_next_box
 - 1.13.16.5.1. load_diskloc_record
 - 1.13.16.5.2. breadth_first_done_check
 - 1.13.16.5.2.1. add_q
 - 1.13.16.5.2.2. delete_q
- 1.13.16.6. load_record
- 1.13.16.7. ct_f3_f10

APPENDIX D

Modifications as a Result of Group Review

MODIFICATION TYPE	FAULT CLASS	BOX NAME	DESCRIPTION
commission	computation	update_tmp_f	incorrect size of record computed
commission	control	cr_def_ty_lst	incorrect predicate in a while/do construct
commission	control	ct_21_da	incorrect predicate in an if/then/else construct
commission	control	ct_21_ra	incorrect predicate in an if/then/else construct
commission	control	ct_23_cd	incorrect predicate in a while/do construct
commission	control	ct_f2	incorrect part in case statement
commission	control	ctrl_2_11_22	incorrect predicate in an if/then
commission	control	fill_w3_txt	incorrect predicate in a while/do construct
commission	control	fill_w3_txt	incorrect predicate in while/do construct
commission	control	fill_w9_txt	incorrect predicate in while/do construct
commission	control	get_9_ch	incorrect predicate in an if/then/else construct
commission	control	get_9_ch	incorrect predicate in a while/do construct
commission	control	get_menu_ch	incorrect predicate in a while/do construct
commission	control	popup_menu	call to incorrect subroutine
commission	cosmetic	box_generator	lower level box name misspelled
commission	data	ct_21_cr	incorrect data presented on screen
commission	data	ct_23_26	incorrect array update within a while/do construct
commission	data	get_edit_ch	incorrect map of user selection to screen number
commission	data	get_menu_ch	incorrect positioning of menu light bar
commission	initialization	cr_def_ty_lst	incorrect syntax used in array initialization
commission	initialization	cr_head_node	incorrect pointer assignment
commission	initialization	ct_21_la	incorrect value assigned to variable
commission	initialization	ct_21_la	incorrect value assigned to variable
commission	initialization	ct_21_la	incorrect value assigned to variable
commission	initialization	ct_21_la	incorrect value assigned to variable
commission	initialization	ct_21_ra	incorrect value assigned to variable
commission	initialization	ct_21_ra	incorrect value assigned to variable
commission	initialization	ct_21_ra	incorrect value assigned to variable
commission	initialization	ct_21_ra	incorrect value assigned to variable
commission	initialization	ct_21_ra	incorrect value assigned to variable
commission	initialization	ct_2_ra	incorrect value assigned to variable
commission	initialization	ct_9_fill	incorrect value assigned to variable
commission	initialization	ct_f2	incorrect value assigned to variable
commission	initialization	ctrl_21	incorrect value assigned to variable
commission	initialization	fill_m_txt	incorrect value assigned to variable
commission	initialization	fill_w3_txt	incorrect value assigned to variable
commission	initialization	fill_w3_txt	incorrect value assigned to variable
commission	initialization	fill_w9_txt	incorrect value assigned to variable
commission	initialization	init_struct	incorrect pointer assignment
commission	initialization	inst_node	value assigned to undeclared variable
commission	initialization	load_diskl_r	incorrect value assigned to variable
commission	initialization	load_diskl_r	incorrect value assigned to variable
commission	initialization	popup_menu	incorrect array element assignment
commission	initialization	popup_menu	incorrect array element assignment
commission	initialization	popup_menu	incorrect array element assignment
commission	initialization	popup_menu	incorrect array element assignment
commission	interface	box_generator	incorrect number of arguments in a subroutine call
commission	interface	ct_2_del	incorrect argument list
commission	interface	ct_2_f4	incorrect argument list
commission	interface	ct_2_la	incorrect argument list
commission	interface	ct_2_ra	incorrect argument list
commission	interface	ctrl_2_11_22	incorrect argument list

