SOFTWARE ENGINEERING LABORATORY SERIES SEL-85-005

SOFTWARE VERIFICATION AND TESTING

DECEMBER 1985

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES SEL-85-005

SOFTWARE VERIFICATION AND TESTING

DECEMBER 1985

NASA
National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by he National Aeronautics and Space Administration, Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (System Developement and Analysis Branch)
The University of Maryland (Computer Sciences Department)
Computer Sciences Corporation (Flight Systems Operation)

the goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document. A version of this document was also issued as Computer Sciences Corporation document CSC/TM-85/6086.

The primary contributors to this document are

Dave Card	(Computer Sciences Corporation)
Betsy Edwards	(Goddard Space Flight Center)
Frank McGarry	(Goddard Space Flight Center)
Cindy Antle	(Computer Sciences Corporation)

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 552
NASA/GSFC
Greenbelt, Maryland 20771

ABSTRACT

General procedures for software verification and validation are provided as a guide for managers, programmers, and analysts involved in software development. The verification and validation procedures described are based primarily on testing techniques. Testing refers to the execution of all or part of a software system for the purpose of detecting errors. Planning, execution, and analysis of tests are outlined in this document. Code reading and static analysis techniques for software verification are also described.

TABLE OF CONTENTS

0022

TABLE OF CONTENTS (Cont'd)

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION 1 - INTRODUCTION

This document describes a recommended set of verification and validation activities that software developers should perform to ensure the delivery of reliable, cost-effective software. Its intent is to provide a practical reference for managers, programmers, and analysts. Most of these verification and validation activities involve testing. These terms are defined as follows:

- Verification is the process of ensuring that each intermediate product of each phase of the software life cycle is consistent with previous phases. This document describes the use of static analysis, code reading, and testing to ensure that the software is a faithful rendition of the design.
- Validation is the process of ensuring that each intermediate product of each phase of the software life cycle accurately responds to the underlying system requirements. This document describes the use of testing techniques to ensure that requirements are reflected in the software.

The objective of testing is to ensure that a reliable and effective product is delivered to the customer. Nevertheless, testing cannot substitute for an effective software implementation; it can only detect its absence. Together with quality assurance and configuration management, however, testing and especially nontesting verification techniques (Section 2) provide the tools that detect problems before their effects become too great.

Departures from the general procedures for static analysis, code reading, and testing outlined in this document must be specified in the pertinent software development plan

(Reference 1). Guidelines for configuration management and quality assurance are described in Reference 2.

The following subsections identify the verification and validation activities to be discussed and show where they occur in the software life cycle. Sections 2 through 7 describe every activity area/phase in detail. Section 8 reiterates the major points made elsewhere in the document.

1.1 VERIFICATION AND VALIDATION ACTIVITIES

Software developers employ many different terms to label verification and testing activities. Often these terms are not consistent even within a single software development organization. This section provides simple definitions of some of the most common terms. The software verification and testing activities discussed in this document include the following:

- Static Analysis--Evaluation of a program by a software tool that does not execute the program
- Dynamic Analysis--Evaluation of a program by monitoring its behavior during execution
- Code Reading--Examination of compiled code by someone other than the developer to identify faults
- Software Inspection--Formal review of a program including the design description, source code, and test plans
- Debugging--Trial-and-error process whereby a programmer attempts to isolate the cause of an error detected during testing
- Unit Testing--Execution of part of a system under controlled conditions to detect errors in internal logic and algorithms

- Integration Testing--Execution of two or more modules under controlled conditions to detect errors in intermodule logic and interfaces
- Build Testing--Execution (in accordance with a test plan) of a set of modules, comprising one or more complete functions, to detect errors
- System Testing--Execution of a complete system as directed by a test plan to detect errors
- Acceptance Testing--Execution of a completed system by nondeveloper personnel to demonstrate that the system satisfies requirements
- Regression Testing--Repetition of previous tests after a system has been changed to ensure that the change has had no unintended effects
- Certification--Process of declaring that a system is acceptable and ready for operation

1.2 APPROACHES TO TESTING

Testing is a primary tool for ensuring the effectiveness and reliability of software. It involves not only running tests but also designing, preparing, and evaluating them. Testing compares the software product against a statement of what was intended, either the requirements specification or the design description. Effective testing will identify significant problems so that some resolution of them can be made prior to software delivery. Figure 1-1 illustrates the test process.

Testing is not debugging (the informal process by which a programmer attempts to isolate the cause of a previously identified failure). Testing starts with known conditions, uses predefined procedures, and attempts to produce predictable outcomes. Only the success or failure of the test is undetermined.

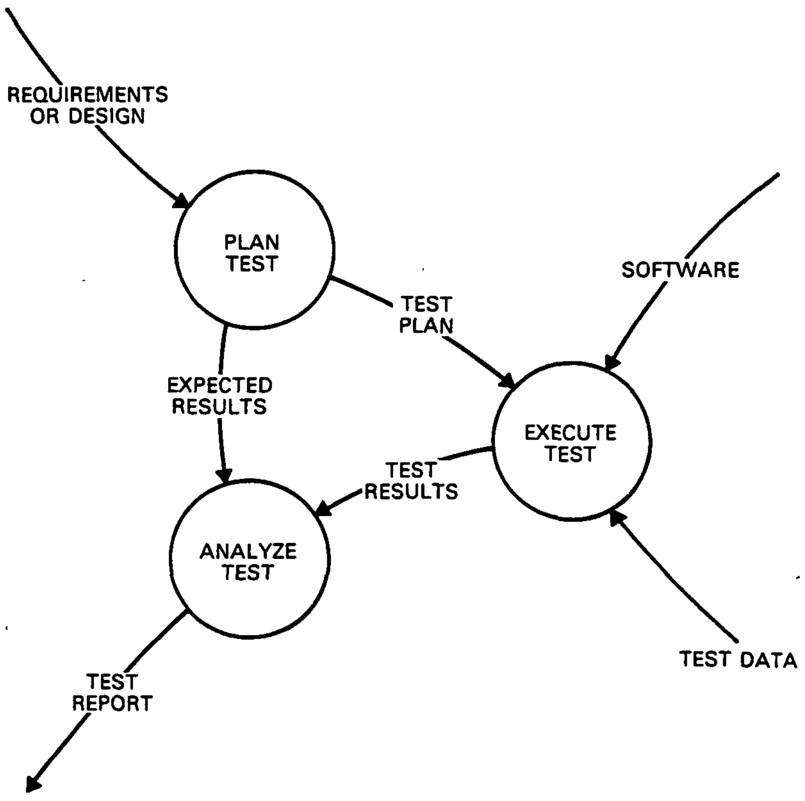


Figure 1-1. The Testing Process

Debugging starts from unknown conditions and its end cannot be predicted. It continues until the error is found.

The following subsections discuss three general strategies for designing tests (i.e., selecting test cases): functional, structural, and operational. Each of these has strengths and weaknesses. A successful tester will employ all of these techniques to some extent. These strategies apply to all phases of testing to some degree.

1.2.1 FUNCTIONAL TESTING

Functional testing is one strategy for designing and selecting test cases. It treats the software like a "black box." Input is supplied, and output is observed. Comparison of the software (requirements) specification with the observed input/output relationship indicates whether an error is present in the software. Functional testing does not require the tester to have any knowledge of the design or operation of the software.

For functional testing to be complete, all possible input must be tested. Clearly, this is not practical for a program of any significant size. Several strategies may be followed to define smaller sets of input for testing that still provide reasonable confidence in the system.

One informal approach is to identify the major functional capabilities specified in the requirements, then select a set of input values that fully exercise those functions. Handling of boundary values and error conditions as well as the main processing stream should be tested. This approach usually drives acceptance test planning.

Other formal approaches to functional testing include equivalence partitioning (Reference 3) and statistical testing (Reference 4). Equivalence partitioning reduces the set of possible input values to a subset of functionally equivalent classes. Statistical testing selects a random subset from among the possible test cases.

1.2.2 STRUCTURAL TESTING

Structural testing is another strategy for designing and selecting test cases. As opposed to functional testing, it treats the software like a "white box." Tests are specified based on an examination of the software structure (rather than the specification). Structural testing compares the detailed system design to its software implementation. Ideally, structural tests should be based on the program design language (PDL) and developed at the same time as the PDL. Structural tests alone do not provide a mechanism for verifying the software against the specification.

Coverage (the degree to which the software is exercised) serves as the basic criterion for completion of structural testing (Reference 3). Three levels of coverage are recognized: Statement coverage requires that every statement in the source code be executed at least once. Condition (or branch) coverage requires that each outcome of every decision be executed at least once. Path coverage requires that every possible sequence of decision outcomes be executed. (Path testing can lead to an impractically large number of tests for a program of any significant size or complexity. For example, backward transfers can produce an infinite number of paths.) In practice, ensuring that every statement and every condition (but not necessarily all combinations of them) is executed provides adequate coverage.

