QW 88 QUALITY WEEK

May 11 - 13, 1988

San Francisco, California

Software Research, Inc.

Robert Poston

Programming Environments, Inc.
Tinton Falls, New Jersey

Topic: ***Automatic Test Case Generation from Requirements***

Robert Poston became founder and president of PEI in 1981, after 20 years in the electronics and computer industries. He has been honored by U.S. Marine Corps, Honeywell Corporation, ITT, and General Electric Company for outstanding service. Mr. Poston holds a B.S.E.E. from California Poltytech and has 30 publications and eight international seminars to his credit. An honored member of IEEE, he serves as Chairman of Task Force for Professional Computing Tools, Program Committee Coordinator of the Computer Standards Conference, and has been a member of the Editorial Board since 1984. Mr. Poston also served as Technical Director of IEEE Seminars from 1981-1984.

John C. Kelly

Jet Propulsion Laboratory
Pasadena, California

Topic: ***Formal Inspection Methods***

John Kelly is a technical staff member in the Software Product Assurance Section at NASA's Jet Propulsion Laboratory in Pasadena, California. His current task is the technology transfer and coordination of formal software inspections at JPL.

Dr. Kelly worked in the Florida Dept. of Transportation in 1972-73 as a statistician. During graduate school he worked with the center for educational technology at FSU, developing software for the CDC Plato System. Dr. Kelly has served on the faculties of Albany Junior College as Assistant Professor of Mathematics, and Furman University in the Dept. of Computer Science. Dr. Kelly received his B.S., M.S., and Ph.D. degrees in Mathematics and Mathematics Education from Florida State Unversity.

Formal Inspections

Quality Week
May 12, 1988
San Francisco, CA

John C. Kelly, Ph.D.

JPL

Software Product Assurance
Section 515, MS 301-476
Jet Propulsion Laboratory
Pasadena, CA 91109
ph. (818)-354-4495

JPL

Contents

- The Problem
- Formal Inspections and the Software Lifecycle
- What are Formal Inspections?
- Data Reported from Formal Inspections
- Benefits

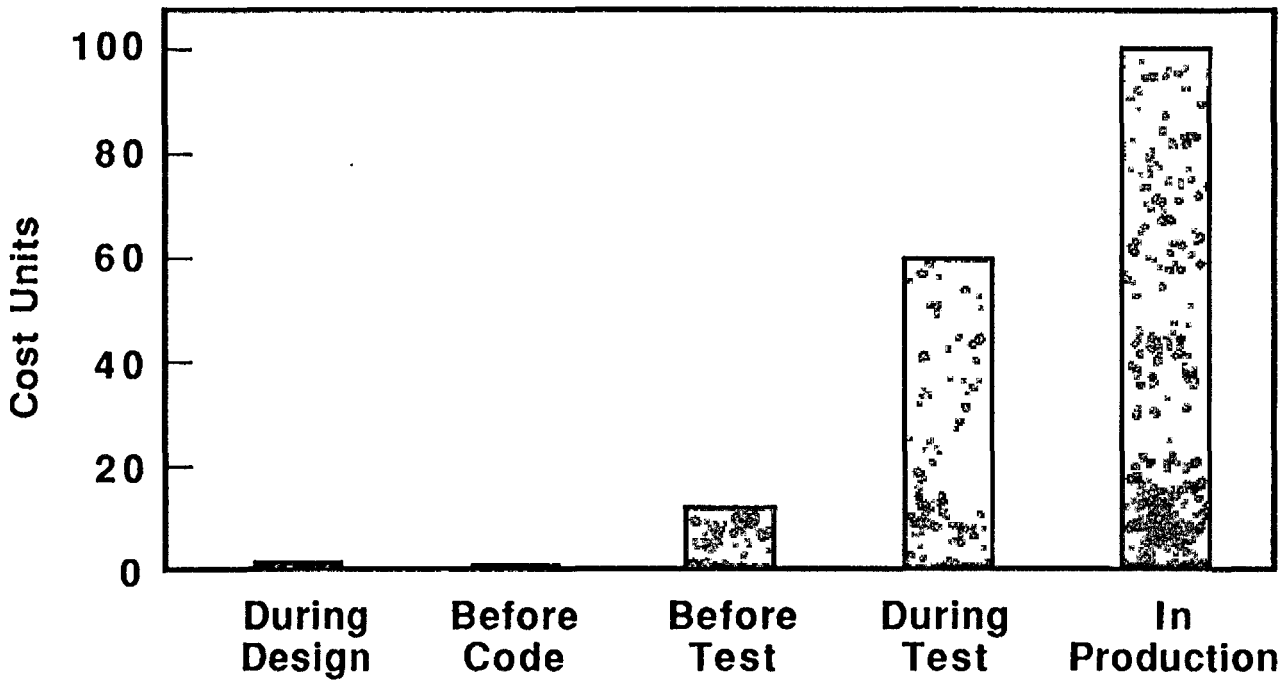
JPL

The Problem

"The cost of reworking errors in programs becomes higher the later they are reworked in the process, so every attempt should be made to find and fix errors as early in the process as possible."

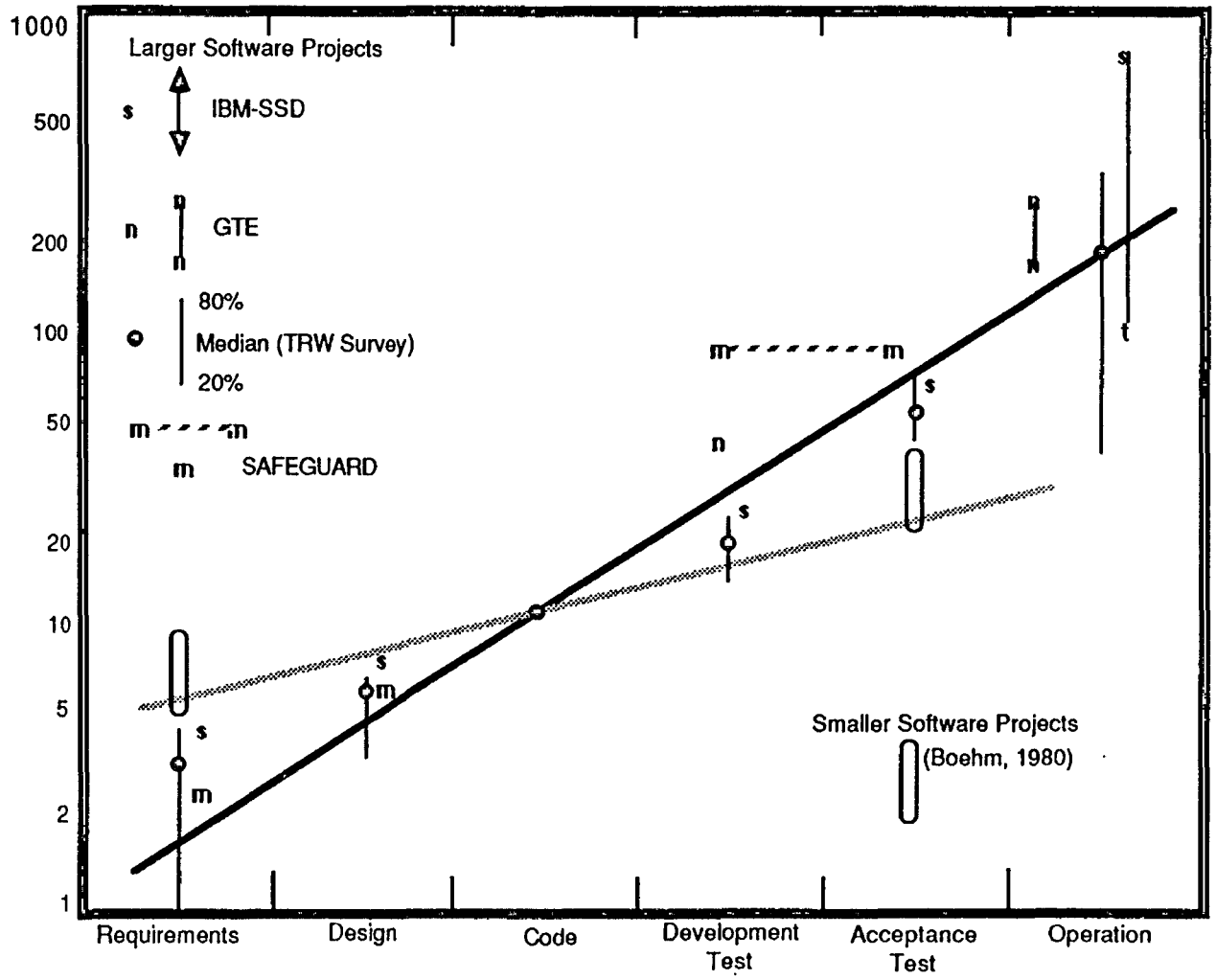
--Michael Fagan, 1976

Relative Cost to Find and Fix Defects When They Are Caught at Different Development Stages



Source: Fagan & Remus, reported in Gilb, 1987

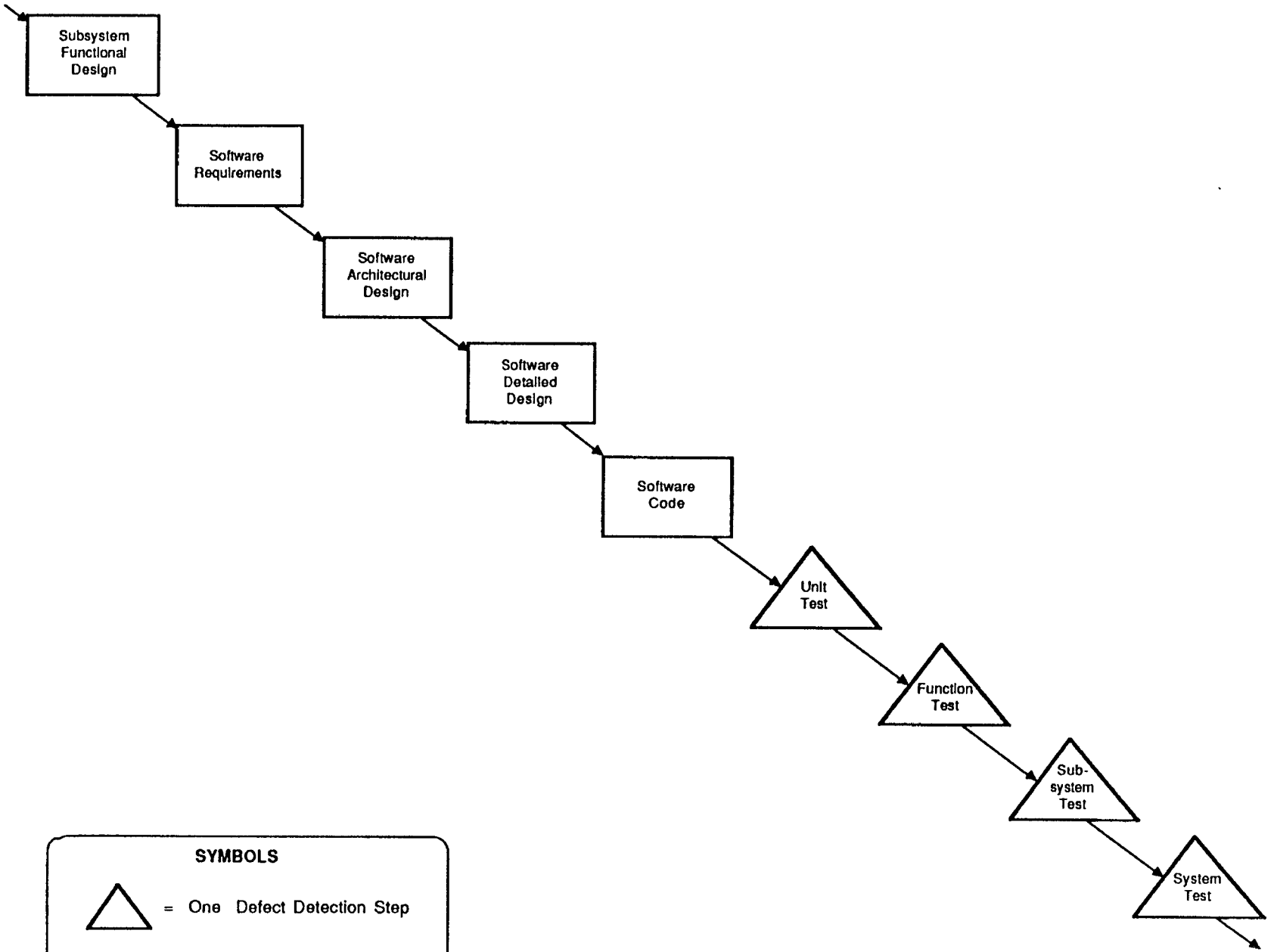
Relative Cost to Locate and Fix Defects at Each Phase of the Life Cycle