MODIFICATION TYPE	FAULT CLASS	BOX NAME	DESCRIPTION
commission	interface	ctrl_2_11_22	incorrect argument list
commission	interface	fill_w3_text	incorrect argument list
commission	interface	fill_w3_txt	incorrect argument list
commission	interface	load_cur_node	incorrect argument list
commission	interface	load_record	incorrect argument list
commission	interface	update_tmp_f	incorrect argument list
omission	control	blanks	incorrect predicate in a while/do construct
omission	control	box_generator	passed flag to lower level and did not check return value
omission	control	ct_21_f6	failure to assign correct value to return flag
omission	control	ct_21_ra	failure to call required subroutine
omission	control	ct_f1	passed flag to lower level and did not check return value
omission	control	ct_f1	incorrect screen presented to user
omission	control	ct_f2	failure to provide for a case statement part
omission	control	ctrl_23_26	incorrect predicate in an if/then/else construct
omission	control	ctrl_23_26	incorrect predicate in an if/then/else construct
omission	control	ctrl_23_26	incorrect predicate in an if/then/else construct
omission	control	ctrl_2_11_22	incorrect predicate in while/do construct
omission	control	get_9_ch	failure to update data structure variable
omission	control	get_9_ch	failure to call required subroutine
omission	control	get_9_ch	failure to update data structure variable
omission	control	get_cur_index	incorrect predicate in an if/then/else
omission	control	get_menu_ch	failure to assign correct value to return flag
omission	control	get_menu_ch	failure to call required subroutine
omission	control	get_menu_ch	omitted section of case statement
omission	control	popup_menu	incorrect brace matching in case statement
omission	control	popup_menu	failure to call required subroutine
omission	cosmetic	ct_f2	failure to draw screen function keys
omission	cosmetic	draw_path	failure to clear function key words
omission	cosmetic	get_menu_ch	failure to turn off light bar
omission	data	box_generator	structure variable not set
omission	data	ct_21_cr	incorrect data presented on screen
omission	data	ct_2_bs	incorrect data presented on screen
omission	data	ct_2_del	failure to assign end-of-string sentinel
omission	data	ct_9_fill	failure to free dynamic memory
omission	data	ct_f2	failure to update data structure variable
omission	data	ctrl_9	failure to free dynamic memory
omission	data	get_9_ch	failure to free dynamic memory
omission	data	load_cur_node	structure variable not set
omission	data	load_cur_node	structure variable not set
omission	data	update_mstr_f	failure to update temporary file
omission	initialization	ct_21_bs	failure to assign end-of-string sentinel
omission	initialization	ct_21_f6	failure to assign end-of-string sentinel
omission	initialization	ct_f2	failure to provide for two necessary variables
omission	initialization	inst_node	failure to initialize pointer
omission	initialization	inst_node	failure to initialize pointer
omission	initialization	load_record	failure to set error flag when an error occurs
omission	interface	box_generator	incorrect argument list
omission	interface	cr_head_node	incorrect argument list
omission	interface	ct_21_f6	incorrect argument list
omission	interface	ct_21_f7	incorrect argument list
omission	interface	ct_21_f7	incorrect argument list
omission	interface	ct_21_f7	incorrect argument list
omission	interface	ct_21_f7	incorrect argument list
omission	interface	ct_23_la	incorrect argument list
omission	interface	ct_23_set_c	incorrect argument list
omission	interface	ct_23_ua	incorrect argument list
omission	interface	ct_2_bs	incorrect argument list
omission	interface	ct_f1	incorrect argument list
omission	interface	ct_f1	incorrect argument list
omission	interface	ctrl_23_26	incorrect argument list