1.2.3 OPERATIONAL TESTING

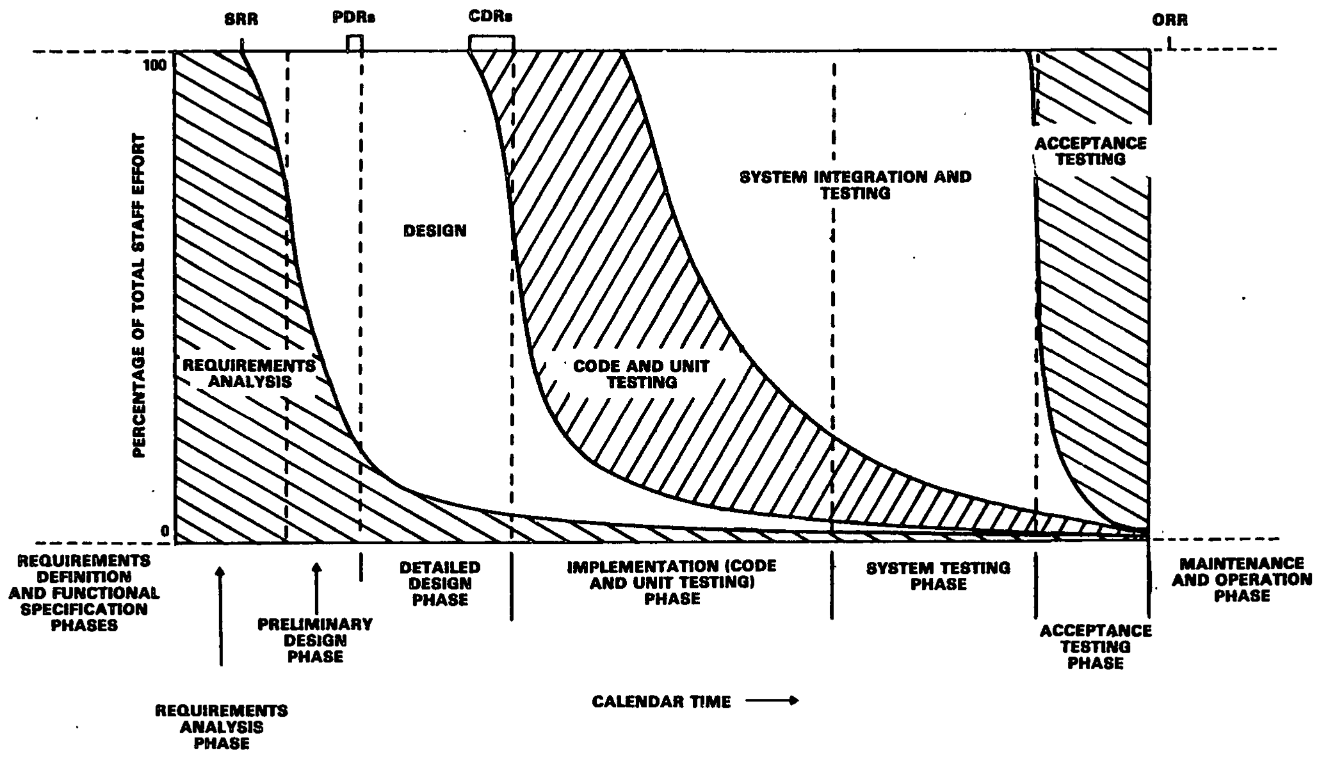
Operational testing is a strategy based on simulating the expected operational environment. Functionally and structurally based testing tend to consider the software in isolation. Yet, other factors such as hardware, operator actions, and the physical environment affect the functioning of software. These factors may not be fully evaluated in a controlled test environment. Frequently, system testing occurs on a development computer rather than the eventual operational computer. These computer configurations can differ significantly. Consequently, developers may adopt operational testing techniques as part of the acceptance testing process. (Acceptance testing should always be performed on the operational computer.)

Operational testing may be performed by simulating the expected workload and environment, running the system in parallel with the existing manual or automatic system, or making cautious use of the system for actual production. Simulating operational conditions and running in parallel can be expensive. Frequently, large systems are released tentatively for operation to multiple users ("beta testing"). During this test period, users record errors and report them to the developers.

1.3 FLIGHT DYNAMICS LIFE CYCLE

This document is based on experience in and analysis of flight dynamics software development. Flight dynamics applications include spacecraft attitude determination/control, orbit adjustment, mission planning, and support tools. Most of these projects are developed in FORTRAN on IBM mainframes or DEC minicomputers. Typical projects produce from 30,000 to 150,000 source lines of code. Schedules range from 13 to 21 months. Figure 1-2 shows the software life cycle for flight dynamics. Reference 1 describes the software life cycle and other software development activities in more detail.

Software verification and validation activities span the entire software life cycle, not just system and acceptance testing. Test planning occurs during requirements analysis and design. Testing begins in implementation and continues throughout the life of the software product. Table 1-1 identifies the major life cycle verification and validation activities and the relevant techniques. The rest of this document explains these verification and validation activities in detail.



NOTE: FOR EXAMPLE, AT THE END OF THE IMPLEMENTATION PHASE (4TH DASHED LINE), APPROXIMATELY 79% OF THE STAFF ARE INVOLVED IN SYSTEM INTEGRATION AND TESTING; APPROXIMATELY 2% ARE ADDRESSING REQUIREMENTS CHANGES OR PROBLEMS; APPROXIMATELY 2% ARE DESIGNING MODIFICATIONS; AND APPROXIMATELY 17% ARE CODING AND UNIT TESTING CHANGES.

Figure 1-2. Software Life Cycle

Table 1-1. Application of Techniques

	Life-Cycle Activities			
Technique	Coding, Unit Testing, Debugging	Integration Testing	System Testing	Acceptance Testing
Static Analysis	X	-	-	-
Dynamic Analysis	X	-	-	-
Code Reading	X	-	-	-
Structural Testing	X	X	X	X
Functional Testing	X	X	X	X
Regression Testing	-	X	X	X
Operational Testing	-	-	-	X

NOTE: X indicates technique is used for this activity.

SECTION 2 - NONTESTING VERIFICATION TECHNIQUES

Static analysis and code reading are nontesting verification techniques that detect many common errors. Static analysis usually precedes code reading. All newly developed code must be read. Especially critical or complex modules may also undergo a formal software inspection as part of the quality assurance process (see Reference 2). The technical manager must allocate sufficient time and resources for these activities during implementation. Experience shows that the benefits of code reading exceed its cost, so skimping on this activity will ultimately prove detrimental.

2.1 STATIC ANALYSIS

Static analysis consists of processing a coded module through a software tool to detect errors and identify problem areas. Examples of such tools include compilers, interface checkers, and source analyzers. The programmer's first step after coding a module is usually to compile it. In addition to the syntax errors reported, most compilers offer other software aids. The cross-reference listing is especially useful. Undeclared arrays and misspelled variable names, for example, stand out with only a cursory inspection of this listing.

Processing a module through the Static Source Code Analyzer Program (Reference 5) can identify problem areas. For example, modules of exceptional complexity (many descendants) or with a large number of unreferenced variables are especially error-prone. Code readers should give extra attention to these modules.

An interface checker, such as RXVP80 (Reference 6), processes a subsystem or complete system to detect inconsistent calling sequences and external references. Because it handles groups of modules, the interface checker may only be applied after some related modules (e.g., a subsystem) have been coded and read.

Generally, static analysis should precede code reading. The results (clean compile, cross-reference, SAP report, etc.) of any static analysis performed should be provided to the code reader. The next section describes some common code-reading techniques.

2.2 CODE READING

Code reading is a systematic procedure for reading and understanding the operation of a program. An experienced developer (not the original programmer) reads code to determine if a program is correct with respect to its intended function. Techniques of code reading include checklists and simulated execution. In practice, developers employ some aspects of both techniques. Some general guidelines for code reading are as follows:

- The code reader need not be a member of the development team.
- The developer should not be present during the code-reading process.
- The intended function of the code must be extractable from the commentary or a separate explanation must be provided.
- Only cleanly compiled code should be presented for code reading.
- Code should be submitted for reading in increments that require 1/2 hour or less of the reader's time (i.e., one to four subroutines).
- Code must be read and returned with comments to the developer within 1/2 day.

- The code-reading process must be recorded in the unit development folder (i.e., date read and by whom).

Code reading is not intended to redesign, evaluate alternatives, or find solutions; its purpose is to find faults and deficiencies. Corrections should be made by the original developer. The technical manager must ensure that all code is read and corrections are made.

The first step in code reading is to extract the intended function of the module by reviewing the code and supporting materials. The following subsections explain two approaches to detecting problems--checklists and simulated execution--and indicate the areas in which each is especially effective. Careful code reading will result in clear, complete, and correct code.

2.2.1 CHECKLISTS

The checklist approach to code reading is the simplest method of code reading. Using this technique, the reader examines the source code to answer a predefined set of questions. Checklists (of questions) are a very effective means of detecting certain common errors and ensuring conformance to standards. Major areas of concern are as follows:

- Verify that calling sequences and argument lists agree in number, type, and units
- Verify that all local variables are defined, initialized, and used
- Verify that indices are correctly initialized and used
- Verify that file definitions and unit numbers are consistent and match the design

- Verify that a path is defined for every outcome of a logical decision (providing a default if necessary)
- Check that the presentation of the code is clear and adequately documented

The Appendix provides a sample checklist for finding common errors. Reference 7 provides more information.

2.2.2 SIMULATED EXECUTION

Another approach to code reading is to simulate the execution of the module under review manually. The reader traces the flow of input data through the module, recording the current values of all variables on a separate sheet of paper. A compiler-generated cross-reference listing provides a good template for this record. At the end of execution or exit from the module, these values are compared with the design specification and any discrepancies are noted.

The input values may be either <u>symbolic</u> (e.g., using T to represent time) or <u>typical</u> (similar to those likely to be encountered in actual operation). Simulated execution is a good technique for uncovering logic errors (especially in complex algorithms) but may fail to detect other types of errors. Code readers should use it in conjunction with other techniques.

SECTION 3 - UNIT TESTING

Unit testing is the process of executing a functional subset of a software system to determine that it performs its assigned function. All modules must be unit tested. Especially complex or critical modules may be unit tested in isolation. Usually the individual programmer develops an informal plan and performs the testing.

3.1 PLANNING

Unit testing does not require the formal planning and procedures that apply to system and acceptance testing. However, some general guidelines for preparing unit tests are as follows:

- A unit must include one or more logically related modules.
- Test data, the test driver, and stubs must be developed by the programmer.
- Unit testing takes place only after the code has been read (Section 2.2).
- After successful unit testing, modules must be submitted to the controlled library.

3.2 EXECUTION

Figure 3-1 illustrates the unit testing process. For a module tested in isolation, unit testing begins by developing a test driver and stubs. Alternatively, a high-level driver that is planned to be part of the final system may assume the role of test driver for a group of modules. Any dummy routines are combined with the modules to be tested in an executable load module. This load module is executed using test data developed by the programmer. Small tested units may be combined to form larger units for further testing.

In this case, programmers may adopt some of the techniques described in Section 4 for integration testing.