Source: Boehm, 1981

Solid Line: Combined Linear Regression Line for Large Projects
Dotted Line: Small Projects

The Software Development Cycle



SYMBOLS

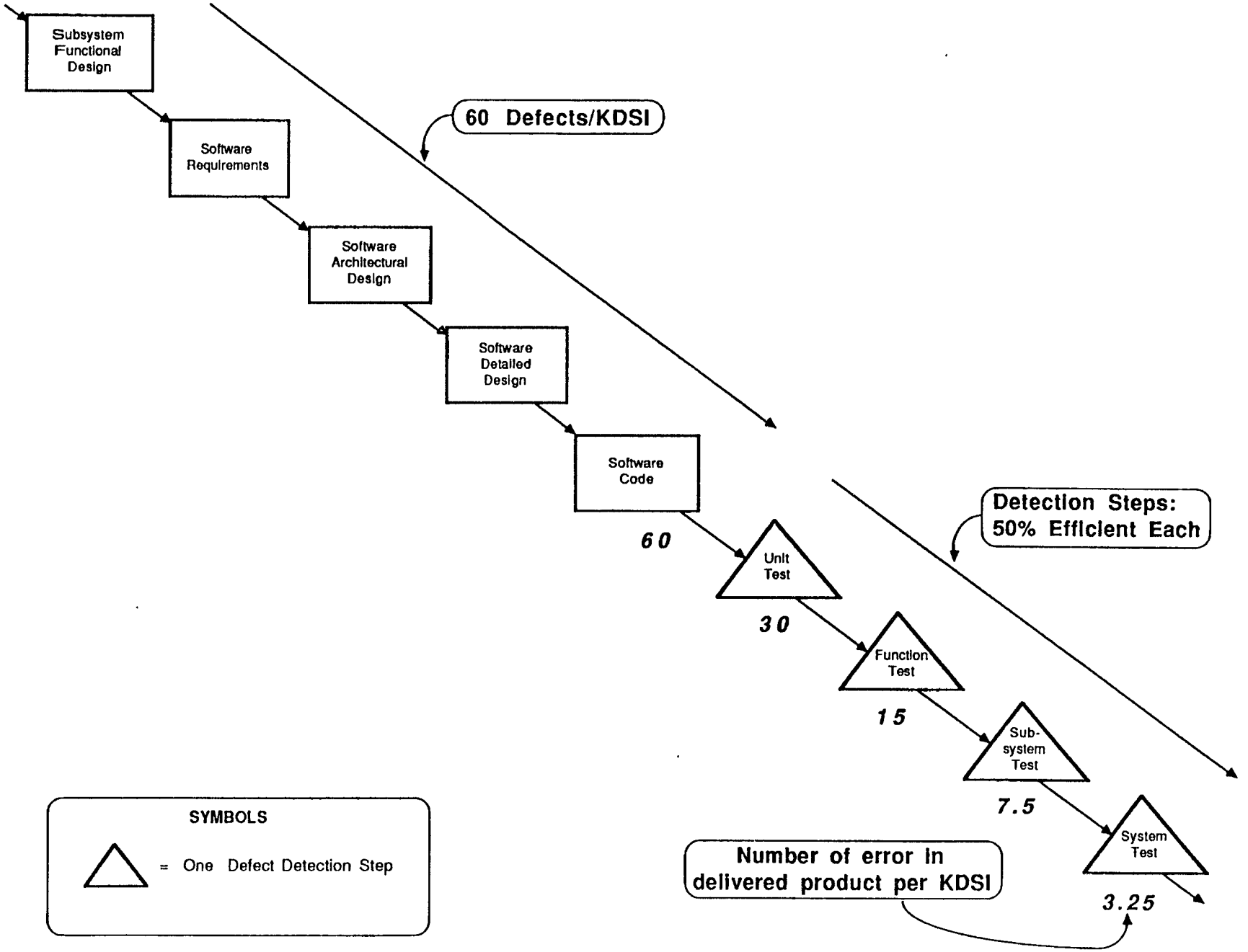
△ = One Defect Detection Step

JPL

Typical Example of Defect Injection and Removal

- Assume 60 Defects Escape pre-Test phases for Every K Lines Written
- Assume Test Steps Are Each 50% Efficient

The Software Quality With Test Only

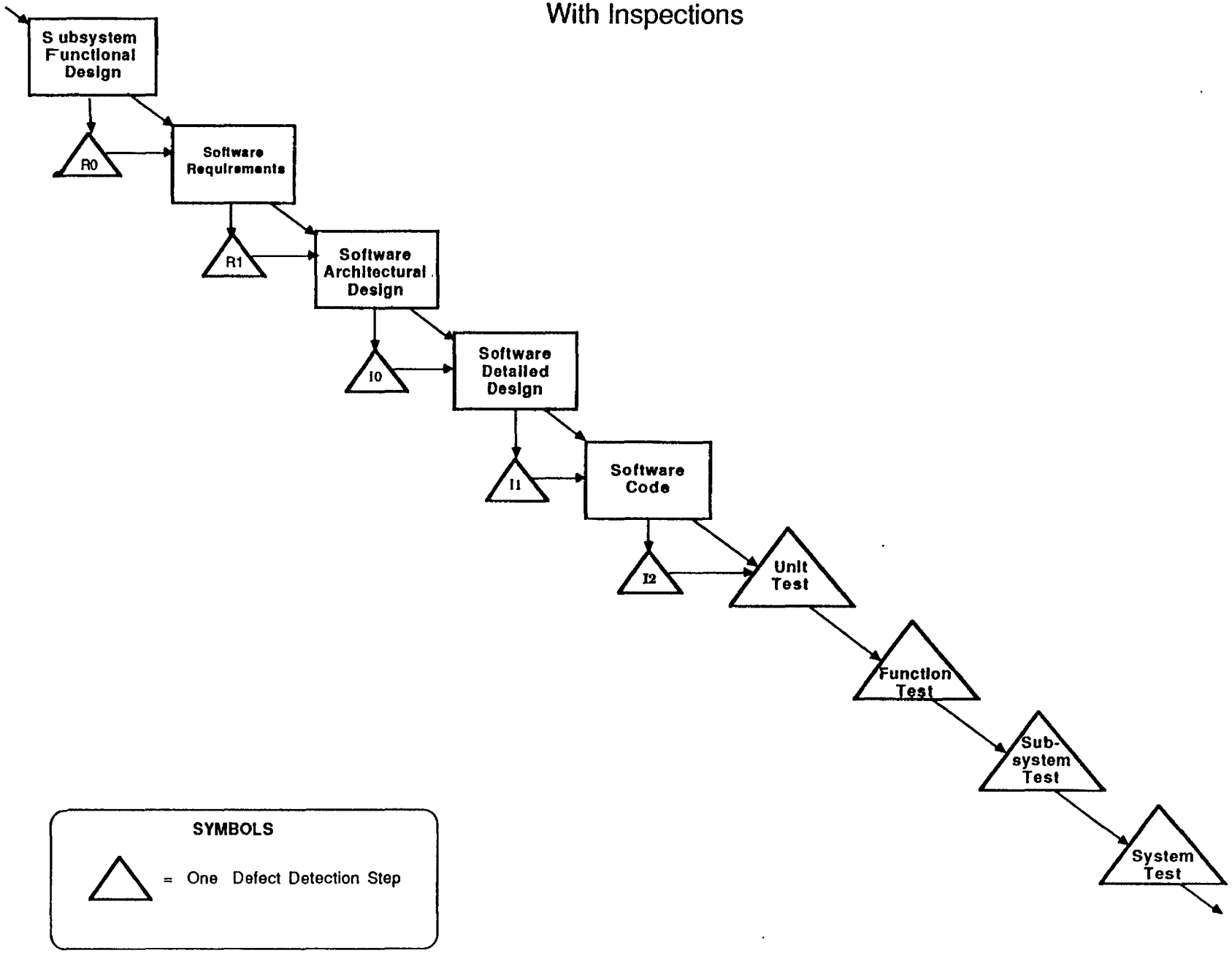


JPL

The Same Example With Formal Inspections

- **Insert inspections into the pre-test phases**
- **The strategy is to find and fix defects when and where they are injected**
- **Now have 9 detection steps instead of 4**