MODIFICATION TYPE	FAULT CLASS	BOX NAME	DESCRIPTION
omission	interface	ctrl_23_26	incorrect argument list
omission	interface	ctrl_23_26	incorrect argument list
omission	interface	ctrl_23_26	incorrect argument list
omission	interface	ctrl_23_26	incorrect argument list
omission	interface	ctrl_23_26	incorrect argument list
omission	interface	ctrl_23_26	incorrect argument list
omission	interface	ctrl_2_11_22	incorrect argument list
omission	interface	ctrl_2_11_22	incorrect argument list
omission	interface	ctrl_2_11_22	incorrect argument list
omission	interface	ctrl_2_11_22	incorrect argument list
omission	interface	ctrl_9	incorrect argument list
omission	interface	ctrl_9	incorrect argument list
omission	interface	ctrl_9	incorrect argument list
omission	interface	get_9_ch	incorrect argument list
omission	interface	get_9_ch	incorrect number of arguments in a subroutine call
omission	interface	get_edit_ch	incorrect argument list
omission	interface	get_menu_ch	incorrect argument list
omission	interface	popup_menu	incorrect argument list
omission	interface	popup_menu	incorrect argument list
omission	interface	pre_order	incorrect argument list
omission	interface	pre_order	incorrect argument list
omission	interface	pre_order	incorrect argument list
omission	interface	process_info	incorrect argument list
omission	interface	update_mstr_f	incorrect argument list
optimization	n/a	box_generator	convert compound predicate in if/then to nested if
optimization	n/a	calc_w_h	eliminate subroutine
optimization	n/a	check_dim	eliminate subroutine
optimization	n/a	cr_def_ty_f	close file channel
optimization	n/a	cr_def_ty_f	eliminate unnecessary local variable
optimization	n/a	cr_diskloc_f	close file channel
optimization	n/a	cr_project_f	close file channel
optimization	n/a	ct_21_cd	convert nested while loop to 'c' function call
optimization	n/a	ct_21_da	eliminate duplicate assignment statements
optimization	n/a	ct_21_da	replace case statement with subroutine call
optimization	n/a	ct_21_del	convert nested while loop to 'c' function call
optimization	n/a	ct_21_f5	eliminate unnecessary subroutine calls
optimization	n/a	ct_21_f5	eliminate unnecessary subroutine calls
optimization	n/a	ct_21_pc	reduce duplication of string assignments
optimization	n/a	ct_21_ua	eliminate duplicate assignment statements
optimization	n/a	ct_23_cd	convert nested while loop to 'c' function call
optimization	n/a	ct_23_cr	convert nested while loop to 'c' function call
optimization	n/a	ct_23_da	eliminate unnecessary cursor positioning
optimization	n/a	ct_23_del	convert nested while loop to 'c' function call
optimization	n/a	ct_23_f4	eliminate subroutine
optimization	n/a	ct_23_f4	eliminate subroutine
optimization	n/a	ct_23_ua	eliminate unnecessary cursor command
optimization	n/a	ct_2_bs	eliminate unnecessary cursor command
optimization	n/a	ct_2_cd	eliminate unnecessary cursor command
optimization	n/a	ct_2_cr	eliminate subroutine
optimization	n/a	ct_2_cr	eliminate subroutine
optimization	n/a	ct_2_f4	check for blanks in box name
optimization	n/a	ct_2_la	eliminate unnecessary cursor positioning
optimization	n/a	ctrl_21	restructure if/then/else to case
optimization	n/a	ctrl_2_11_22	removed unnecessary call to common service
optimization	n/a	draw_m_oln	eliminate unnecessary local variable
optimization	n/a	draw_path	eliminate unnecessary print statements
optimization	n/a	draw_w_t	eliminate unnecessary print statements
optimization	n/a	fill_menu_text	converted to common service
optimization	n/a	fill_w9_text	eliminate subroutine

MODIFICATION TYPE	FAULT CLASS	BOX NAME	DESCRIPTION
optimization	n/a	get_21_m_c	replace calculation with global constant
optimization	n/a	get_edit_ch	remove repeated code in the procedure
optimization	n/a	get_menu_ch	eliminated unnecessary assignment
optimization	n/a	get_tlb_name	read field from record instead of whole record
optimization	n/a	load_def_types	removed redundant check in while/do predicate
optimization	n/a	load_record	structure variable assignment not required
optimization	n/a	update_tmp_f	check return flag after i/o operation