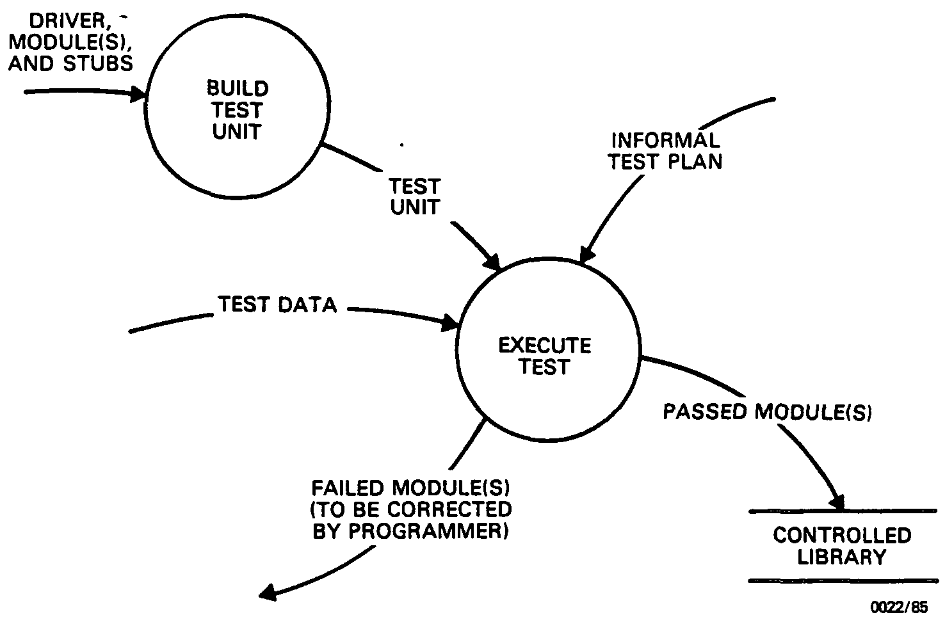


Figure 3-1. Unit Testing Process

In the test driver, input variables are initialized and passed to the routine to be tested. Additional debug output features may be added to the driver. An example of a test driver is shown in Figure 3-2. In this example, values of the input and output variables are saved for later inspection.

A FORTRAN stub consists of a SUBROUTINE statement, a line of debug output, and a RETURN, as follows:

```
    SUBROUTINE STUBX (VAR1, VAR2)
    WRITE (1, 100)
100 FORMAT (' *** ENTERED STUBX ***')
    RETURN
    END
```

Dynamic debugging tools may help to monitor and analyze unit tests.

ORIGINAL PAGE IS
OF POOR QUALITY

```
       PROJECT:  ZBCBA                         MEMBER: DOCTEST                          DATE: 85/07/25
       LIBRARY:  TASK449                                                                TIME: 15:45
       TYPE:     FORT                                                                   PAGE: 1
START
 COL   ----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8
 ---------------------------------------------------------------------------------------------
   1   C
   1   CC    -THIS IS A TEST DRIVER USED FOR UNIT TESTING
   1   C
   1   CC    -DECLARE VARIABLES
   7          INTEGER*4 IDEBUG.IPRINT.ILEN
   7          REAL*4 NUM
   7          CHARACTER*8  NAME
   1   C
   1   CC    -INITIALIZE VARIABLES
   7          DATA NAME/'IPARM   '/
   7          IDEBUG = 4
   7          IPRINT = 6
   7          IRET = 0
   7          NUM = 0.
   7          ILEN = 0
   1   C
   1   CC    -DUMP INPUT VARIABLES
   7          WRITE(6.100)IDEBUG.IPRINT.NAME
   2    100   FORMAT(' **** ENTERING TEST DRIVER ****'./.
   6         *         ' DEBUG LEVEL IS:    '.I4.'     DEBUG UNIT IS:    '.I4./.
   6         *         ' NAME OF VARIABLE IS:    '.A8)

   1   C
   1   CC    -CALL ROUTINE TO BE TESTED
   7          CALL GEINIT(IRET.IDEBUG.IPRINT.NAME.ILEN.NUM)
   1   C
   1   CC    -DUMP OUTPUT VARIABLES
   7          WRITE(6.200)IRET.IDEBUG.IPRINT.NAME.ILEN.NUM
   2    200   FORMAT(' AFTER CALL OF ROUTINE TO BE TESTED './.
   6         *         ' RETURN CODE IS:  '.I4./.
   6         *         ' DEBUG LEVEL IS:    '.I4.'     DEBUG UNIT IS:    '.I4./.
   6         *         ' NAME OF VARIABLE IS:    '.A8./.
   6         *         ' ILEN =   '.I4.'    NUM =   '.F8.4./.
   6         *         ' **** RETURNING FROM TEST DRIVER **** ')
   1   C
   7           END
```

Figure 3-2. Example of a Test Driver

3.3 <u>ANALYSIS</u>

After testing, the output from the test driver and the tested modules is examined. If the test cases were handled satisfactorily, the unit is submitted to the controlled library. Transfer of modules to the controlled library must be approved by the technical manager. Subsequent changes to modules in the controlled library follow special procedures (Reference 2).

SECTION 4 - INTEGRATION TESTING

After unit testing is completed, all modules must be integration tested. A single integration team performs this function for the entire system. During integration testing, the system is slowly built up by adding one or a few modules at a time to the core of already integrated modules. The two basic approaches to incremental integration testing are top-down and thread testing (Reference 8). Figure 4-1 contrasts these approaches.

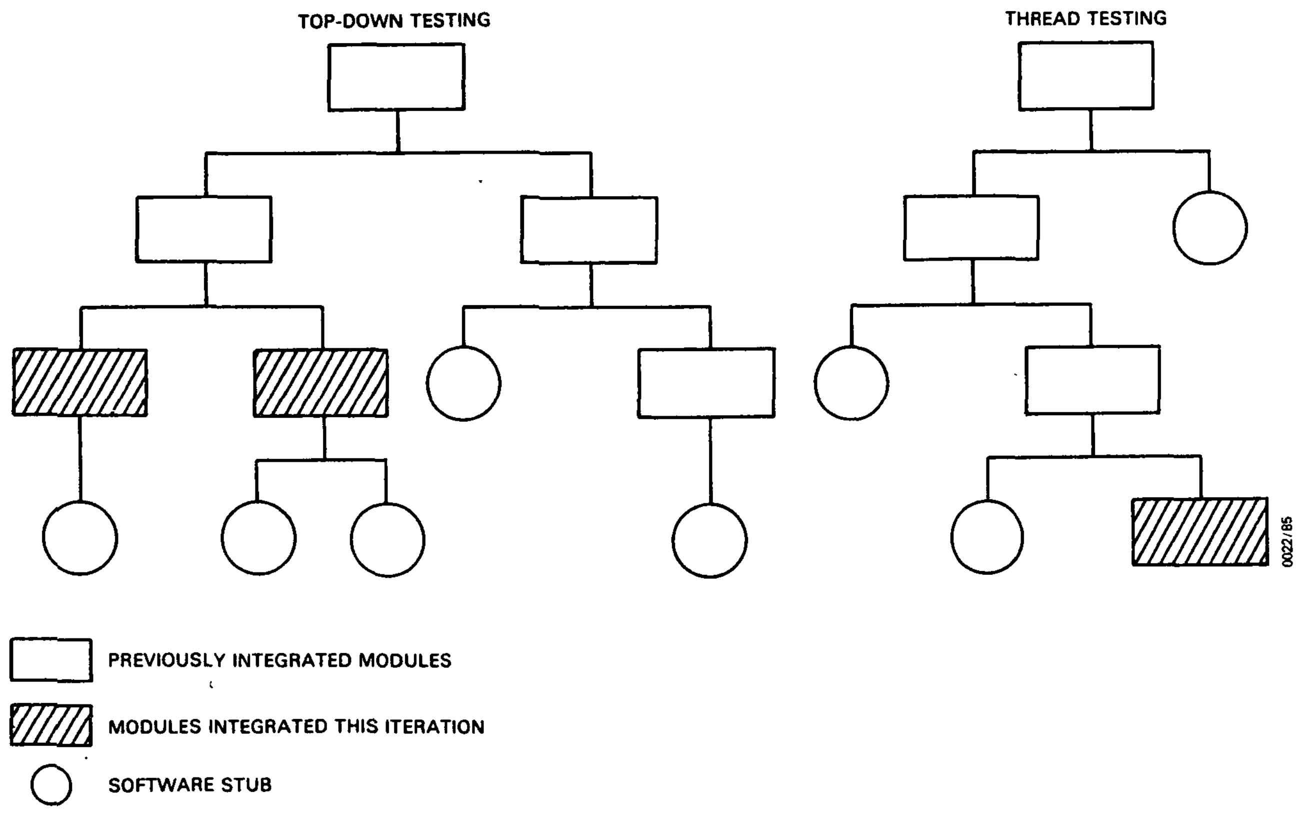

TOP-DOWN TESTING
THREAD TESTING
0022/85
PREVIOUSLY INTEGRATED MODULES
MODULES INTEGRATED THIS ITERATION
SOFTWARE STUB

Figure 4-1. Integration Testing Techniques

Both begin with the top-level driver and require the substitution of stubs for not-yet-integrated modules. Top-down testing integrates additional modules level by level. Thread testing builds a single end-to-end path that demonstrates a basic functional capability, then adds on to that. Top-down testing generally requires less time, but thread testing allows some operational capability to be demonstrated sooner.

4.1 PLANNING

Integration testing usually does not require the formal planning that applies to system and acceptance testing. However, its procedures are more carefully controlled than unit testing. The development of formal integration test plans must be specified in the software development plan if they are required. Some important points to consider are as follows:

- Usually, a subset of the development team performs all integration testing.
- This testing may rely on data and procedures defined in the system test plan, but formal integration test plans are usually not developed in this environment.
- Alternatively, integration testing may be based on the individual programmer's unit tests.
- Only modules already accepted into the controlled library may be integration tested.

4.2 EXECUTION

Integration testing follows the hierarchical system structure. Modules must be integrated a few at a time so that the location of newly detected errors will be constrained. Two or more modules on successive levels should never be integrated in the same test. Modules called by those currently being integrated are represented by stubs. Successive levels of modules are integrated into the system by replacing stubs with actual software modules. Additional stubs are added and replaced as needed. Figure 4-2 illustrates the integration testing process.