The Software Development Cycle
With Inspections



SYMBOLS

△ = One Defect Detection Step

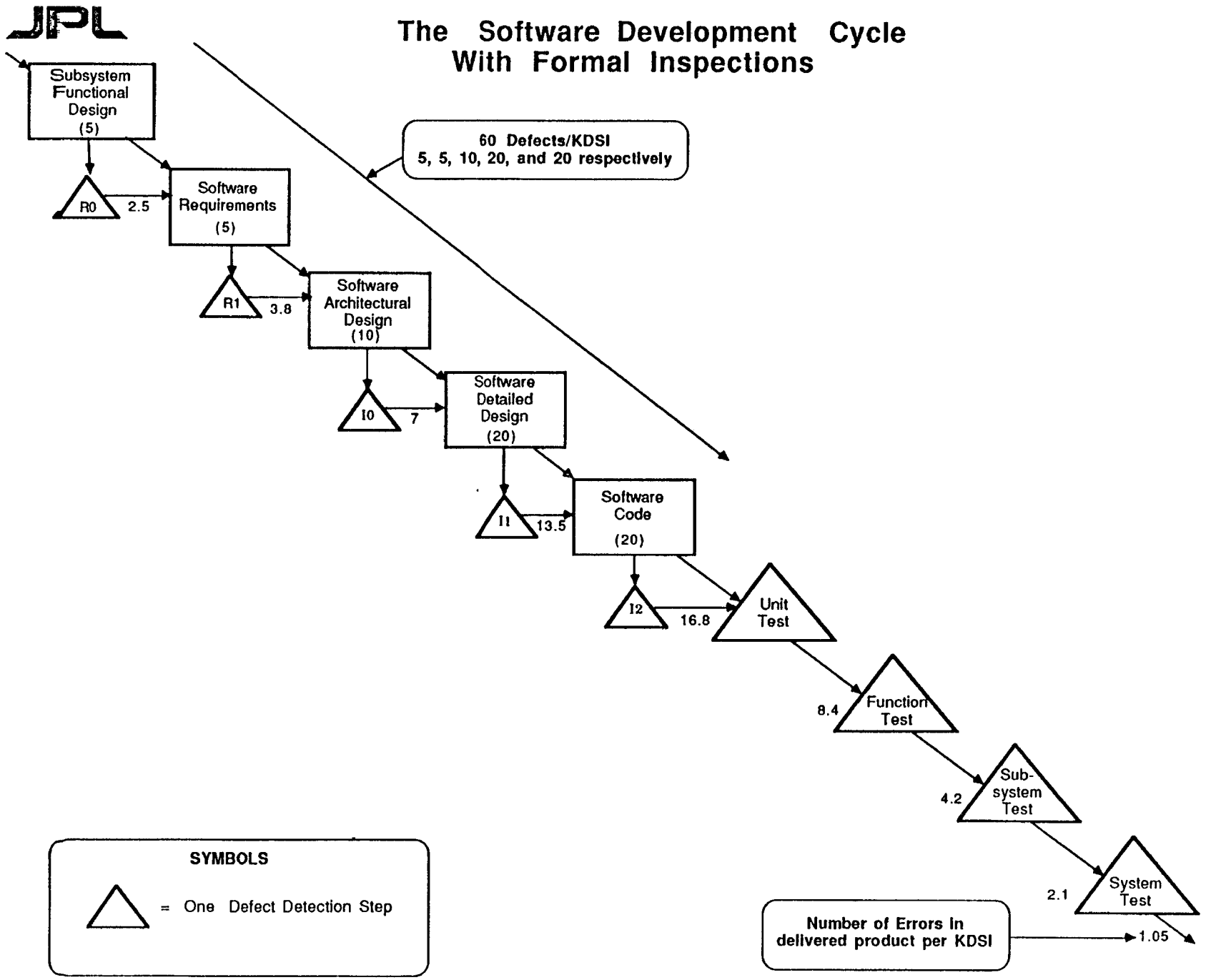
JPL

Must make two assumptions:

1) **How escaping defects are spread across phases**
   (use 5, 5, 10, 20, 20)

2) **Inspection efficiency**
   (use 50% - conservative)

The Software Development Cycle With Formal Inspections



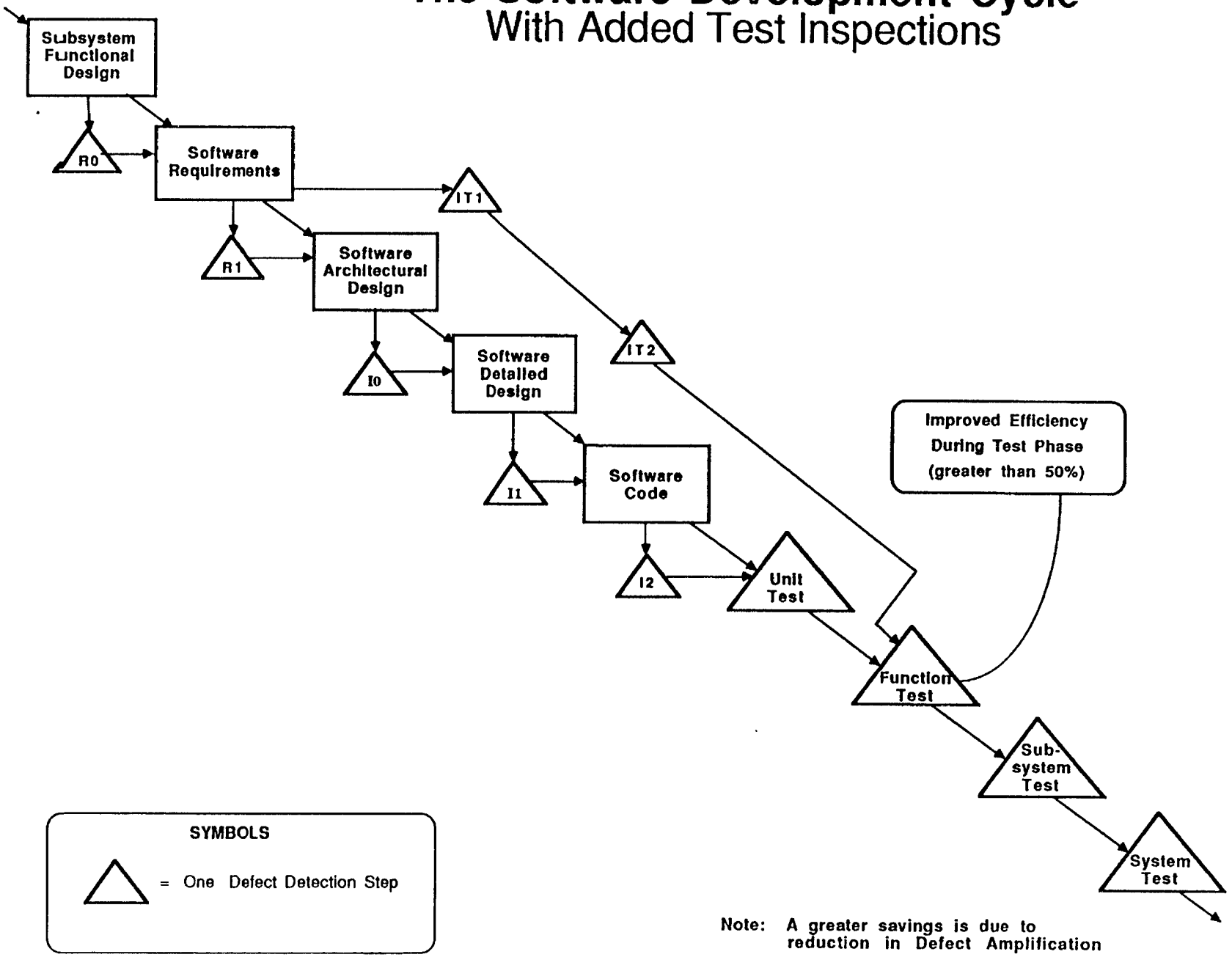
JPL

Test Related Inspections

- Test Inspections (IT1 and IT2) help produce high quality test cases

- Result is to increase efficiency of test phases

The Software Development Cycle
With Added Test Inspections



Note: A greater savings is due to reduction in Defect Amplification

JPL

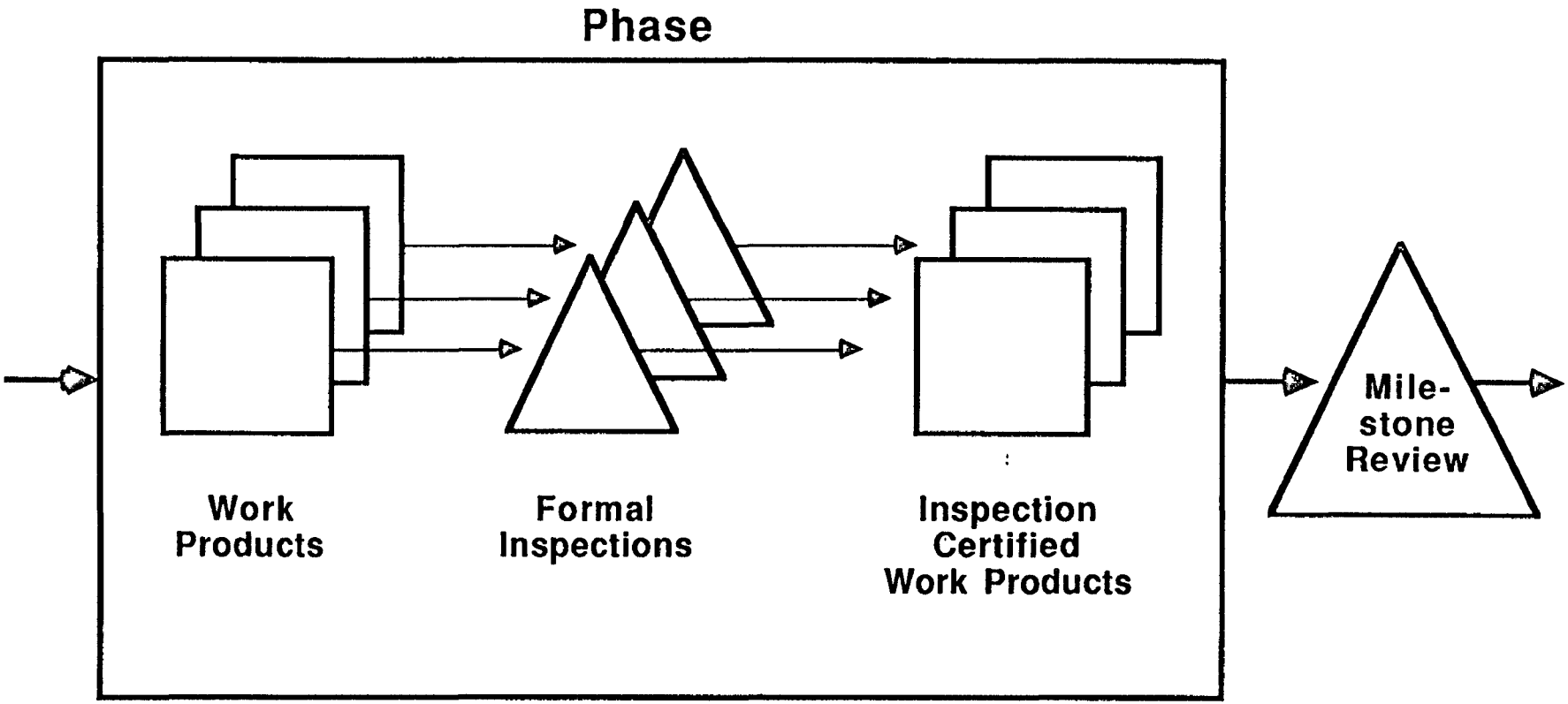
DIFFERENCES

		FORMAL INSPECTIONS	MILESTONE REVIEW*
1.	Occur	Inside phases	Between phase
2.	Configuration management	Internal	Baselined
3.	Size of review material	Small	Large
4.	Attendees	Small group (invitation only, assigned roles, no managers)	Large group (open)
5.	Purpose	Find and fix defects early	Product conformance with requirements & stds. Place products under configuration mngmt. Validate conformance with schedule and resource constraints.

*For more information see "Guidelines for Planning and Conducting Formal Reviews."

Source: JPL D-363, Rev. D, March 1987

JPL Formal Inspections Are "In-Process" Reviews



Source: J. Kelly, 1987

JPL

What are Formal Inspections?