Modifications as a Result of Functional Verification

MODIFICATION TYPE	FAULT CLASS	BOX NAME	DESCRIPTION
commission	control	ct_21_del	incorrect argument in 'c' function call
commission	control	ct_21_ra	incorrect argument in 'c' function call
commission	control	ct_23_cr	incorrect argument in 'c' function call
commission	control	ct_23_ua	incorrect predicate in if/then/else construct
commission	control	ct_f2	logic error in case statement
commission	control	ctrl_21_24	unnecessary subroutine call in part statement
commission	control	ctrl_23_26	call to incorrect subroutine
commission	control	get_21_m_ch	incorrect predicate in while/do construct
commission	control	pre_order	incorrect predicate in if/then construct
commission	control	update_defty_f	incorrect predicate in if/then/else construct
commission	control	update_defty_f	incorrect predicate in if/then/else construct
commission	cosmetic	print_box	incorrect spacing in printed BDL
commission	cosmetic	scrn_1	increase number of seconds display is on screen
commission	cosmetic	stub	failure to leave display on screen for five seconds
commission	data	cr_def_ty_f	incorrect string written to file
commission	initialization	cr_def_ty_f	incorrect pointer assignment to file
commission	initialization	ct_21_cd	incorrect assignment to local variable
commission	initialization	ct_f2	incorrect assignment to local variable
commission	interface	box_generator	incorrect argument list
commission	interface	ctrl_2_11_22	incorrect argument list
commission	interface	process_info	incorrect argument list
omission	control	ct_21_cd	incorrect predicate in while/do construct
omission	control	ct_21_da	logic error in subroutine
omission	control	ct_21_pc	failure to provide required part in a case statement
omission	control	ct_21_ua	logic error in subroutine
omission	control	ct_2_ra	incorrect parenthesis matching
omission	control	draw_2_22	failure to test return flag
omission	control	get_menu_ch	failure to incorporate loop controller
omission	control	popup_menu	failure to provide required if/then construct
omission	control	process_info	failure to test return flag
omission	cosmetic	cr_diskloc_f	failure to leave display on screen for five seconds
omission	cosmetic	ct_21_cr	failure to print a string of blanks
omission	cosmetic	get_cur_index	failure to display error message
omission	data	ct_21_f7	failure to update a permanent file
omission	data	update_mstr_f	failure to update a permanent file
omission	initialization	ct_21_pc	local variable not declared
omission	initialization	ct_f1	local variable not declared
omission	initialization	init_struct	failure to properly initialize data record
omission	interface	ct_21_cd	incorrect argument list
omission	interface	ct_21_cr	incorrect argument list
omission	interface	ct_23_da	incorrect argument list
omission	interface	ct_23_pc	incorrect argument list
omission	interface	ct_2_cd	incorrect argument list
omission	interface	ct_f1	incorrect argument list
omission	interface	ct_f1	incorrect argument list
omission	interface	ct_f2	incorrect argument list
omission	interface	ct_f2	incorrect argument list
omission	interface	ctrl_23_26	incorrect argument list
omission	interface	ctrl_21_24	incorrect argument list
omission	interface	ctrl_4_saving	incorrect argument list
omission	interface	ctrl_9_sel	incorrect argument list
omission	interface	fill_menu_txt	incorrect argument list
omission	interface	ld_9	incorrect argument list
omission	interface	load_23_ty_des	incorrect argument list

MODIFICATION TYPE	FAULT CLASS	BOX NAME	DESCRIPTION
omission	interface	popup_menu	incorrect argument list
optimization	n/a	cr_def_ty_1	incorporate more efficient file handling function
optimization	n/a	cr_head_node	eliminate subroutine
optimization	n/a	ct_21_bs	convert while/do to 'c' function call
optimization	n/a	ct_21_del	eliminate unnecessary assignment statement
optimization	n/a	ct_21_la	eliminate unnecessary 'c' function call
optimization	n/a	ct_23_bs	convert while/do to 'c' function call
optimization	n/a	ct_23_bs	eliminate if/then/else construct
optimization	n/a	ct_23_cd	convert while/do to 'c' function call
optimization	n/a	ct_23_cr	convert while/do to 'c' function call
optimization	n/a	ct_23_la	eliminate unnecessary 'c' function call
optimization	n/a	ct_23_ra	eliminate unnecessary 'c' function call
optimization	n/a	ct_2_bs	convert while/do to 'c' function call
optimization	n/a	ct_2_bs	eliminate if/then/else construct
optimization	n/a	ct_2_del	eliminate unnecessary assignment
optimization	n/a	ct_2_del	convert while/do to 'c' function call
optimization	n/a	ct_2_pc	incorporate test of current character
optimization	n/a	ct_9_fill	eliminate subroutine call
optimization	n/a	ct_9_fill	eliminate unnecessary print statements
optimization	n/a	ct_f2	eliminate unnecessary assignment statements
optimization	n/a	ctrl_23_26	eliminate unnecessary call to subroutine
optimization	n/a	ctrl_6_ask	eliminate part in case statement
optimization	n/a	ctrl_9_sel	eliminate unnecessary assignment
optimization	n/a	ctrl_9_sel	eliminate unnecessary string
optimization	n/a	draw_menu_oln	eliminate function call
optimization	n/a	fill_w3_txt	eliminate unnecessary assignment statement
optimization	n/a	get_9_ch	eliminate redundant code
optimization	n/a	get_edit_ch	subroutine eliminated
optimization	n/a	get_menu_ch	incorporate modula operation of pick-list window
optimization	n/a	get_menu_ch	consolidate part statements
optimization	n/a	instan_node	eliminate unnecessary string assignment
optimization	n/a	load_record	substitute if/then for if/then/else construct