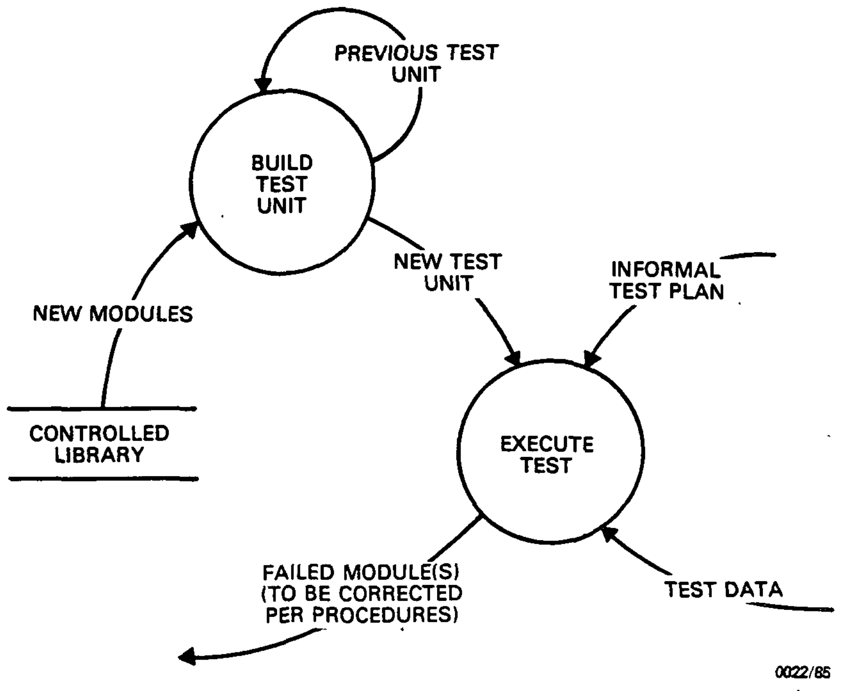


Figure 4-2. Integration Testing Process

4.3 ANALYSIS

Analysis of integration testing results is informal unless formal test plans are required. The integration team compares test results with design specifications to determine whether a problem exists. Problems are reported to the technical manager, who refers them to the original programmers for correction. Changes to modules in the controlled library must be approved by the technical manager and reported on change report forms (see Appendix). Changed modules must undergo unit testing again.

SECTION 5 - BUILD/SYSTEM TESTING

Large software systems are implemented as a series of increasingly complete partial systems called builds. A build is a portion of a software system that satisfies, wholly or in part, a subset of the requirements (Reference 9). The first build of a system consists of the most basic functional capabilities. New capabilities are added to each build, and each build must be tested separately. The final build comprises the whole system. In the case of a system implemented in three builds, the builds would typically be defined as follows:

1. Main drivers; data input subsystem; and interfaces to graphics, data base, and operating systems
2. Primary processing subsystem from input to output
3. Full system including results output subsystem, error handling, and secondary processing options

The capabilities of each build for a system must be defined in the software development plan (Reference 1). Generally, each build should add a major function to the system. Figure 5-1 provides an example of build definition.

Build and system testing are performed and evaluated by a system test team. The test plans and test results should be reviewed by a quality assurer (Reference 2). System test planning and procedures are formal. Some important points to consider include the following:

- Usually, system tests are planned and executed by a subset of the development team or by an independent test team.
- Testing time may have to be scheduled well in advance to ensure adequate computer resources.

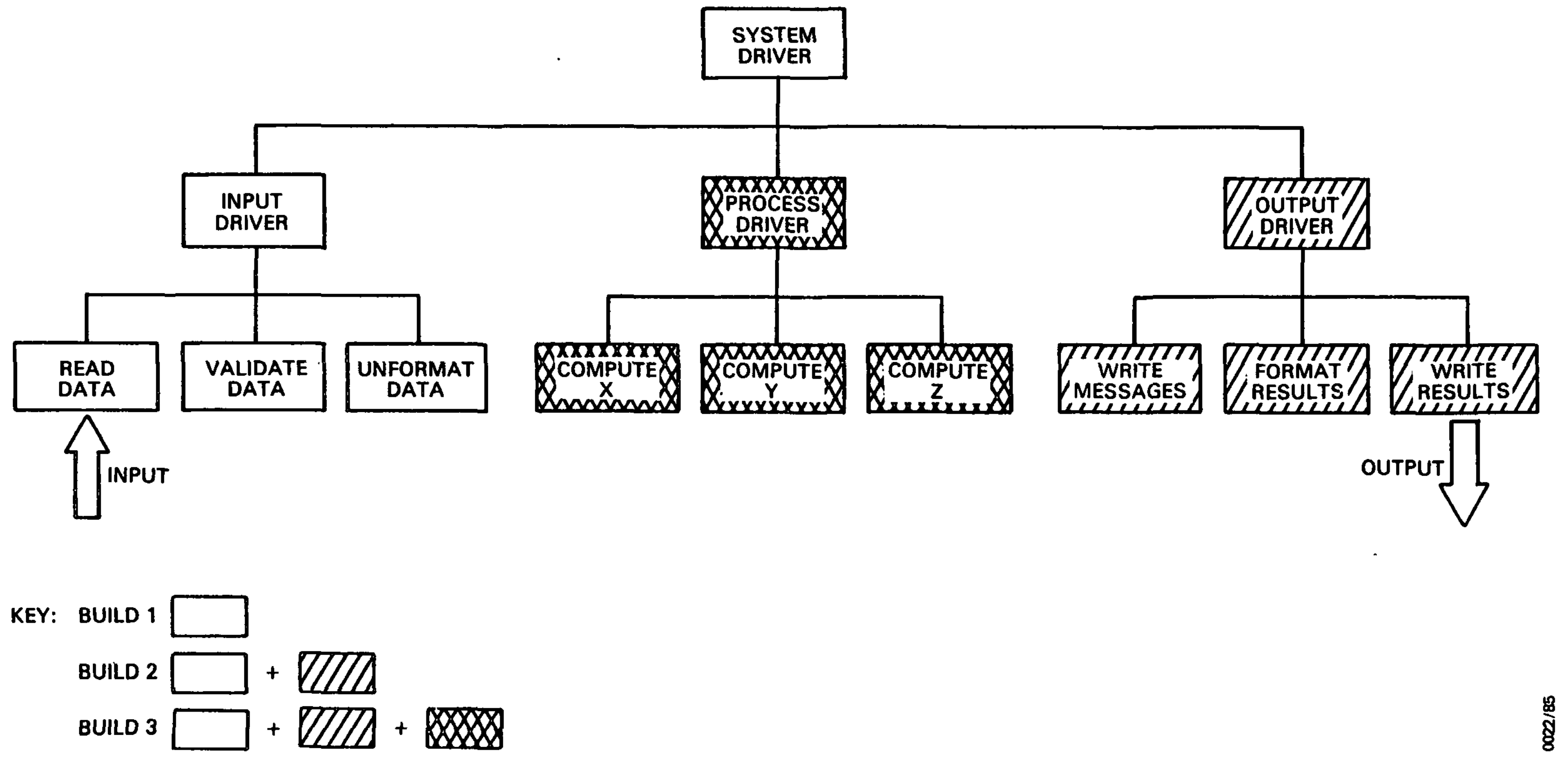


Figure 5-1. Definition of Builds

- Test plans, data, and results must be archived to provide for regression testing.
- Discrepancies and changes must be formally recorded and reviewed.
- System testing must probe both functional and operational requirements.

5.1 PLANNING

Build tests and system tests are very similar. Both build and system tests are based on the design (of a build or the system, respectively) and on the requirements. A set of formal system test plans must be produced by the development team and quality assured. Figure 5-2 shows the test planning process. Build tests may be subsets of the system tests. This, however, must be indicated in the system test plan, or separate build test plans must be developed. The complete description of a system test will include the following:

- Purpose of test, i.e., specific capabilities or requirements tested
- Detailed specification of input
- Required environment, e.g., data sets required, computer hardware necessary
- Operational procedure, i.e., how to do the test
- Detailed specification of output, i.e., the expected results

An example of an individual test plan is given in the Appendix. A complete test plan consists of a set of such individual plans.

Build/system test plans may be developed based on an analytical test plan or on the acceptance test plan prepared by the analysis team. An analytical test plan shows how the computational accuracy of the system can be verified. It may be incorporated in the build/system test plan by reference, or it may be partially rewritten and incorporated in the build/system test plan proper. In either case, the build/system test plan will have two parts: software tests to demonstrate conformance to operational requirements and analytical tests to demonstrate satisfaction of functional requirements. Effective testing depends largely on selecting appropriate test cases. Test cases should be selected to exercise fully the function, structure, capacity, and configuration of the system as described in Table 5-1.

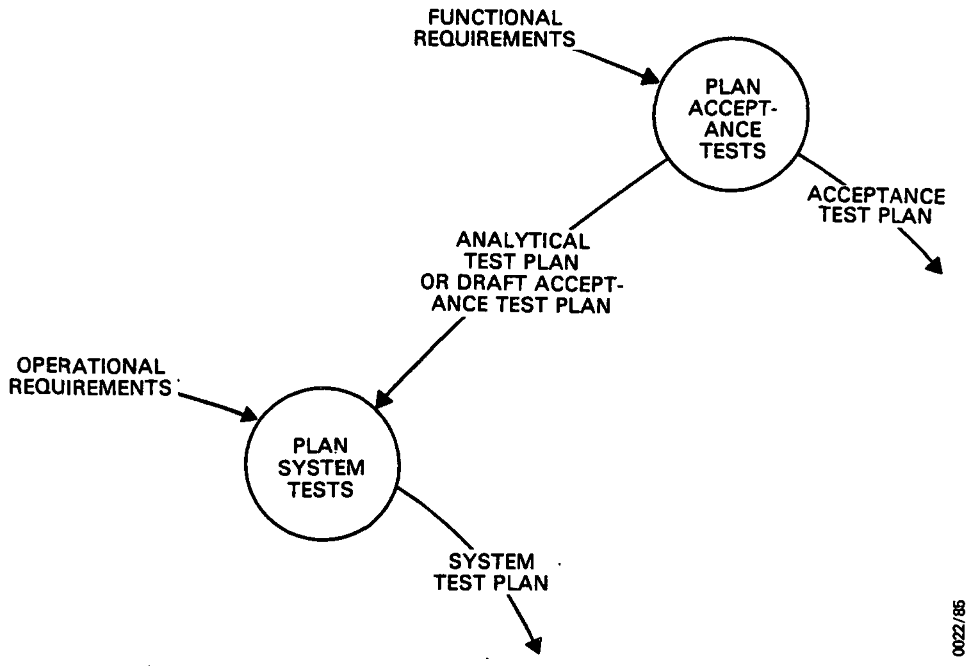


Figure 5-2. Test Planning Process

5.2 EXECUTION

Figure 5-3 outlines the steps involved in build/system testing. A system load module is built by the technical manager, or by the librarian at the direction of the technical manager, from the source in the controlled library. The load