(contents)

- Objective
- Formal Inspections vs. Walkthroughs
- Description of Formal Inspections
  - Phases
  - Participant Roles
  - Types of Inspections
  - Basic Rules

JPL

Primary Objective of Formal Inspections

Remove Defects As Early As Possible in the Development Process

Formal inspections achieve this objective by:

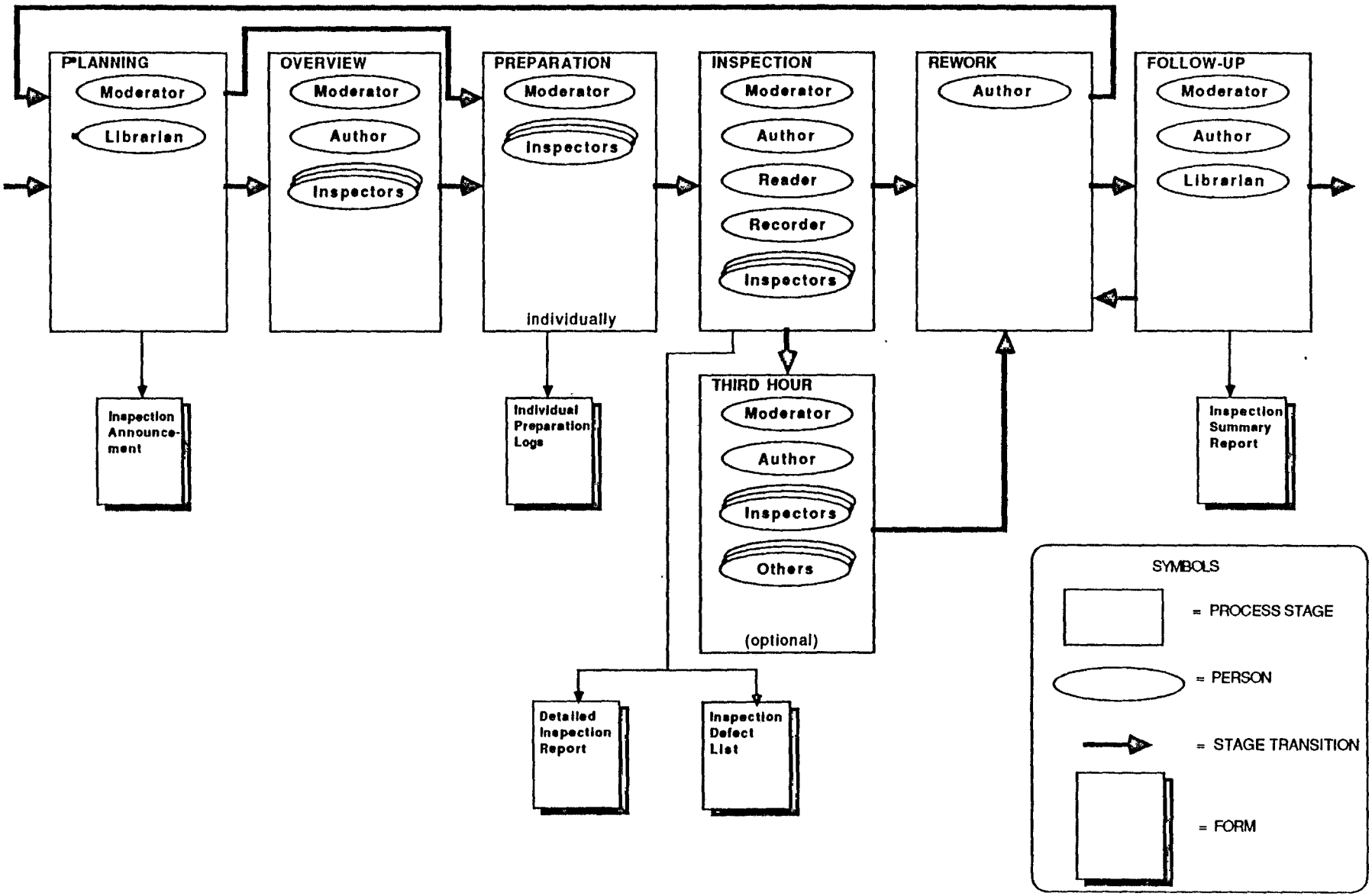
- Identifying potential defects during individual preparation
- Verifying that identified items are actual defects
- Recording the existence of defects
- Providing the author with a list of defects to fix
- Ensuring that fixes are performed and correct

Differences between Formal Inspections and Walk-throughs

Properties	Inspection	Walk-through
1. Formal moderator training	Yes	No
2. Definite participant roles	Yes	No
3. Who "drives" the inspection	Moderator	Author
4. Use "How to find errors" checklists	Yes	No
5. Use distribution of defects	Yes	No
6. Follow-up to reduce bad fixes	Yes	No
7. Less future errors because of detailed error feedback to individual programmer	Yes	Incidental
8. Improve inspection efficiency from analysis of results	Yes	No

Source: Fagan 1976

Formal Inspection Process



Source: J Kelly, 1987

JCK:19

JPL

Primary Objectives by Stages

Process Stage	Objective	Participant(s)
1. Planning	Coordinate Inspection	Moderator
2. Overview	Education	Group
3. Preparation	Find errors/Education	Individual (all)
4. Inspection	Find errors	Group
5. Third hour	Discuss solutions and resolve discrepancies	Group
6. Rework	Fix Problems	Author
7. Follow-up	Ensure all fixes are correctly made	Moderator & Author

Source: Fagan 1976, Gilb 1987

JPL

Roles in Formal Inspections

- Moderator
- Author
- Reader
- Recorder
- Other Inspectors

JPL

Types Formal Inspections

- R0 Functional Design Inspection
- R1 Software Requirements Inspection
- I0 Architectural Design Inspection
- I1 Detailed Design Inspection
- I2 Source Code Inspection
- IT1 Test Plan Inspection
- IT2 Test Procedures & Functions Inspection

JPL

Basic Rules for Formal Inspections

- Inspections are carried out at a number of points inside designated phases of the software life cycle.
- Only technical documents and code are inspected.
- Inspections are carried out by peers representing the areas of the life cycle affected by the material being inspected (usually limited to 6 or less people).
- Management is not present during inspections. Inspections are not to be used as a tool to evaluate workers.

JPL

Basic Rules (continued)

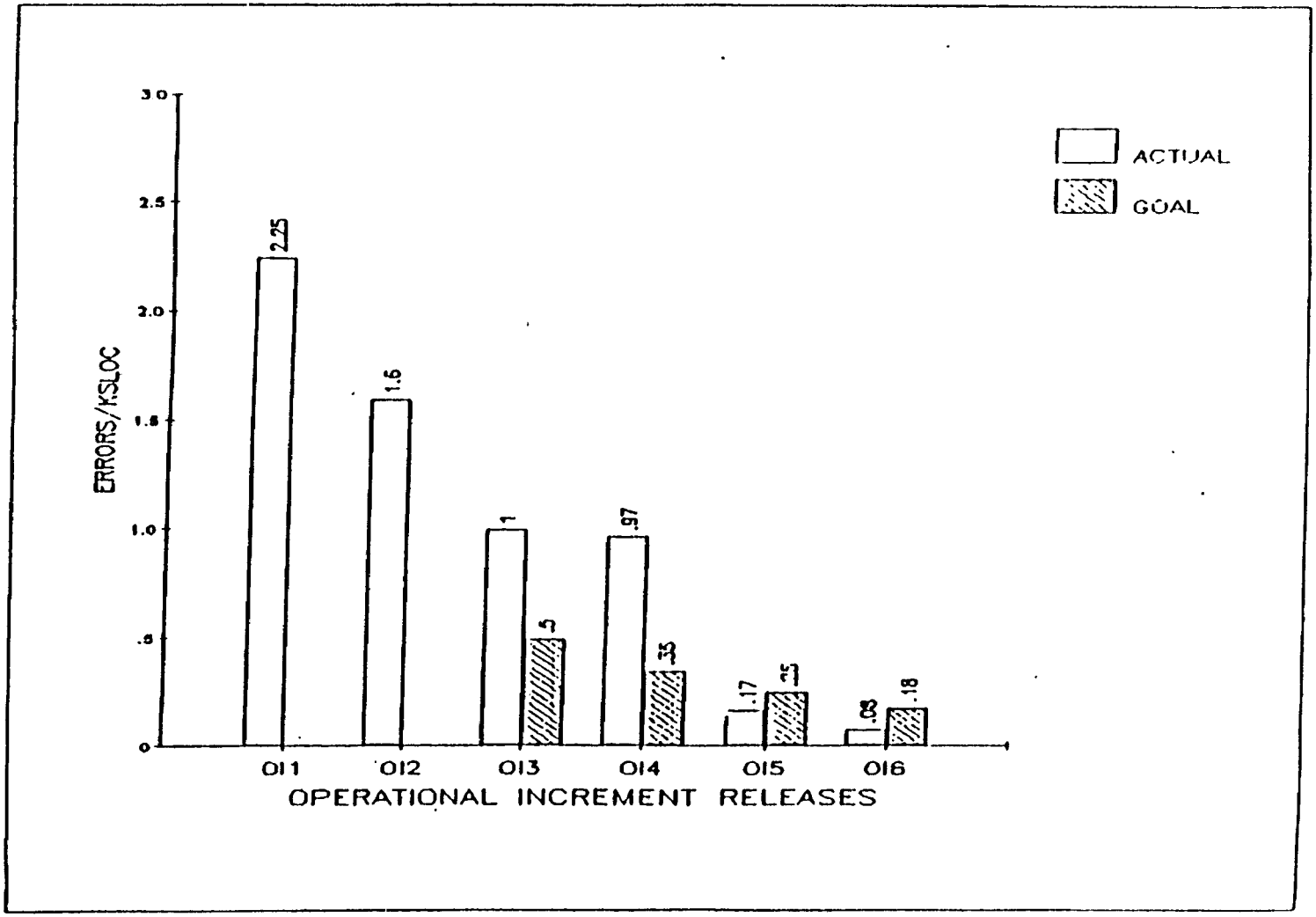
- Inspections are carried out in a prescribed series of steps.
- Inspection meetings are limited to two hours.
- Inspections are led by a trained moderator.
- Inspectors are assigned specific roles.

JPL

Basic Rules (continued)

- Checklists of questions are used to define the task and to stimulate defect finding.
- Material is inspected at a particular rate which has been found to give maximum error finding ability.
- Statistics on number and types of defects are kept.