Modifications as a Result of Code Review

MODIFICATION TYPE	FAULT CLASS	BOX NAME	DESCRIPTION
commission	computation	ct_23_cd	incorrect arithmetic operator
commission	computation	ct_23_ua	incorrect arithmetic operator
commission	computation	ct_2_bs	incorrect assignment to local variable
commission	control	append_c_s	incorrect predicate in an if/then construct
commission	control	ct_23_pc	incorrect predicate in an if/then construct
commission	control	ct_23_pc	incorrect predicate in an if/then construct
commission	control	ct_fl	incorrect predicate in an if/then construct
commission	control	ct_fl	incorrect predicate in an if/then construct
commission	control	ct_fl	incorrect predicate in an if/then construct
commission	control	ctrl_21_24	mismatched braces
commission	control	get_cur_index	incorrect argument list in 'c' function call
commission	control	get_menu_ch	incorrect predicate in a case statement
commission	control	load_cur_node	incorrect use of 'dot' operator
commission	control	load_cur_node	incorrect use of 'dot' operator
commission	control	load_cur_node	incorrect use of 'dot' operator
commission	control	load_cur_node	incorrect use of 'dot' operator
commission	cosmetic	ctrl_21_24	incorrect spelling of a variable
commission	cosmetic	ctrl_21_24	incorrect spelling of a variable
commission	cosmetic	ctrl_2_11	incorrect spelling of a function
commission	cosmetic	ctrl_2_11	incorrect spelling of a function
commission	cosmetic	ctrl_31	incorrect spelling in message
commission	cosmetic	ctrl_31	incorrect spelling in message
commission	data	ct_21_cd	incorrect initialization of local variable
commission	data	ctrl_21_24	incorrect pointer assignment
commission	initialization	ct_21_cd	incorrect initialization of local variable
commission	initialization	ct_2_cd	incorrect declaration of variable
commission	initialization	ct_fl	incorrect declaration of variable
commission	initialization	ctrl_9	incorrect pointer assignment
commission	initialization	ctrl_9	incorrect pointer assignment
commission	initialization	ctrl_9	incorrect pointer assignment
commission	initialization	ctrl_9	incorrect pointer assignment
commission	initialization	ctrl_9	incorrect pointer assignment
commission	initialization	get_cur_index	incorrect declaration of variable
commission	interface	cr_def_ty_1	incorrect argument list in 'c' function call
commission	interface	ct31_del_bbs	incorrect argument list
commission	interface	ct31_ins_lines	incorrect argument list in 'c' function call
commission	interface	ct_21_cr	incorrect argument list
commission	interface	ct_21_pc	incorrect argument list
commission	interface	ct_23_cd	incorrect argument list in 'c' function call
commission	interface	ct_23_cd	incorrect argument list
commission	interface	ct_23_cr	incorrect argument list in 'c' function call
commission	interface	ct_23_da	incorrect argument list
commission	interface	ct_23_del	incorrect argument list
commission	interface	ct_23_del	incorrect argument list in 'c' function call
commission	interface	ct_23_pc	incorrect argument list
commission	interface	ct_23_ra	incorrect argument list
commission	interface	ct_23_ua	incorrect argument list
commission	interface	ct_2_bs	incorrect argument list in 'c' function call
commission	interface	ct_2_del	incorrect argument list in 'c' function call
commission	interface	ct_fl	incorrect argument list
commission	interface	ct_fl	incorrect argument list
commission	interface	ctrl_21_24	incorrect argument list
commission	interface	ctrl_21_24	incorrect argument list
commission	interface	ctrl_21_24	incorrect argument list
commission	interface	ctrl_21_24	incorrect argument list
commission	interface	ctrl_21_24	incorrect argument list
commission	interface	ctrl_21_24	incorrect argument list
commission	interface	ctrl_21_24	call by value should be call by reference
commission	interface	ctrl_21_24	call by reference should be call by value
commission	interface	ctrl_23_26	incorrect argument list