Table 5-1. What To Test

Testing Area	Capabilities To Test
Function	Computational accuracy Invalid data handling Special case processing Boundary value handling
Structure	Data flow management Error recovery Control options Recycle and restart
Capacity	Volume of data processed Rate of data processing Resources consumed (e.g., CPU)
Configuration	Minimum hardware set Maximum hardware set Alternate processors

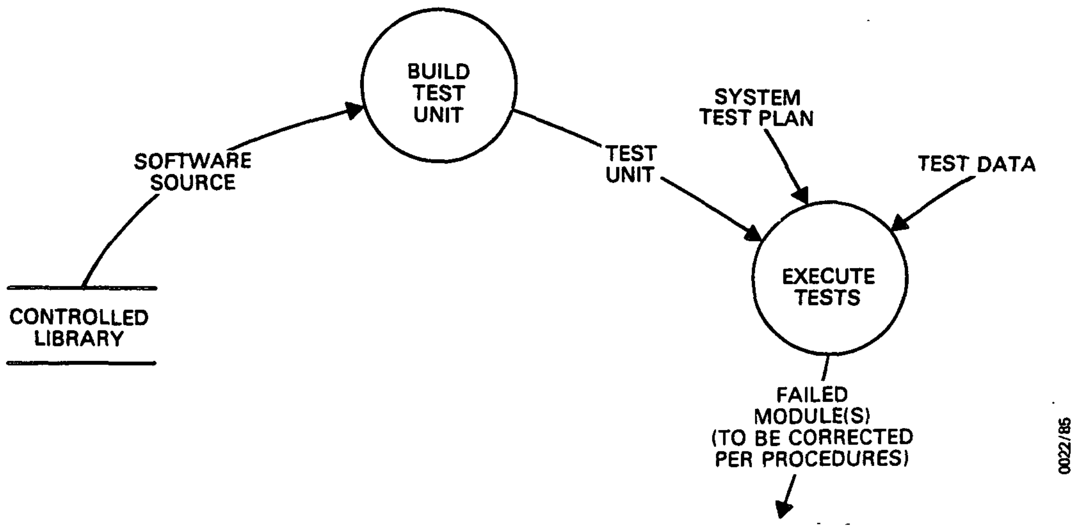


Figure 5-3. System Testing Process

module data set and the specific load module member being tested are managed by the librarian and must be identified on the test report. The test data, which may include control parameter data sets (e.g., JCL and NAMELISTs) as well as prepared data files, are generated at the time the test plans are written. These items are also managed by the librarian.

Test execution follows the pattern prescribed in the system test plan. All printed output is kept by the system test team for each test executed. When a separate organization for testing is not possible, developers on the system test team should test sections of the system developed by members other than themselves.

5.3 ANALYSIS

The build/system test results are reviewed by the development team and manager. Any discrepancies between system requirements and system performance are evaluated and assigned to a programmer for correction.

Changes are made in accordance with established configuration management procedures. The programmer must complete a change report form (CRF) detailing the nature of the change (Reference 2). The Appendix includes a sample CRF. Copies of all CRFs are kept by the technical manager. The technical manager is also responsible for tracking the discrepancies over time for each build and for the complete system.

A summary of testing progress should be prepared periodically, listing the tests started, tests completed, errors identified, and errors resolved. System testing continues until no more errors are identified. However, build/system tests must be planned to have a high probability of detecting errors if any are present. Testers should not try to avoid errors!

5.4 <u>TEST DATA</u>

Effective testing depends on the timely availability of appropriate test data. Test data must be created that support the test plans. The types of data sets needed and their contents are indicated in the test plan. Some important points to consider about test data include the following:

- The system test team is responsible for generating all system test data, including control (parameter and command) data sets.
- All system test data (including control parameter data sets) are kept under configuration control.
- All test data should be checked before testing. At the minimum, a listing of the data should be scanned by the test team for obvious errors.
- Test data may be created by a data generator or a simulator, or by manual entry.

Data generators produce random or systematically varying data that conform to specified characteristics such as mean, range, amplitude, and frequency. Data generators provide a simple, readily available source of test data, especially for integration testing.

A simulator reproduces the output from the actual operation of a device for input to a data processing system. Simulators are usually mission specific. All simulator software must itself be tested and its output verified. It is important, therefore, that simulator software development begin as early as possible to avoid becoming a bottleneck.

Developers manually create much of the test data used in software development. This is especially true for unit and integration testing. Developers also define control parameters (e.g., NAMELISTs), initialize COMMON variables via

BLOCK DATA routines, and provide interactive responses to prompts from the system.

5.5 <u>REGRESSION TESTING</u>

Regression testing is the testing that must be performed after functional improvements or repairs have been made to a system (Reference 8) to confirm that the changes have had no unintended side effects. Correction of errors relating to logic and control flow, computational errors, and interface errors are examples of conditions that necessitate regression testing. Cosmetic errors generally do not affect other capabilities and therefore do not require that regression testing be performed.

During system testing, regression testing is based on a specified subset of the system tests. It is performed by the system test team. Some regression testing must be performed for each new build of a system. This ensures that previously demonstrated capabilities have not been adversely affected by later development and/or error corrections.

SECTION 6 - ACCEPTANCE TESTING

The purpose of acceptance testing is to certify that the software system satisfies its requirements. Acceptance testing should not begin until the software has successfully completed system testing. Ideally, system documentation should also be ready. However, acceptance testing of a series of releases may be conducted in some situations in which an early initial operational capability is desired. Some important points to consider about acceptance testing include the following:

- Acceptance testing involves an interorganizational team.
- Testing time may have to be scheduled well in advance to ensure adequate computer resources.
- Acceptance tests should be run in an environment that is the same as or very similar to the planned operational environment.
- Test plans, data, and results must be archived to provide for regression testing.
- Discrepancies and changes must be formally recorded and reviewed.
- Acceptance testing must probe both functional and operational requirements.

6.1 PARTICIPANTS

Three groups of people participate in acceptance testing: analysts, developers, and operators. Each group has a specific role and responsibilities. However, in some cases, one organization may fill more than one role.

The analysts are responsible for conducting the tests. In addition, they must

- Generate the acceptance test plan
- Produce the simulated data for testing
- Evaluate the test results
- Identify any errors in the software
- Monitor the status of the testing

At least one member of the analysis team must be present at the acceptance test for it to be an official test. Other members of the analysis team may also be present during acceptance testing.

The developers are responsible for executing the software as directed by the analysts during testing. As a part of acceptance testing, the developers will

- Provide the analysts with all printed output of the tests
- Prepare (in advance) all required job control language (JCL) or command files necessary for executing the software
- Maintain configuration control over all the software
- Correct any errors identified in the software
- Generate new load modules after corrections have been made so that testing can continue

At least one member of the software development team must be present at the acceptance test for it to be an official test. Other members of the software development team also may be present at the acceptance tests to help in the execution of the software.

The operators ultimately will be responsible for executing the software on a regular basis. They may be present as observers. However, acceptance testing does not provide the required formal training in the use of the software. During the course of testing, the operators may make recommendations to the analysts and developers.

6.2 PLANNING

All acceptance tests executed are based on the acceptance test plan written by the analysts prior to the start of the acceptance testing phase. The tests in this plan should be traceable to the system requirements document. That is, every major requirement specified must be included in at least one test. Conversely, all capabilities implied in the test plan must originate in the requirements. The acceptance test plan should include the following information:

- An introduction that includes
  - Purpose of the tests
  - Type and level of testing
  - Testing schedule
- A test description for each test that includes
  - Purpose of the test
  - Detailed specifications of the input
  - Required environment (e.g., the use of graphics devices)
  - Operational procedure
  - Detailed specifications of the output
  - Pass or fail criteria
- A requirements traceability matrix showing the general requirements covered by each test

All tests in this plan must be passed for the software to be certified, unless an exception has been noted and formally approved.

6.3 EXECUTION

The acceptance test plan defines the procedures for executing the acceptance tests. It should be followed as closely as possible. The order of testing nominally will be the order of the tests in the acceptance test plan. This order may be changed during testing by a decision of the accepting organization. Figure 6-1 illustrates the acceptance testing process. Acceptance testing continues even if errors are found, unless the error itself prevents continuation. Procedures for data and source maintenance during the testing phase must comply with the relevant configuration management guidelines (Reference 2). The following subsections describe error correction procedures and the test environment.