Product Error Rates*
Shuttle's Primary Avionics Software Systems (PASS)
(After Introduction of Formal Inspection)



* Number of Valid Discrepancy Reports Post Delivery per KSLOC

Source: Kolkhorst, 1986

JPL

More Results from Projects
Formal Inspections

Project	Defects/Productivity
AETNA Life and Casualty 4,439 LOC	0 Defects in use 25% reduction development resource
IBM Respond, U.K. 6,271 LOC	0 Defects in use 9% reduction cost compared walk throughs
Standard Bank of S. Africa 143,000 LOC	0.15 Defects/KLOC in use 95% reduction in corrective maintenance cost
American Express 13,000 LOC	0.3 Defects/KLOC in use

Source: Fagan 1986

JPL

Requirement of Formal Inspection

or Similar Techniques by the

JPL Software Management Standard

"Peer review or technical walk-throughs shall be held on a regularly scheduled basis throughout the project/task"

--3.11.4, p. 3-26, JPL-D4000

Source: JPL D-4000

JPL

Walk-throughs vs. Formal Inspections
Comparison of two similar projects at IBM

	PROJECT X	PRS
Improved programming technologies	Yes	Yes
Reviews	Walk-throughs	Inspections[1]
Number of statements	10,000	6,250
Total detail design, code and test personnel	64 person months	41 person months
Duration	14 months	7 months
System Test Errors	51	11
Pilot installation	26	0 (also 0 errors in first 6 months of operation)
Total defects	77	11
Coding rate[2] (LOC/Person months)	155	153
Test error rate (Errors/KLOC)	7.7	1.76

1. Design, source code and test plan inspections only.
2. Includes time spent in design, code and test (including moderator's time for PRS).

Source: IBM Tech Report 1978

JPL

Benefits of Formal Inspections for Software Development

- Improved quality
- Contributes to project tracking
- Improved communication between developers
- Aids in the project education of personnel
- Cost savings through early fault detection and correction

Alka Shah

Bank of America
San Francisco, California

Topic: ***Is Certification Really Necessary?***

Alka Shah has 15 years of experience in software development, 10 years of which has been in software quality assurance. She took her M.B.A. at British Tutorial University in Kenya, Nairobi. Ms. Shah is currently Vice President in charge of BankAmericard Member Project Management for Bank of America in San Francisco. She is also President of the Bay Area Quality Assurance Association, a member of the Board of Directors for ASTE, and a member of the Project Management Institute.

SOFTWARE CERTIFICATION

TEST PROFESSIONALS

- Test Manager
- EDP Auditor
- Senior Test Engineer
- Test Engineer
- Test Technician
- System Assurance Engineer

What is really needed in today's market ?

RECOGNITION

- Experience
- Certificate of Performance

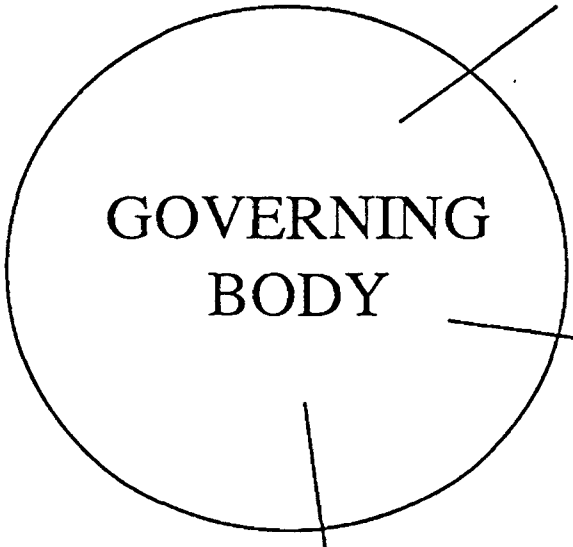
network

universities

organizations

?

GOVERNING BODY



- Define skill assessment
- Identify standards & procedures for testing
- Update materials periodically with the progress and changes

TESTING

- The process of executing a program with the intent of finding errors.

- An activity which certifies that after following a set process, a product functions as specified in:

  - User Documentation
  - Final Product Requirements
  - Product Specifications

- A method of identifying defects in software

- To make sure that the product does not abend

etc.
etc.

TESTING:

WHAT ARE THE GOALS?

- Requirements (specifications)
- Zero Defect
- Acceptable Quality Level
- User Acceptance Criteria
- Cost of Quality

(source: R. Robinson)

REQUIREMENTS / SPECIFICATIONS:

- What is the requirement?
- Is it adequately specific?
- Can it be quantified in some manner?
- Can the product be tested so as to demonstrate the specification, or is it an illusion?
- What are the success criteria for each requirement?

ZERO DEFECT:

- Is this a real possibility or a dream?
- What is a defect?
- At what point is the measurement taken?
- Does zero defect make business sense?

ACCEPTABLE QUALITY LEVEL

- Does the product fit within reasonable tolerances?
- How was "Reasonable" defined?
- What is the cost of failure and debugging?

USER ACCEPTANCE CRITERIA

- Who ultimately pays the bills / buys the product?
- What have they been promised?
- What are their real expectations? Reasonable?
- What are their business risks?
- What are their comfort zone with system failure?
- What will it take to get repeat business?

COST OF QUALITY

- Quality is free, but is perfection worth the price?
- If the cost of quality is high, zero defect is cheap
- There is a place for quick and dirty, but is this it?
- How often does the end user get a real say in how much testing?

TESTING:
KNOWLEDGE & TOOLS

- Test Plan
- Test Scripts
- Test Tools
- Incident Reports
- Regression Testing
- Risk Assessment
- Implementation Test Specifications
- Installation Verifications
- Production Release Report
- Performance Analysis
- Performance Evaluation Report

CONTENTS OF A TEST PLAN

1. Introduction / Overview
2. Project Functionality
3. Objectives of Test
4. Completion Criteria
5. Schedules
6. Resources
7. Responsibility by Phase
8. Tools
9. Integration
10. Tracking Procedures, CM & Problem Reports
11. Pass / Fail Criteria
12. Risk / Contingencies
13. Training
14. Test Cases

QUICK & DIRTY TEST PLANS

- Limited time to write a test plan prior to starting the test.
- Goal is to provide a testing framework to conduct current tests.
- Approach
  1. Review document set
  2. Write boilerplate section of the plan (General sections required by company)
  3. Outline tests that must be done
  4. Write test cases for initial test set
  5. Make initial schedule estimates
  6. Expand minimum test set using project documentation
  7. Write additional test cases

(source: Guy Jenkins)

ISSUES

- Testing Importance
- How to Control Testing Cost
- Reduce Maintenance Cost

WHAT ARE SOME OF THE CHARACTERISTICS A USER LOOKS FOR IN SOFTWARE

- Usability (friendliness)
- Reliability
- Maintainability
- Integrity
- Flexibility

CHARACTERISTICS

• Usability	Operability Training
• Reliability	Error Tolerance Consistency Accuracy Simplicity
• Maintainability	Consistency
• Integrity	Access Control Access Audit
• Flexibility	Modularity Generality Expendability

CONCEPTS OF TESTING

- Top down testing
- Bottom up testing
- Integration testing
- Big bang testing
- White box testing (unit testing)
- Black box testing

CERTIFICATION REQUIREMENT

- Experience
- References
- Course / Examination

RE-CERTIFICATION

- Is it necessary?
- Benefits:
  - Candidates gains latest knowledge
  - Organization benefits by continuing membership
  - Greater recognition in the D P environment

GRAND FATHERING

William G. Bently

Miles, Inc.
Mishawaka, Indiana

Topic: ***Automated Software Testing: Advanced Technologies***

William Bently is in charge of the data management R&D group at Miles, Inc. This group develops high-quality medical software products for diabetes care and urinalysis. Previous to entering the biomedical computer field six years ago, Mr. Bently headed a small R&D group which developed realtime software. His multi-disciplinary interests are evident in his educational background: a B.A. in mathematics from Oberlin College and an M.S. in biology from Ball State University.

AUTOMATED SOFTWARE TESTING: ADVANCED TECHNOLOGIES

W.G. Bently

OVERVIEW

This paper addresses a specific automated software testing technology; path testing. Several proposed strategies will be reviewed and a new method, Ct with K=1 coverage, will be presented. This concept was developed by Edward Miller at Software Research, Inc. and is being employed experimentally in analyzing a 27,000 line C program developed at Miles, Inc. The viewpoint expressed in this paper will be that of a practitioner.

THE NEED FOR A THEORY OF PROGRAM BEHAVIOR

A thorough test of a program would elicit its behavior over the entire input space. In practice, we are constrained to small samples of the input space. The ideal solution to this problem would be to develop a method for the selection of a sample that would be necessary and sufficient for proving the program has no errors; i.e. equivalent to an exhaustive test. The methods discussed in this paper are not sufficient (Goodenough 1975), but they can be seen to be intuitively necessary, and hopefully will help move the technology closer to the ultimate goal of sufficiency.

Succession of metrics
FIGURE 1

In order to know how much program behavior has been observed, it will be necessary to develop a theory of program behavior. An analogy with algorithm design will illustrate the need for such a theory. Humans design algorithms on the basis of a few cases, and often use partial execution to refine the algorithm (Kant 1985).

Algorithm development
FIGURE 2

After the algorithm is implemented as code, the testing staff may add a few more cases. But how many test cases are necessary? How many are sufficient?

Algorithm testing
FIGURE 3

THE NEED FOR A SCIENCE OF SOFTWARE TESTING

Science is based on measurement. We need instruments that quantify the software behavior observed; automated instruments that can be uniformly applied during the testing process. The intuition and experience of the practitioner are fallable, and become less useful as programs become larger and more complex. On the other hand, intuition and experience can serve as a valuable guide in the development of automated tools that greatly extend our capabilities of observing and measuring program behavior.

One such instrument is TCAT, a product of Software Research, Inc., which yields a branch coverage metric. The C1 coverage metric is based on the notion that it is necessary, during testing, to exercise each decision branch within the program at least once. TCAT has proven to be a practical and productive tool during the testing of all Miles' data management software products. The primary purpose of branch coverage has been the identification of missing test cases. The coverage concept has also proven to be valuable as a guide during the walkthroughs that preceed writing of test harnesses.

C1 is necessary, but not sufficient. A simple program consisting of two decisions may be used to illustrate this point.

Eg. 1 - Digraph
FIGURE 4

Exercising the paths;

a b d e g
a c d f g

yields 100% C1 coverage. Yet errors may occur in the paths;

a b d f g
a c d e g

The path testing methods discussed in this paper represent various ways of addressing this notion that it is necessary to test a set of paths that is larger and more diverse than a typical C1 cover.

REVIEW OF TESTING THEORIES

Ultimately, we are testing the correspondence between intended program behavior and actual program behavior. Testing theories are based upon different sources of information on intended and actual program behavior. Functional testing is based on specifications, whereas structural testing is based on program implementation.

Structural methods are more easily automated, and lead to an interactive style of testing. These methods elicit program behavior under controlled conditions, allowing the tester to observe the behavior and compare it with expected results. This process often "draws out" of the human mind information regarding the program that is difficult, if not impossible to capture in the form of rigorous specifications. Structural test methods are divided into two categories; those based on data flow and those based on control flow.

Testing theories
FIGURE 5

In this paper, the focus will be on control flow. Although control flow methods are not sufficient, they are necessary. On a deeper level, control flow methods are intimately related to program proofs (Howden 1976).

This does not mean that control flow should be used exclusively. Each testing strategy has been found to be effective in discovering different classes of errors.

For this reason, the different strategies should be viewed as complementary rather than competing (Woodward 1986).