MODIFICATION TYPE	FAULT CLASS	BOX NAME	DESCRIPTION
commission	interface	ctrl_23_26	incorrect argument list
commission	interface	ctrl_23_26	incorrect argument list
commission	interface	ctrl_23_26	incorrect argument list
commission	interface	ctrl_23_26	incorrect argument list
commission	interface	ctrl_23_26	incorrect argument list
commission	interface	ctrl_23_26	incorrect argument list
commission	interface	draw_menu_oln	incorrect argument list in 'c' function call
commission	interface	fill_menu_text	incorrect argument list in 'c' function call
commission	interface	fill_menu_text	incorrect argument list in 'c' function call
commission	interface	fill_menu_text	incorrect argument list in 'c' function call
commission	interface	fill_menu_text	incorrect argument list in 'c' function call
commission	interface	fill_menu_text	incorrect argument list in 'c' function call
commission	interface	fill_menu_text	incorrect argument list in 'c' function call
commission	interface	init_struct	incorrect argument list
commission	interface	ld_9	incorrect argument list
commission	interface	update_tmp_f	incorrect argument list in 'c' function call
omission	control	ct_21_cr	incorrect predicate in an if/then construct
omission	control	ct_21_cr	incorrect predicate in an if/then construct
omission	control	ct_23_cr	omitted required if/then construct
omission	control	ct_23_pc	mismatched braces
omission	control	ct_f1	incorrect predicate in an if/then construct
omission	control	ctrl_21_24	failure to include "break" in case statement
omission	control	ctrl_21_24	mismatched braces
omission	control	ctrl_31	omitted required "break" in a case statement
omission	control	pre_order	incorrect predicate in an if/then construct
omission	cosmetic	ctrl_2_11_22	failed to place cursor in the appropriate position
omission	data	ct_23_cr	omitted required 'c' function call
omission	initialization	blanks	incorrect pointer assignment
omission	initialization	blanks	incorrect pointer assignment
omission	initialization	blanks	incorrect pointer assignment
omission	initialization	blanks	incorrect pointer assignment
omission	initialization	ct_21_bs	failure to declare local variable
omission	initialization	get_tlb_name	incorrect argument list
omission	interface	box_generator	incorrect argument list
omission	interface	cr_def_ty_f	incorrect argument list
omission	interface	ct_23_bs	incorrect argument list
omission	interface	ct_23_bs	incorrect argument list in 'c' function call
omission	interface	ct_f2	incorrect argument list
omission	interface	ct_f2	incorrect argument list
omission	interface	ctrl_2_11	incorrect argument list
omission	interface	ctrl_2_11	incorrect argument list
omission	interface	ctrl_2_11	incorrect argument list
omission	interface	ctrl_2_11	incorrect argument list
omission	interface	ctrl_2_11	incorrect argument list
omission	interface	ctrl_2_11_22	incorrect argument list
omission	interface	ctrl_9	incorrect argument list
omission	interface	init_struct	incorrect argument list
omission	interface	popup_menu	incorrect argument list
optimization	n/a	cr_def_ty_f	eliminate unnecessary local variable
optimization	n/a	cr_def_ty_f	eliminate duplicate prints statements
optimization	n/a	cr_def_ty_l	eliminate unnecessary assignment
optimization	n/a	ct_23_bs	eliminate unnecessary 'c' function call
optimization	n/a	ct_23_la	eliminate unnecessary 'c' function call
optimization	n/a	ct_23_ra	eliminate unnecessary 'c' function call
optimization	n/a	init_struct	free dynamic memory
optimization	n/a	make_windows	eliminate unnecessary 'c' function call
optimization	n/a	make_windows	eliminate unnecessary 'c' function call
optimization	n/a	make_windows	eliminate unnecessary 'c' function call
optimization	n/a	make_windows	eliminate unnecessary 'c' function call
optimization	n/a	make_windows	eliminate unnecessary 'c' function call