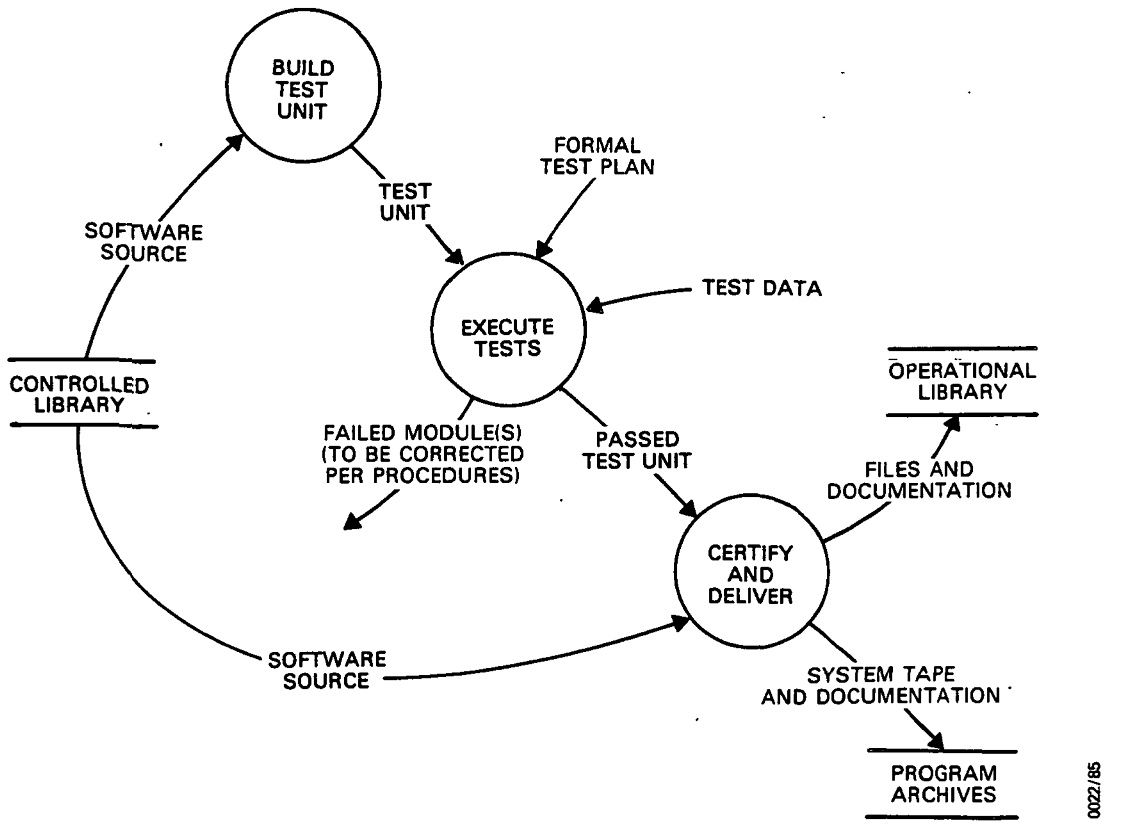


Figure 6-1. Acceptance Testing Process

6.3.1 ERROR CORRECTION

During acceptance testing, errors may be found in the software, in the data being used to test the software, or in the JCL or command procedures being used. Formal procedures must be followed for recording errors and making corrections to these items. Temporary corrections to the current JCL, control parameters, or command procedures may be made immediately by the developers (with the approval of the technical manager) so that testing can continue. In the case of a test data error, testing of that function must stop until the analyst regenerates the test data set. However, both of these types of errors must be reported on a problem report (see Appendix). Corrections to the controlled library must be documented with a change report (see Appendix).

Software errors are treated more formally. When a software error has been detected, a problem report must be initiated. One form must be initiated for each error found. Any given test may result in the detection of more than one error. At the end of a scheduled testing session, the analyst must complete a test results report for all tests attempted. The report should describe the errors found.

The software development technical manager assigns responsibility for making each correction to a member of the development team. The developer then identifies which routines need to be changed and copies those routines from the controlled library. When the changes have been made and unit tested, the controlled library can be updated. The problem reports addressed by these changes should be completed at this time. Change reports must also be submitted.

Generation and release of new load modules must be coordinated with the analysts. The new load module must be regression tested by the developers. This load module or task image is then released for further acceptance testing.

6.3.2 TESTING ENVIRONMENT

The acceptance testing environment should be as similar as possible to the planned operational environment. If the software is not being developed on the target machine, only the load module or source files (together with control files) are transferred to the operational computer. This may be done by magnetic tape or other route. Data set naming conventions, JCL classes, and system defaults may differ from the development computer to the target computer. The development team is responsible for ensuring that the system is ready for acceptance testing in every respect.

Consideration must be given to scheduling the resources necessary to perform the acceptance testing. A system that uses a lot of CPU time or memory may seriously impact other projects on the same computer. Strict timing tests may need to be run that cannot afford to be affected by other users. In these cases, reserved blocks of time may be required for acceptance testing. Time on peripherals, especially terminals and graphics devices, may also need to be scheduled.

Arrangements must be made between the developers and analysts for batch testing. The operational conditions specified in the test plan must still be satisfied. Test output must be retrieved and delivered to the analysts.

6.4 ANALYSIS

The results of each test must be studied by the analysis team to determine whether the test was successful. Successful completion of a test requires that

- All tested options ran without causing any abnormal terminations or other execution time errors

- All tested options worked correctly
- Numerical results were within the tolerance specified in the acceptance test plan
- All plots, tables, and control displays met all criteria specified for them, including the use of proper units and a clear description of all variables

Any part of the software that fails to meet these criteria must be reported as in error, and the problem must be corrected. Acceptance testing will likely generate both problem reports and test result reports. These are discussed below.

6.4.1 PROBLEM REPORTS

Problem reports must be <u>initiated</u> during the testing session for all errors detected. These forms provide a mechanism for ensuring that all errors discovered are corrected. The form is <u>completed</u> after the error correction has been implemented and tested. The Appendix includes a sample problem report.

The analysts will classify all errors according to seriousness. The specifics of the classification scheme will be decided upon prior to the start of testing. The most severe errors (as classified by the analyst) will be corrected first, and a decision will be made whether to correct very minor errors, make some cosmetic changes, or wait until the maintenance phase has begun. Acceptance testing will continue regardless of the errors found unless the error itself prevents further testing.

6.4.2 TEST RESULT REPORTS

In addition to problem reports, the outcome of each test attempted is presented in a set of test result reports:

- A preliminary report will be generated by the analyst at the end of the testing session and will be provided to the developers by the start of the next testing session. The purpose of this report is to provide an immediate assessment of the status of the system and testing progress.

- A detailed report will be generated by the analysts and provided to the developers within 3 days of the completion of the test session. These reports will describe any problems with the software but will not attempt to indicate where the error is occurring in the code.

- Periodic summary status reports will be generated by the analysts. For each test, these reports will indicate
  - Actual start and end dates of the testing
  - Date the output was received by the analysis team
  - Date on which the detailed report is due
  - Date on which the detailed report was delivered
  - Summary of the results

- Periodic error rate plots (Figure 6-2) may be generated by the developers. The cumulative number of errors found should level off as testing progresses. Consequently, the error rate will stabilize. Project A in Figure 6-2 provides a good example in which the error rate plot indicates that testing is complete. This provides a visual mechanism for determining when the system is ready for certification.

The Configuration Analysis Tool (CAT) can also be helpful in tracking testing progress (Reference 10). CAT provides an

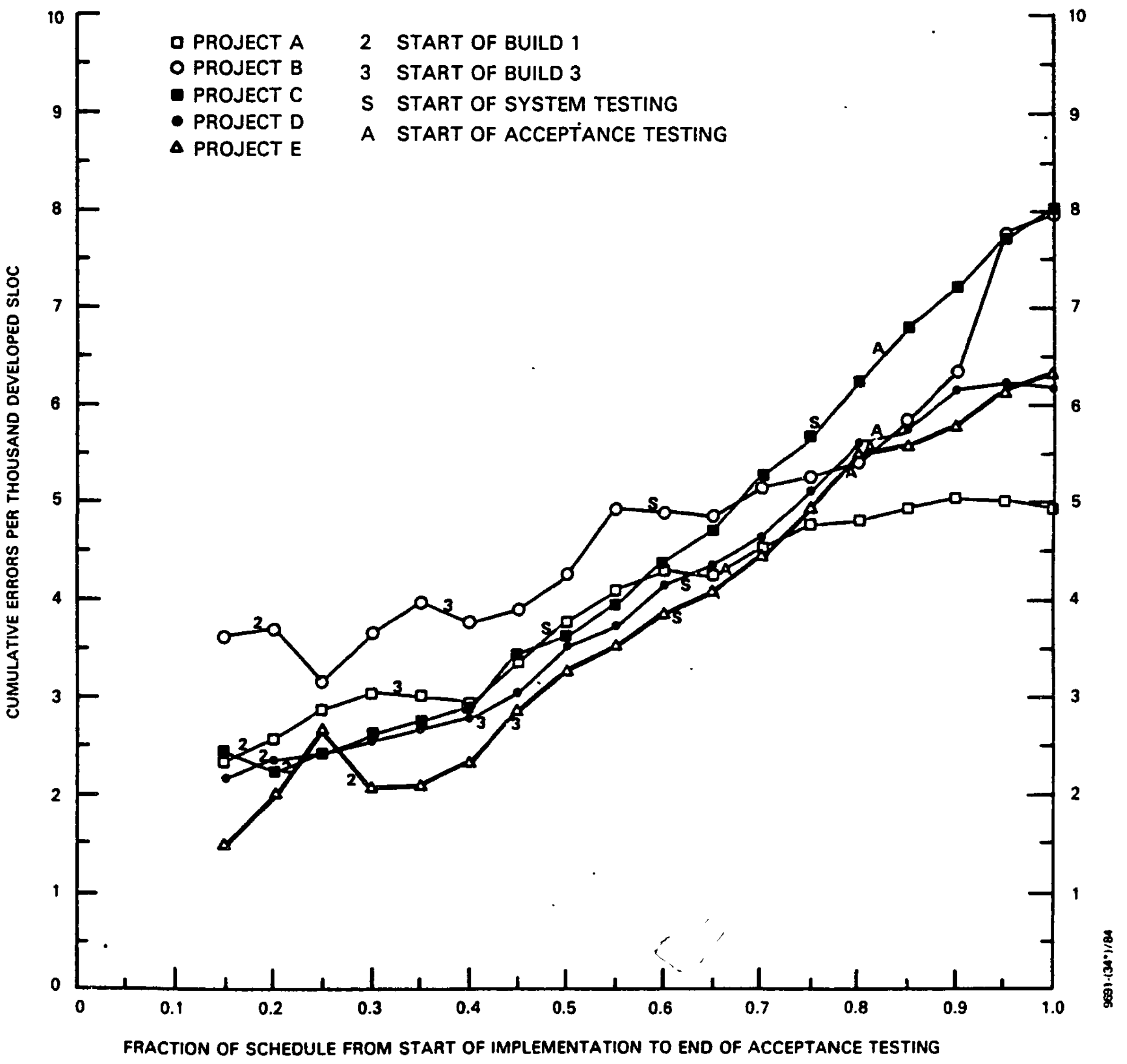


Figure 6-2. Error Rate Plot

automated process for managing problem and test reports (Reference 2).

6.5 TEST DATA

The timely availability of test data is just as important to acceptance testing as it is to system testing (see Section 5.4). Some important points to consider about acceptance test data include the following:

- The analysis team is responsible for generating acceptance test data.
- The development team is responsible for generating control (parameter and command) data sets.
- All acceptance test data and control data sets are kept under configuration control.

6.6 REGRESSION TESTING

A subset of (benchmark) tests should be designated from among the complete set of acceptance tests. This subset of tests will become the basis for subsequent regression testing of the system. This subset should exercise most of the basic capabilities of the system and should be reexecuted after any corrections have been made to the software. The purpose of these tests is to ensure that the corrections have not affected any other part of the system. These tests will also be used during the maintenance phase.