PATH TESTING IS THE NEXT LOGICAL STEP

In figure 1, the coverage measures were listed in order of increasing effectivity. The state-of-the-art is currently somewhere between branch, which is routine practice in some shops, and full path, which is impossible. Path testing is significantly better than branch testing (Howden 1976), and is therefore worth pursuing.

LIMITATIONS OF PATH TESTING

Path testing shares all the insufficiencies of control flow testing, such as the inability to reveal missing functions. There are difficulties related to path testing (Howden 1987);

1. A fault may require that a path be iterated a large number of times.

2. A fault may require a complex pattern of path iteration in two or more loops.

3. There may be infeasible paths. (An infeasible path contains contradictory conditions and is therefore not executable. These are difficult to find, since the conditionals may depend on the changing values of variables in a complex way.)

4. There may be too many paths to test, even after eliminating infeasible paths. The number of paths is potentially infinite due to the presence of loops.

PATH SELECTION

Out of the potential infinity of paths, what constitutes an effective sample? Investigators have suggested a variety of methods that constrain the set of paths to be examined.

Path testing strategies
FIGURE 6

Some of these strategies are;

1. BOUNDARY INTERIOR (Howden 1975) - A classification is proposed based upon the way a path traverses a loop. The idea is to cover the boundaries by minimal traversal and maximum traversal of the loop.

2. LEVEL-i (Paige 1978) - Level-i paths attempt to capture the notion of depth of nesting of iterations. The level-i paths will include a basis set of paths, i.e. a set of paths such that any path through the graph may be expressed as a linear combination of paths in the basis set.

Special path definitions
FIGURE 7

3. LCSAJs (Woodward 1984) - Woodward has proposed several levels of coverage that begin with the customary statement and branch coverage. Higher levels are based on LCSAJs and pairs of LCSAJs. The LCSAJs for Example 1 (first introduced in Figure 4) are listed in Figure 8.

LCSAJ cover
FIGURE 8

(Woodward 1984) has demonstrated that LCSAJ testing is a reasonable next step beyond branch testing, since the number of added paths (beyond branch testing) is fairly small.

Woodward 1984
FIGURE 9

4. DATA FLOW (Fosdick 1976) - In this approach, the two major structural analysis methods are combined. Data flow analysis evolved from global optimization techniques used within compilers. This method is based on our intuitive notions regarding the reading and writing of variables. It is intuitively plausible, that each path between assignment of a variable, and usage of that variable should be executed Although this is an appealing concept, there are some practical difficulties related to the tracking of array elements, members of structures and variables identified by pointers (Frankl 1986).

5. BOUND ON ITERATION (Sneed 1986) - Sneed has reported a commercial tool, SOFTEST, which measures branch and forward path coverage (paths without cycles). The Ct metric also falls into this general category.

DIGRAPH

Control flow may be represented by a directed graph. In order to illustrate Ct coverage, it is necessary to first set forth some informal definitions, which are given in Figure 10.

Digraph definitions
FIGURE 10

Ct K=1 TEST COVERAGE METRIC

The basic concept in Ct coverage is to keep track of how many times each loop is traversed. The minimum iteration count 'K' is specified for a given test. The use of Ct K=1 coverage is similar to the strategy employed in C1 coverage. In C1, each decision outcome is exercised at least once. In Ct K=1, each loop in the program is exercised at least once. This covers loop initialization problems and corresponds with the intuitive notion of necessity.

The concept of Ct coverage and associated issues will be presented informally through a series of examples that have been adapted from (Miller 1988). Figure 11 lists the Ct K=Ø paths for example 1, the example program first presented in Figure 4.

Eg. 1 - No repetition
FIGURE 11

In this digraph, and all further digraphs, code segments will be represented by edges, since this makes the control flow easier to follow. In this example, there is no repetition, so there are no additional paths for K > Ø. Note that the Ct K=Ø paths are the same as the level-Ø paths. This may result in a general problem with Ct coverage, since the number of level-Ø paths can be very large, as witnessed in Woodward's results, which are displayed in Figure 9. The mean is somewhat misleading in this case, since most of the routines tested had less than one hundred level-Ø paths.

Example 2 illustrates a program with repetition; it has two simple loops, segments 'b' and 'd'.

Eg. 2 - Graph
FIGURE 12

Figure 13 lists the Ct paths for the cases K=Ø and K=1. The notation 'b' means that edge 'b' is executed exactly once. The notation '[b]' is used to indicate that edge 'b' is executed at least once.

Eg. 2 - Repetition K=Ø,1
FIGURE 13

Figure 14 lists the Ct paths for K=2. As K gets larger than one, the number of paths increases quickly, potentially limiting the usefulness of K>1 for analyzing large programs.

Eg. 2 - Repetition K=2
FIGURE 14

Since this work is experimental, the K=1 methodology is not yet fully developed. We are using path notation adapted from the theory of regular expressions. For instance, the '+' operator may be used to solve a possible problem with decisions which occur within loops. This is illustrated in Example 3.

Eg. 3 - Graph
FIGURE 15

Loops are enumerated by the way in which they are entered. The loop beginning with segment 'b' should be counted as a single case; it should not matter whether segment 'c' or 'd' is traversed. The Ct paths for Example 3 are listed in Figure 16.

Eg. 3 - '+' Operator
FIGURE 16

The '+' indicates that either segment 'c' or 'd' may be traversed on the given path. Another problem in the development of an automated tool for Ct coverage has been the presence of unstructured constructs in the 'C' language programs. 'C' allows branches out of loops, for example; "break", "return" and "exit."

Some preliminary Ct K=1 results have been derived from initial runs of the automated tool and checked manually. These results, which do not include the '+' correction, are shown in Figure 17.

Preliminary results
FIGURE 17

These results, along with other preliminary results, suggest that the 'C' functions in this program fall into one of two categories; those that have a managable number of paths (less than 100), and those that are "explosive." For instance, Ct can be used as a complexity metric. On the basis of these results, the largest function, "getscn," should be carefully examined, and perhaps partitioned into smaller functions. In some cases, functions with a large number of Ct paths, such as "putbox," may be testable, because of the correspondingly large number of infeasible paths.

FUTURE DIRECTIONS

There are several areas fundamental to path testing that require further investigation;

1. Refinement of the Ct K=1 coverage method.
2. Ct K>1.
3. A system level Ct K=1 tool (similar to STCAT, a system testing tool from Software Research, Inc.).
4. Infeasible paths.
5. Automatic generation of test cases.

We hope to have the first item completed by mid-year. The second area is inherently difficult due to the overwhelming number of paths, the extraordinary difficulty of constructing test cases and the complexity of controlling the iteration counts. The third area is a matter of engineering, and the fourth is an open question.

The last area is particularly interesting. This problem may be approached by using the finite state machine as a program model. Several investigators have suggested methods for generating test cases on the basis of a finite state machine model derived from functional specifications (Chow 1978, Bauer 1979). It should be feasible to derive the model from the control structure of the program. The state machine model is equivalent to the reduced digraph representation, and due to its universality, is capable of representing any program. In this model, the terminal and decision points within the program are interpreted as program states and represented by nodes in the diagram. A path becomes a sequence of state transitions, as represented by a sequence of arcs, taken from the entry node to the exit node. The advantage of this model is that there is a direct correspondence between the regular expressions accepted by the automaton and the equivalence classes determined by the path structure of the program. The automated test program would therefore; derive the state diagram from the source code, generate the set

of regular expressions, and convert these into test plans. A possible problem in this approach may be the large number of states generated by loops in the control structure of the program (Masuyama 1983).

REFERENCES

Bauer 1979
Bauer, J.A. and Finger, A.B., "Test plan generation using formal grammars," Proceedings of the 4th International Conference on Software Engineering, IEEE Computer Society, Sept. 1979, pp. 425-432.

Chow 1978
Chow, T.S., "Testing software design modeled by finite-state machines," IEEE Trans. Software Engineering, Vol. SE-4, May 1978, pp. 178-187.

Fosdick 1976
Fosdick, L.D. and Osterweil, L.J., "Data flow analysis in software reliability," ACM Computing Surveys, Vol. 8, Sept. 1976, pp. 305-330.

Frankl 1986
Frankl, P.G. and Weyuker, E.J., "Data flow testing in the presence of unexecutable paths," Proceedings of the Workshop on Software Testing, IEEE Computer Society Press, July 1986 pp. 4-13.

Goodenough 1975
Goodenough, J.B. and Gerhart, S.L., "Toward a theory of test data selection," IEEE Trans. on Software Engineering, June 1975, pp. 156-173.

Howden 1975
Howden, W.E., "Methodology for the generation of program test data," IEEE Trans. Computers, vol C-24, May 1975, pp. 554-560.

Howden 1976
Howden, W.E., "Reliability of the path analysis testing strategy," IEEE Trans. on Software Engineering Vol. SE-2, Sept. 1976, pp. 208-215.

Howden 1987
Howden, W.E., Functional Program Testing and Analysis, McGraw-Hill, 1987, pp. 96-99.

Kant 1985
Kant, E., "Understanding and automating algorithm design," IEEE Trans. on Software Engineering, Vol. SE-11, November 1985, pp. 1361-1374.

Masuyama 1983
Masuyama, H., "A software function test design using state transition diagram," Trans. (D), I.E.C.E., Japan, Vol. 66-D, Nov. 1983, pp. 1294-1301.

Miller 1988
Miller, E.F., "Technology Brief: Ct test coverage metric - technical explanation", Software Research, Inc., April 1988.

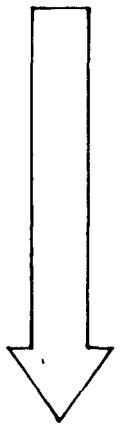
Paige 1978
Paige, M.R., "An analytical approach to software testing, IEEE COMPSAC 78, pp. 527-532.

Sneed 1986
Sneed, H.M., "Data coverage measurement in program testing," Proceedings of the Workshop on Software Testing, IEEE Computer Society Press, July 1986, pp. 34-40.

Woodward 1984
Woodward, M.R., "An investigation into program paths and their representations," Proceedings of the Second Software Engineering Conference, Nice, France, E. Girard (Ed.), June 1984, pp. 209-216.

Woodward 1986
Girgis, M.R. and Woodward, M.R., "An experimental comparison of the error exposing ability of program testing criteria," Proceedings of the Workshop on Software Testing, IEEE Computer Society Press, July 1986 pp. 64-73.