MODIFICATION TYPE	FAULT CLASS	BOX NAME	DESCRIPTION
optimization	n/a	make_windows	eliminate unnecessary 'c' function call
optimization	n/a	make_windows	eliminate unnecessary 'c' function call
optimization	n/a	make_windows	eliminate unnecessary 'c' function call
optimization	n/a	make_windows	eliminate unnecessary 'c' function call
optimization	n/a	make_windows	eliminate unnecessary 'c' function call

APPENDIX E

SCREEN TRANSITION MATRIX

Stimulus List Screen

NEXT EVENT	PREVIOUS EVENT								
	INIT	STR	OTH	CR	F1	F2	F4	F6	F7
STR	0.93	0.00	0.20	0.60	0.00	0.00	0.00	0.00	0.00
OTH	0.01	0.02	0.03	0.02	0.00	0.00	0.00	0.00	0.00
CR	0.01	0.60	0.20	0.00	0.00	0.00	0.00	0.00	0.00
F1	0.01	0.10	0.20	0.10	0.00	0.00	0.00	0.00	0.00
F2	0.01	0.04	0.03	0.04	0.00	0.00	0.00	0.00	0.00
F4	0.01	0.20	0.30	0.20	0.00	0.00	0.00	0.00	0.00
F6	0.01	0.02	0.02	0.02	0.00	0.00	0.00	0.00	0.00
F7	0.01	0.02	0.02	0.02	0.00	0.00	0.00	0.00	0.00
S3	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
S8	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
S22	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
S23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
S24	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00

APPENDIX F

SAMPLE TEST CASE

Test Case : 1

Pass _____ Fail _____

1. ___ On screen S2 do event STR and expect
2. ___ On screen S2 do event F4 and expect S21
3. ___ On screen S21 do event STR and expect
4. ___ On screen S21 do event CR and expect
5. ___ On screen S21 do event F6 and expect S22
6. ___ On screen S22 do event STR and expect
7. ___ On screen S22 do event F4 and expect S21
8. ___ On screen S21 do event STR and expect
9. ___ On screen S21 do event F4 and expect S24
10. ___ On screen S24 do event STR and expect
11. ___ On screen S24 do event F4 and expect S26
12. ___ On screen S26 do event STR and expect
13. ___ On screen S26 do event F1 and expect S3
14. ___ On screen S3 do event s and expect
15. ___ On screen S3 do event ENDD and expect
16. ___ On screen S26 do event STR and expect
17. ___ On screen S26 do event F1 and expect S3
18. ___ On screen S3 do event B and expect S6
19. ___ On screen S6 do event y and expect
20. ___ On screen S6 do event EXITT and expect

VITA

Charles C. Senn Jr. was born in Bethesda, Maryland, on February 3, 1951. He graduated from McMinn County High School, Athens, Tennessee, in June, 1969, and in July reported to the United States Naval Academy where he had been awarded a congressional appointment. He graduated in June, 1973 and accepted a commission as a Second Lieutenant in the United States Marine Corps. In January, 1975, he was designated a Naval Flight Officer and served as a Radar Intercept Officer aboard the McDonnell Douglas F4 Phantom until February 1980, when he resigned his commission. For the next four years Mr. Senn operated a small residential construction business in Beaufort, South Carolina. In July, 1984, he accepted a position with Hiwassee Builders Supply, Athens, Tennessee, as a computer systems manager, and three years later, enrolled as a graduate student in the University of Tennessee's Computer Science Department. Mr. Senn is currently employed with IBM and he and his family reside in Charlotte, North Carolina.