6.7 SOFTWARE CERTIFICATION AND DELIVERY

The criterion for certification of the software is that it passes all the identified acceptance tests. Once the software has passed all tests to the satisfaction of the analysis team and no outstanding discrepancies remain (or none whose delayed correction has not been approved), it is ready to be delivered. At this point, the system tape is generated by the software development team. This tape will contain, in a standard format (Reference 11), the following:

- All the source code and other files needed to create the load module
- Load module or executable task image
- JCL, control files, and any other files necessary for system execution
- Copies of all the acceptance test files (data, JCL, NAMELISTs, etc.)
- Copies of all the acceptance test results

All the information on this tape should be exact copies of the items used in the acceptance tests. When the information is unloaded from the tape, the maintenance team should be able to recreate fully the system as well as the acceptance tests. System documentation should be delivered at the same time as the software. Copies of all system tapes and documentation reside in the customer program archives.

The files from which the system tape was generated may also be made available to the maintenance team. When the development computer and target computer are the same or in communication with each other, it is easier to build the operational system from these files than from the system tape. However, this file transfer does not substitute for the official delivery of a system tape unless specified as such by the customer.

SECTION 7 - MAINTENANCE TESTING

Testing during maintenance includes many activities. The operational use of a system constitutes testing, because this is how most errors are found during the maintenance phase. Operation also suggests ideas for enhancements. Corrections, enhancements, and new releases require that regression testing as well as development testing be performed.

Preparations for maintenance testing must be made during the earlier phases of system and acceptance testing. Archiving the system is the essential step in preparing for maintenance. Archived materials include the following:

- System and acceptance test plans
- Test data and results associated with executing the system against that data (at least the test data and results from the benchmark regression tests must be saved), usually on the system tape
- Software source and supporting materials on the system tape

The system tape (or other delivered materials) should be used to create the original system for operations and maintenance. The benchmark regression tests must be run by the maintenance team as soon as the system is built, to confirm that it is in good working order.

7.1 REGRESSION TESTING

Whenever maintainers make a change or enhancement to the system, regression testing must be performed. The purpose of this testing is to ensure that a change in one part of the system does not propagate errors into previously functioning parts of the system. Regression testing includes executing the previously defined set of benchmark tests and studying the results. New tests must be created for any enhancements added to the system.

Usually, complex test cases are executed first. These test the overall functionality of the system. If these are unsuccessful, then simpler, localized cases are run to isolate the problem in the system. The error can then be identified and corrected, and the system retested.

As part of the testing process, the test results are evaluated for correctness. If the tests being run are the standard set of baseline tests, the results can be compared to the results that were archived from previous successful execution of the tests. If the tests are tests created for a new function or error correction, the results should be checked carefully by hand. When that test is correct, it should become a part of the benchmark set of tests.

7.2 OPERATIONAL TESTING

The operational environment cannot usually be duplicated exactly during system and acceptance testing. Operation uncovers hardware/software incompatibility and operator errors that may not show up during development testing. Incompatibilities also arise when changes are made to the hardware configuration after the software is developed, or when the operating system is upgraded or modified. Developers try to anticipate the types of errors that operators will make so that they can be managed; however, they are generally not completely successful.

Operational testing can be performed by running the new system in parallel with the old or by confining its use to noncritical applications with careful oversight. These conditions continue until a sufficient level of confidence in the system is achieved. New releases of a system may also undergo operational testing. Thus, testing continues throughout the life of a software system. Reference 12 provides additional information about managing operational software in the flight dynamics environment.

SECTION 8 - SUMMARY

Testing provides the last chance for developers to get the software right. It cannot be conducted in a haphazard manner. Some important points to remember about testing include the following:

- To be effective, tests must be planned.
- Many testing approaches must be used to ensure complete testing.
- Configuration control of software and test data must be maintained at all times.
- Test results must be fully documented. Errors must be reported formally.
- It is better to prevent errors than to find them during testing.

Testing cannot substitute for a careful software implementation; however, together with quality assurance and configuration management, effective testing ensures that the software product ultimately delivered is reliable and satisfies the customer's requirements.

APPENDIX - SAMPLE TEST AND VERIFICATION REPORTS

This Appendix provides samples of some frequently used test and verification reports. These items are samples only. Local guidelines prescribe the specific reports to be used. The reports included are as follows:

- Code reading checklist (Figure A-1)
- Test plan element (Figure A-2)
- Change report (Figure A-3)
- Problem report (Figure A-4)

Checklists may be completed during code reading. If used, they are kept in the programmers' unit developement folder. Individual test plans are compiled into the system test and acceptance test documents. Change reports are submitted when a controlled item is changed. Problem reports are initiated when an error is found, then completed when the correction is implemented. Change and problem reports are kept by the configuration manager.

CODE READING REPORT

CONTENT	ARE ALL CONSTANTS DEFINED?	☐
	ARE ALL UNIQUE VALUES EXPLICITLY TESTED FOR INPUT PARAMETERS?	☐
	ARE VALUES STORED AFTER THEY ARE CALCULATED?	☐
	IS A PATH DEFINED FOR EVERY POSSIBLE OUTCOME OF A LOGICAL DECISION?	☐
	ARE ALL INPUT DEFAULTS EXPLICITLY TESTED?	☐
	IF A KEYWORD HAS MANY UNIQUE VALUES, ARE THEY ALL CHECKED?	☐
	IS THE SPECIFIED PRECISION SUFFICIENT FOR THE REQUIRED ACCURACY?	☐
	ARE FLAGGED/MISSING DATA VALUES CORRECTLY EXCLUDED FROM COMPUTATIONS?	☐
	ARE DISPLAY FORMATS LARGE ENOUGH?	☐
	ARE DATA SETS PROPERLY OPENED AND CLOSED?	☐
	ARE LEADING AND TRAILING RECORDS RECOGNIZED AND HANDLED APPROPRIATELY?	☐
	ARE UNIT NUMBERS UNIQUE?	☐
	ARE DOUBLE-PRECISION CONSTANTS USED IN DOUBLE-PRECISION EXPRESSIONS?	☐
	ARE CORRECT UNITS OR THE APPROPRIATE CONVERSION USED (e.g., DEGREES, RADIANS)?	☐
	IS THE CORRECT SIGN (POSITIVE OR NEGATIVE) USED, ESPECIALLY IN BIAS ADJUSTMENTS?	☐
	ARE ALL COUNTERS PROPERLY INITIALIZED (0 OR 1)?	☐
	ARE ABSOLUTE AND SYMBOLIC (PARAMETER) VALUES USED APPROPRIATELY?	☐
	ON COMPARISON OF TWO BYTES, ARE ALL BITS COMPARED, IF NECESSARY?	☐
FORMAT	ARE THE VARIABLE NAMES INFORMATIVE?	☐
	DOES THE PROLOG DEFINE ALL VARIABLES USED?	☐
	IS THE CODE INDENTED TO SHOW ITS STRUCTURE?	☐
	IS THE CODE ADEQUATELY COMMENTED?	☐
	DOES THE PROLOG DEFINE OPERATIONAL CONSTRAINTS AND ASSUMPTIONS?	☐
	ARE CONTINUATION LINES CLEARLY INDICATED?	☐
	DO STATEMENT LABELS OCCUR IN AN OBVIOUS SEQUENCE?	☐

DATE ______________ NAME ______________ MODULE ______________

0022-(109)/85

Figure A-1. Code Reading Checklist

Subsystem: FADS Date: 9/10/83
Test Number: R2-01 Load Module: ADS0909A

Purpose: Verify the capability to compute an epochal attitude for a batch using the differential corrector algorithm. Verify the use of convergence criteria and the calculation and display of residuals and statistics.

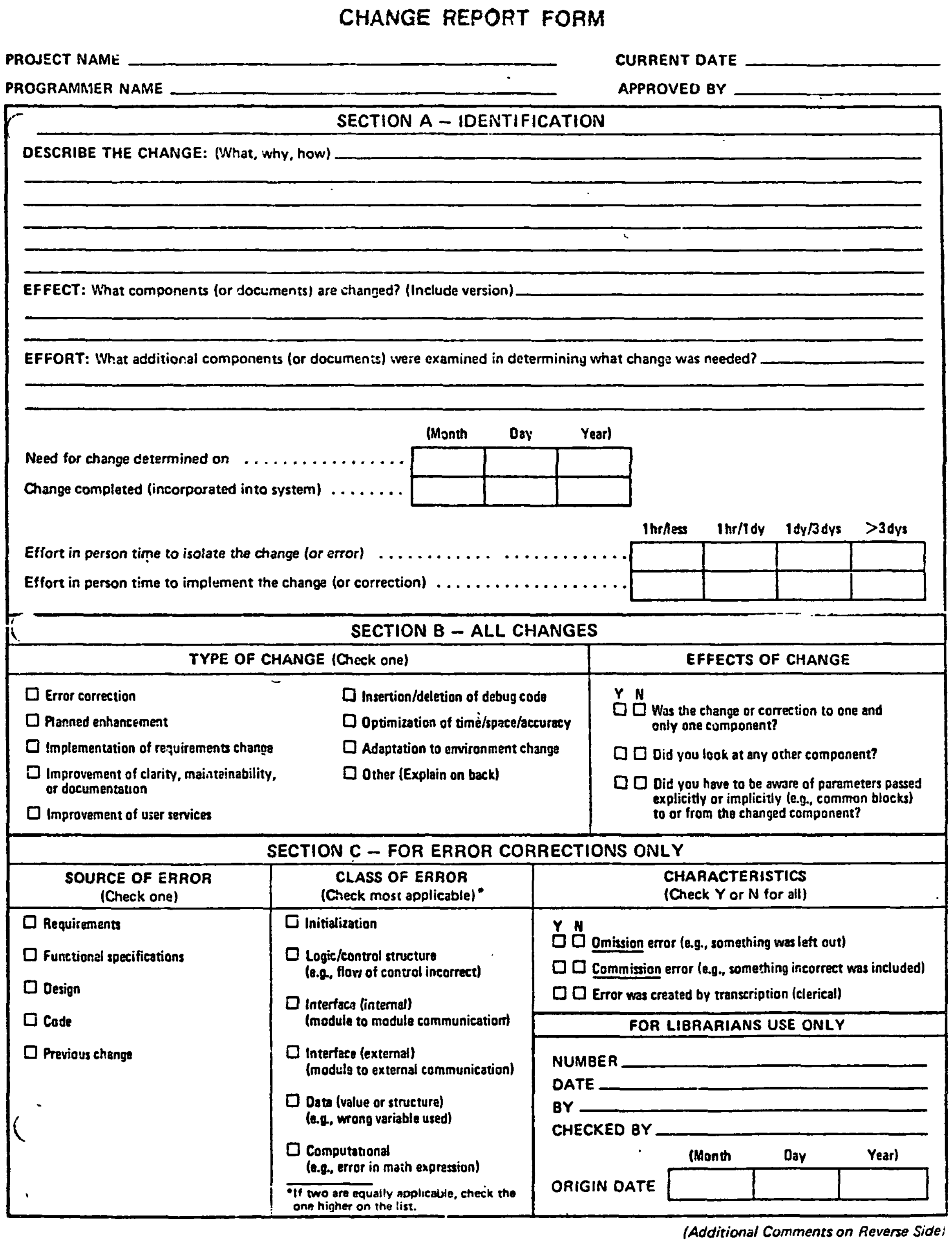
Input: Use data with a constant attitude of roll, pitch, and yaw equal to zero, and no biases, no noise. Use a short span of data (approximately 10 minutes) with gyro scale factors set to 2 to correct the gyro data.