NECESSARY



SUFFICIENT

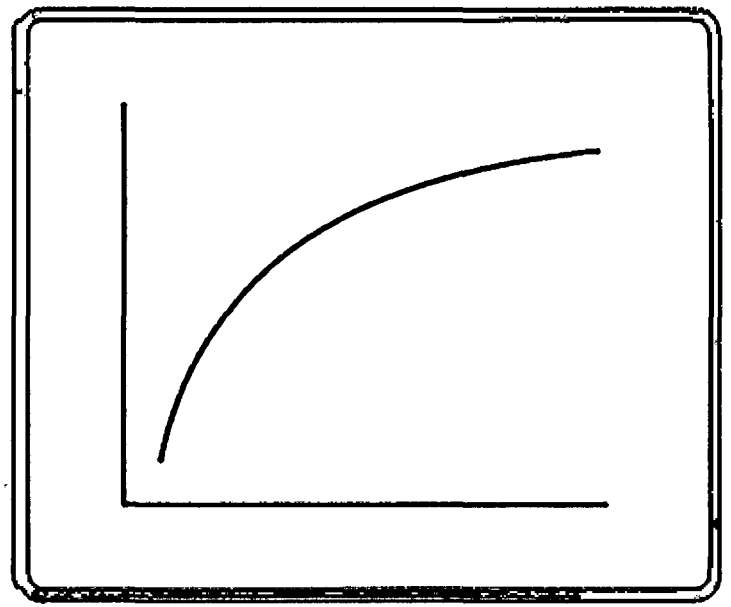
STATEMENT *C0*

BRANCH *C1*

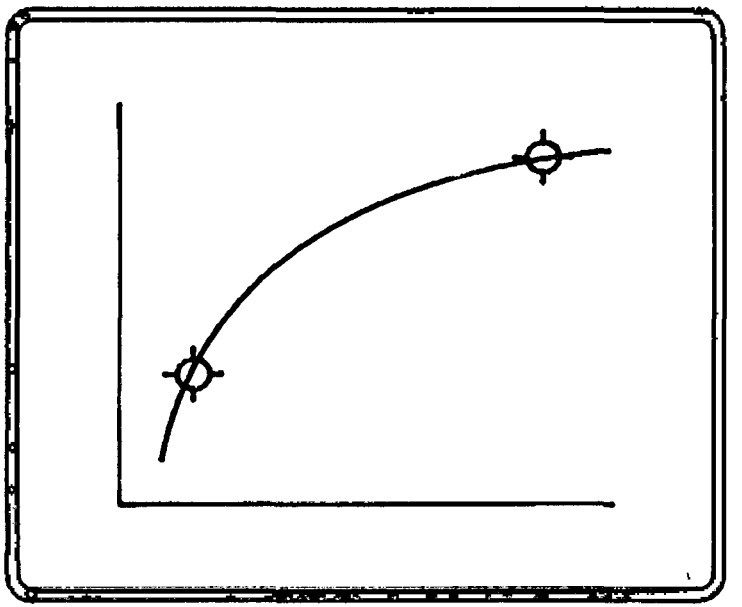
PATH *Ct*

PROOF

SUCCESSION OF METRICS
PATH TESTING

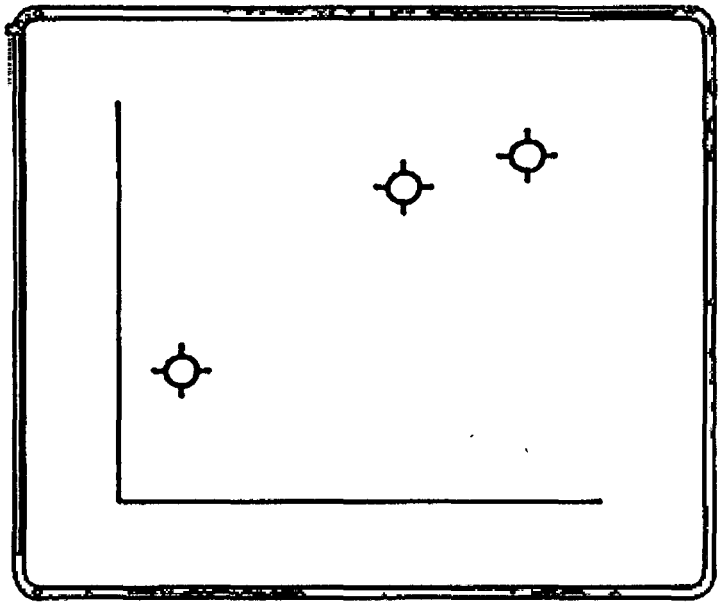


INTENDED BEHAVIOR

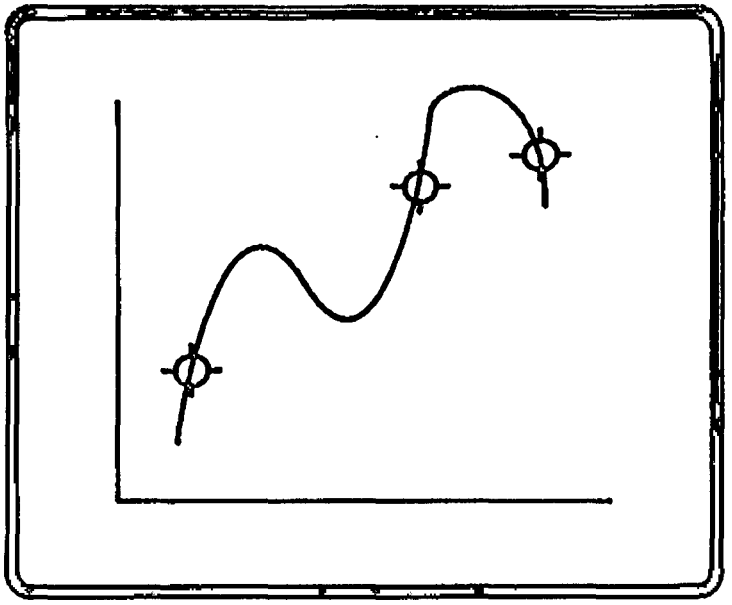


DESIGN CASES

ALGORITHM DEVELOPMENT
PATH TESTING

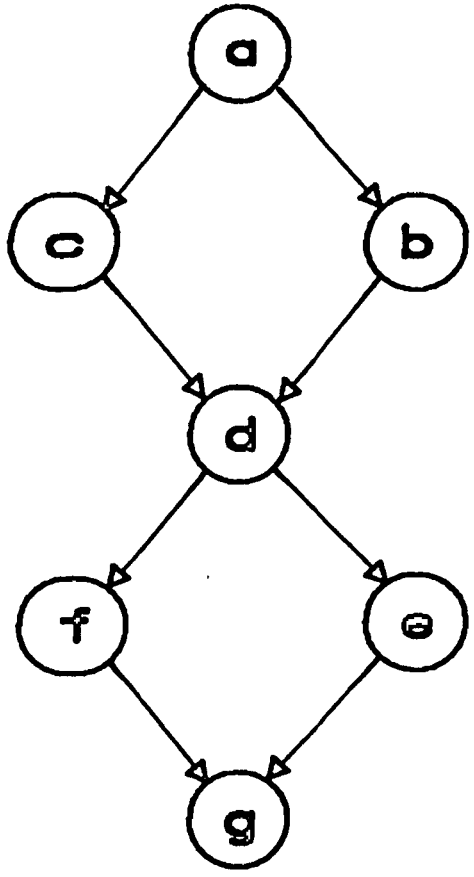


TESTS LOOK GOOD



ACTUAL BEHAVIOR

ALGORITHM TESTING
PATH TESTING



SEGMENTS

a b c d e f g

PREDICATES

x y

```
a
IF (x)
                b
ELSE
                c
END
d
IF (y)
                e
ELSE
                f
END
g
```

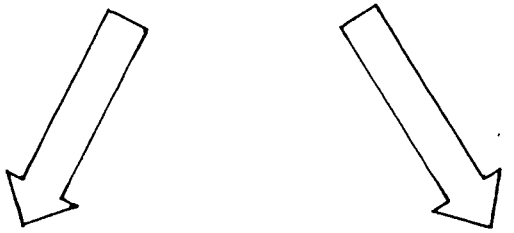
EG. #1 - DIGRAPH PATH TESTING

FUNCTIONAL

SPECIFICATIONS

STRUCTURAL

CODE



DATA FLOW

CONTROL FLOW

TESTING THEORIES
PATH TESTING

1. BOUNDARY INTERIOR (Howden)

2. LEVEL-i (Miller, Paige)

3. LCSAJ (Woodward, Hennell)

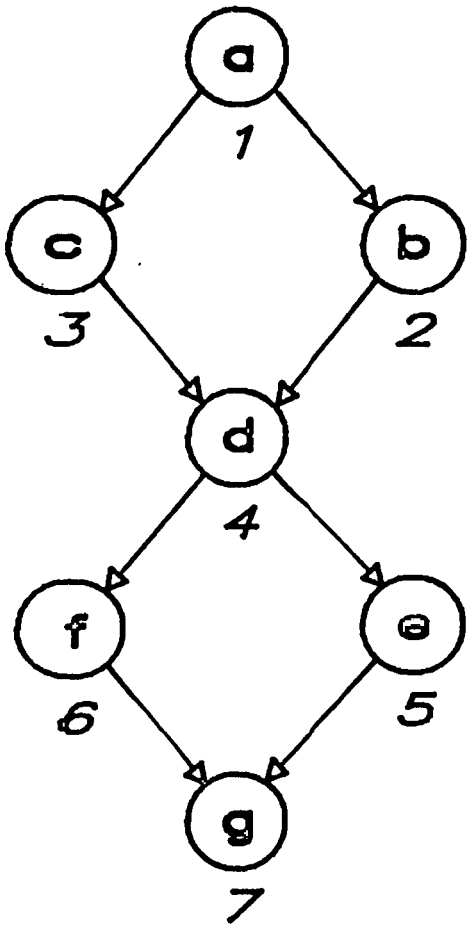
4. DATA FLOW (Osterweil, Fosdick)

5. BOUND ON ITERATION (Sneed)

PATH TESTING STRATEGIES
PATH TESTING

DD-PATH	path which begins on a decision node or entry node and ends on a decision node or exit node, and contains no other decision nodes
LEVEL-i PATH	simple path that begins and ends on nodes of some lower level paths; and remaining nodes are not on any lower level paths
LCSAJ	Linear-Code-Sequence-and-Jump (or JJ-path) is a path which begins at either the target node of a jump or an entry node and ends either on a program jump or an exit node, and contains no other jumps

SPECIAL PATH DEFINITIONS
PATH TESTING



LCSAJs

1: a a c
2: a b d
3: c d f
4: c d e f
5: d d f
6: d e g

MINIMUM PATH LCSAJ COVER

1: a b d e g
2: a c d f g
3: a c d e g

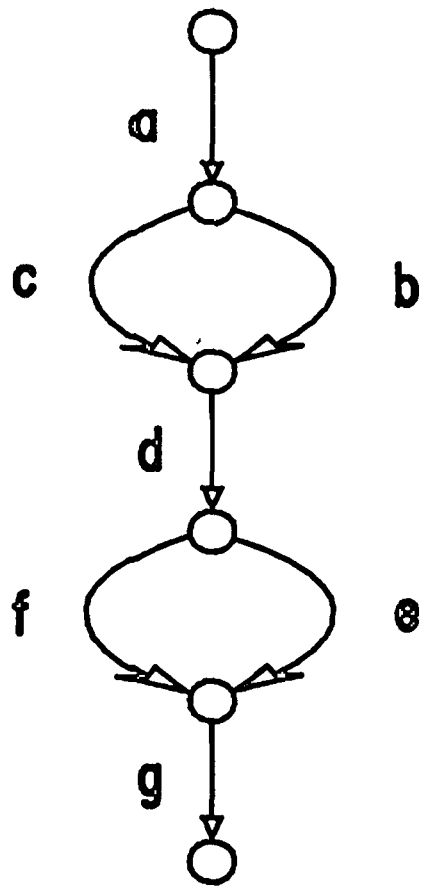
EG. #1 - LCSAJ COVER PATH TESTING

	MAX.	MEAN
BASIC BLOCKS	158	27.3
DD-PATHS	164	25.2
LEVEL-i PATHS	45,287,485	425,882.2
LEVEL-0 PATHS	45,287,476	425,558.3
LEVEL-1 PATHS	31,349	330.0
LEVEL-2 PATHS	18	0.4
LCSAJs	338	47.4
STATEMENT COVER	10	3.6
BRANCH COVER	11	4.1
LCSAJ COVER	63	9.7