Environment: This test must be run using an IBM 3250 graphics device. A processed engineering data set is required.

Procedure: Process data with no residual editing using a zero a priori weight matrix and equal (1.0) data weights. Verify offset of statistics parameters in displays through comparison of debug output. Test all five convergence criteria. Use an initial a priori estimate of off null and do not solve for biases.

Expected Results: Reduction of residuals and related statistics for each iteration until the solution converges to an epochal null attitude.

Figure A-2. Test Plan Element



CHANGE REPORT FORM

PROJECT NAME ______________________ CURRENT DATE ____________

PROGRAMMER NAME ______________________ APPROVED BY ____________

SECTION A – IDENTIFICATION

DESCRIBE THE CHANGE: (What, why, how) ______________________

EFFECT: What components (or documents) are changed? (Include version) ______________________

EFFORT: What additional components (or documents) were examined in determining what change was needed? ______________________

	(Month	Day	Year)
Need for change determined on			
Change completed (incorporated into system)			

	1hr/less	1hr/1dy	1dy/3dys	>3dys
Effort in person time to isolate the change (or error)				
Effort in person time to implement the change (or correction)				

SECTION B – ALL CHANGES

TYPE OF CHANGE (Check one)

- ☐ Error correction
- ☐ Planned enhancement
- ☐ Implementation of requirements change
- ☐ Improvement of clarity, maintainability, or documentation
- ☐ Improvement of user services
- ☐ Insertion/deletion of debug code
- ☐ Optimization of time/space/accuracy
- ☐ Adaptation to environment change
- ☐ Other (Explain on back)

EFFECTS OF CHANGE

Y N

- ☐ ☐ Was the change or correction to one and only one component?
- ☐ ☐ Did you look at any other component?
- ☐ ☐ Did you have to be aware of parameters passed explicitly or implicitly (e.g., common blocks) to or from the changed component?

SECTION C – FOR ERROR CORRECTIONS ONLY

SOURCE OF ERROR (Check one)

- ☐ Requirements
- ☐ Functional specifications
- ☐ Design
- ☐ Code
- ☐ Previous change

CLASS OF ERROR (Check most applicable)*

- ☐ Initialization
- ☐ Logic/control structure (e.g., flow of control incorrect)
- ☐ Interface (internal) (module to module communication)
- ☐ Interface (external) (module to external communication)
- ☐ Data (value or structure) (e.g., wrong variable used)
- ☐ Computational (e.g., error in math expression)

*If two are equally applicable, check the one higher on the list.

CHARACTERISTICS (Check Y or N for all)

Y N

- ☐ ☐ Omission error (e.g., something was left out)
- ☐ ☐ Commission error (e.g., something incorrect was included)
- ☐ ☐ Error was created by transcription (clerical)

FOR LIBRARIANS USE ONLY

NUMBER ____________

DATE ____________

BY ____________

CHECKED BY ____________

	(Month	Day	Year)
ORIGIN DATE			

(Additional Comments on Reverse Side)

Figure A-3. Change Report Form

Current Date ________

________ Attitude System Maintenance Report Number ________
(system name)

A. Project Name ________ Need for change determined on (Mo., Day, Yr.) ________

Describe problem ________

____

____

What components/subroutines/modules are suspected? ________

Proposed method for testing modifications: ________

B. Change (NON-ERROR) (fill out this section if change is NOT an error correction). This change is being made because of a change in: (Check all that apply)

____ requirements/specifications	____ software environment
____ new information/data	____ optimization
____ design	____ other (specify): ________
____ hardware environment	

C. ERROR ONLY (fill out this section if change IS an error correction). The following activities were used in error detection or isolation: (Check all that apply)

____ normal use	____ trace/dump
____ test runs	____ cross reference/attitude list
____ code reading	____ system error messages
____ reading documentation	____ project specific error messages
____ other (specify): ________	

D. Please give any information that may be helpful in categorizing and understanding the change on the reverse side of this form.

Person filling out this form (signature) ________

Management approved (signature) ________ Date ________

Change started on date (month, day, year) ________

Time spent on this change (includes isolation, implementation and testing):

____ less than 1 day ____ 1 day to a week ____ more than a week

Which of the following best describes the error:

____ requirements/specification error	____ code error
____ design error	____ clerical error
____ other: Describe ________	

Was this error related to a previous maintenance change ____ yes ____ no ____ can't tell

What components/subroutines/modules were changed? ________

New load module name ________ Data set ________

New nonresident table name ________ Data set ________

Ready for parallel tests (librarian signature) ________ Date ________

Parallel tests complete (signature) ________ Date ________

Management approved for operational use (signature) ________ Date ________

Operational (librarian signature) ________ Date ________

Figure A-4. Problem Report

REFERENCES

1. Software Engineering Laboratory, SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. T. Page, S. Eslinger, et al., April 1983

2. Computer Sciences Corporation, Product Assurance for Flight Dynamics Software Development, Q. L. Jordan, April 1986

3. G. J. Myers, "Test-Case Design," The Art of Software Testing. New York: John Wiley & Sons, 1979

4. C. K. Cho, An Introduction to Software Quality Control. New York: John Wiley & Sons, 1980

5. Software Engineering Laboratory, SEL-78-202, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide, W. J. Decker and W. A. Taylor, April 1985

6. General Research Corporation, RM-2419, RXVP80, The Verification and Validation System for FORTRAN, User's Manual, 1982

7. Computer Sciences Corporation, Programmer's Handbook for Flight Dynamics Software Development, R. J. Wood, February 1986

8. M. S. Deutsch, "Verification and Validation," Software Engineering. New Jersey: Prentice Hall, 1979

9. Computer Sciences Corporation, Digital Systems Development Methodology, S. Steppel, T. L. Clark, P. C. Belford, et al., March 1984

10. Software Engineering Laboratory, SEL-80-104, Configuration Analysis Tool (CAT) System Description and User's Guide, W. J. Decker and W. A. Taylor, December 1982

11. Software Engineering Laboratory, SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

12. Computer Sciences Corporation, CSC/TM-85/6716, Operational Software Modification Procedures, D. Kistler, June 1985

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-001, The Software Engineering Laboratory, V. R. Basili, M. V. Zelkowitz, F. E. McGarry, et al., May 1977

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-003, Structured FORTRAN Preprocessor (SFORT), B. Chu and D. S. Wilson, September 1977

SEL-77-004, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-001, FORTRAN Static Source Code Analyzer (SAP) Design and Module Descriptions, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

SEL-78-003, Evaluation of Draper NAVPAK Software Design, K. Tasaki and F. E. McGarry, June 1978

SEL-78-004, Structured FORTRAN Preprocessor (SFORT) PDP-11/70 User's Guide, D. S. Wilson and B. Chu, September 1978

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-78-202, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 2), W. J. Decker and W. A. Taylor, April 1985

SEL-79-001, SIMPL-D Data Base Reference Manual, M. V. Zelkowitz, July 1979

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-001, Functional Requirements/Specifications for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, A. L. Green, and C. E. Goorevich, February 1980

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-80-104, Configuration Analysis Tool (CAT) System Description and User's Guide (Revision 1), W. Decker and W. Taylor, December 1982

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve As a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-106, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, May 1985

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-203, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo, June 1984

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-003, Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description, P. Lo, September 1982

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-306, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, Q. L. Jordan, and F. E. McGarry, November 1985

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-83-104, Software Engineering Laboratory (SEL) Data Base Retrieval System (DARES) User's Guide, T. A. Babst, W. J. Decker, P. Lo, and W. Miller, August 1984

SEL-83-105, Software Engineering Laboratory (SEL) Data Base Retrieval System (DARES) System Description, P. Lo, W. J. Decker, and W. Miller, August 1984

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

SEL-84-002, Configuration Management and Control: Policies and Procedures, Q. L. Jordan and E. Edwards, December 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, Proceedings From the Ninth Annual Software Engineering Workshop, November 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team, R. Murphy and M. Stark, October 1985

SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., May 1985

SEL-85-005, Software Verification and Testing, D. N. Card, C. Antle, and E. Edwards, December 1985

Agresti, W. W., Definition of Specification Measures for the Software Engineering Laboratory, Computer Sciences Corporation, CSC/TM-84/6085, June 1984

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[3]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

Basili, V. R., "SEL Relationships for Programming Measurement and Estimation," University of Maryland, Technical Memorandum, October 1979

[3]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: Computer Societies Press, 1980 (also designated SEL-80-008)

[1]Basili, V. R., "Quantitative Evaluation of Software Methodology," Proceedings of the First Pan-Pacific Computer Conference, September 1985

[3]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?", Journal of Systems and Software, February 1981, vol. 2, no. 1

[3]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[1]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," Proceedings of the International Computer Software and Applications Conference, October 1985

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

[3]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981

[1]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," Proceedings of the IEEE/MITRE Expert Systems in Government Symposium, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity and Cost, October 1979

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

[1]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering, August 1985

Basili, V. R., and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland Technical Report, TR-1501, May 1985

[2]Basili, V.R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

[1]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984

[3]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[3]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[3]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: Computer Societies Press, 1978

[1]Card, D. N., "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

[1]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering, August 1985

[3]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: Computer Societies Press, 1983

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

[1]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," Proceedings of the Hawaiian International Conference on System Sciences, January 1985

[1]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

[1]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering, August 1985

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[1]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

[3]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: Computer Societies Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

[2]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.