WOODWARD 1984
PATH TESTING

NODE	graph primitive, consecutive sequence of statements with one entry and one exit
EDGE	directed arc between two nodes
(SUB)PATH	sequence of connected nodes
PATH	path from entry to exit node
SIMPLE PATH	path containing no node or edge more than once
SEGMENT	a simple path between two S-nodes which contains no other S-nodes (where S-node is an entry, exit, junction or decision node)

DIGRAPH DEFINITIONS
PATH TESTING



DD-paths

1: a b d
2: a c d
3: d e g
4: d f g

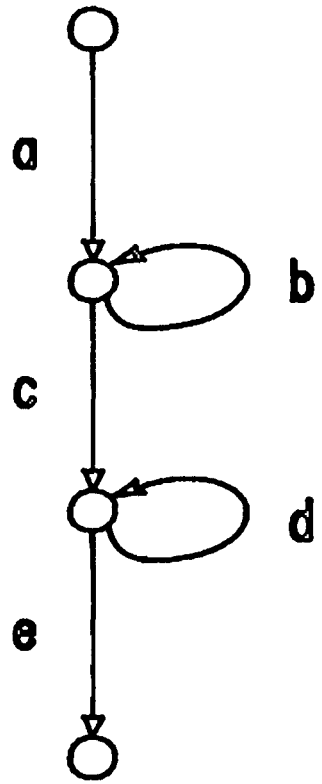
Ct K=0 (also level-0)

1: a b d e g
2: a c d e g
3: a b d f g
4: a c d f g

level-1

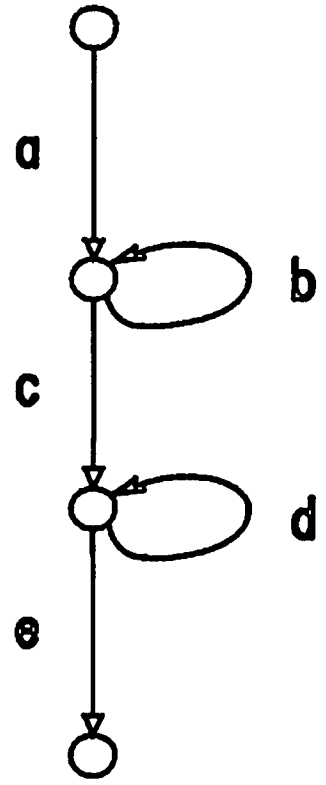
none

EG. #1 - NO REPETITION PATH TESTING




```
a
WHILE (x)
                b
END
c
WHILE (y)
                d
END
e
```

EG. #2 - GRAPH
PATH TESTING



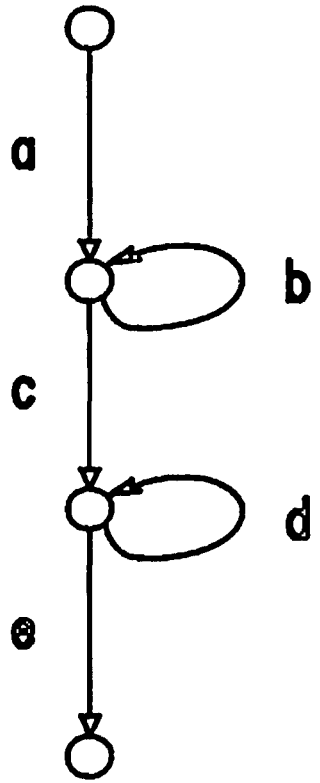
K=0

1: a c e

K=1

1: a c e
2: a c [d] e
3: a [b] c e
4: a [b] c [d] e

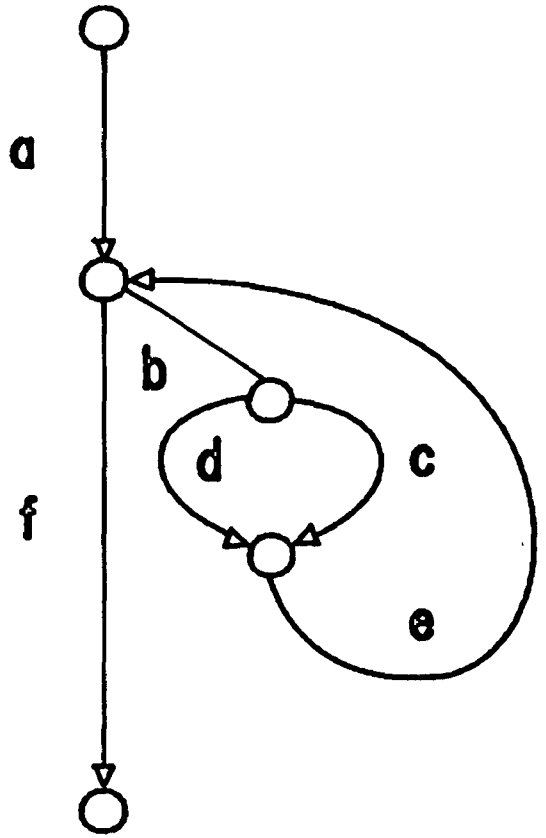
EG. #2 - REPETITION K=0,1
PATH TESTING



K=2

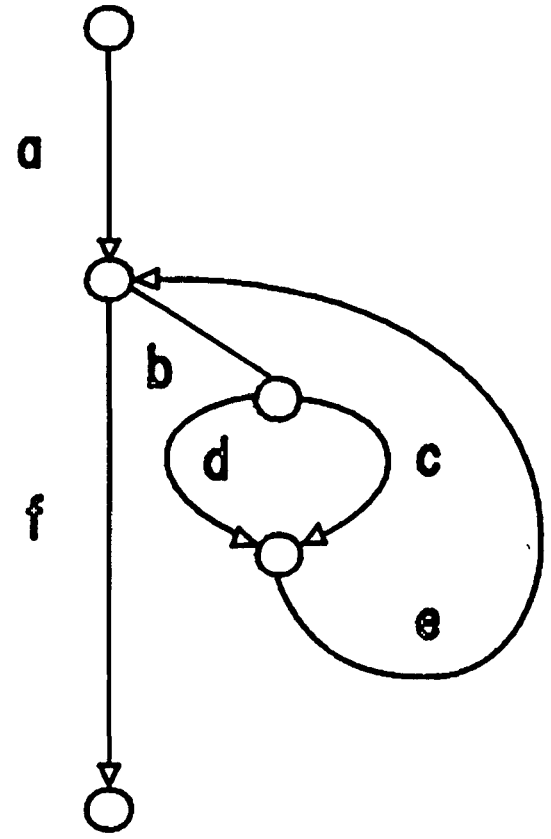
1:	a			c				e
2:	a			c	d			e
3:	a			c	d	[d]		e
4:	a	b		c				e
5:	a	b		c	d			e
6:	a	b		c	d	[d]		e
7:	a	b	[b]	c				e
8:	a	b	[b]	c	d			e
9:	a	b	[b]	c	d	[d]		e

EG. #2 - REPETITION K=2
PATH TESTING




```
a
WHILE (x)
                b
                IF (y)
                                c
                ELSE
                                d
                ENDIF
                e
ENDWHILE
f
```

EG. #3 - GRAPH PATH TESTING



K=0

1: a f

K=1

1: a f
2: a [b (c + d) e] f

K=2

1: a f
2: a b c e f
3: a b d e f
4: a b c e [b (c + d) e] f
5: a b d e [b (c + d) e] f

EG. #3 - '+' OPERATOR PATH TESTING

FUNCTION	LINES	Ct K=1 PATHS	INFEASIBLE
YNUM	30	6	3
GETFIL	42	5	0
PUTFLD	42	6	0
FKEYDS	44	18	0
CKSUM	82	73	0
PUTBOX	136	576	476
GETSCN	346	537,000,000	?

Ct K=1 PRELIMINARY RESULTS
PATH TESTING

Vondes Barnett

Federal Express
Memphis, Tennessee

Topic: ***Testing an Innovative Customer Automation System***

Vondes Barnett is Senior Quality Assurance Analyst for Federal Express. Mr. Barnett started his career in electronics in the U.S. Navy in the 1970's, and worked for 10 years with the Federal Aviation Administration as hardware technician and systems performance analyst. At Federal Express he was senior engineer on a satellite telecommunications project. His current assignment at FedEx is managing the development of a customer automated system which represents a giant step in computer-aided customer service.

Cem Kaner

Electronic Arts
San Mateo, California

Topic: ***Test Planning for Consumer Software***

Cem Kaner has extensive experience in software testing management, user interface design, and free-lance writing. He has published numerous papers, and his recently completed book *Testing Computer Software* will soon be released by TAB Books. Dr. Kaner is currently Manager of Software Testing in the Creativity Division of Electronic Arts. He previously supervised software testing projects at Micropro, and started up a Software Testing Department at Telenova where he worked as a Human Factors Analyst/Software Engineer, designing and coding the Telenova Station Set. Dr. Kaner holds a Ph.D. in Experimental Psychology from McMaster University in Hamilton, Canada and a B.A. in Math and Philosophy from Brock University, St. Catharines, Canada.

Peter L. Morse

Microsoft
Seattle, Washington

Topic: ***Automated Testing in an Interactive Environment***

Peter Morse is manager of Applications Testing for Microsoft Corporation. He holds a B.S. in Electrical Engineering and an M.S. in Computer Science. Prior to joining Microsoft he spent eight years as a Systems Engineer and Manager at Data General. Other experience includes work as a Systems Analyst and Electrical Engineer.

Douglas Hoffman

Informix Software
Menlo Park, California

Topic: ***Technology of Quality***

Douglas Hoffman has been Manager of Quality Assurance at Informix Software for over a year. Mr. Hoffman has been active for over 16 years in the softare quality field, having worked 10 years for Hewlett Packard and six years managing Q.A. and support organizations for start-up companies. Mr. Hoffman received a B.S. in Computer Science and an M.S. in Electrical Engineering from U.C. Santa Barbara and an M.B.A from Santa Clara University.

Darl P. Patrick

Sandia National Laboratories
Albuquerque, New Mexico

Topic: ***Certification of Hardware with Imbedded Software***

Darl Patrick has been in charge of Quality Assurance for automated test equipment at Sandia National Labs for eight years, and his current task at Sandia is certification of high-risk automated test equipment for the Department of Energy. Mr. Patrick spent 24 years in the military and holds an M.A. in Electrical Engineering from Naval Post Graduate School in Monterey, California.

QW 88

CERTIFICATION OF AUTOMATED TEST SUITES WITH EMBEDDED SOFTWARE

Darl P. Patrick, MTS

Sandia National Laboratories
Division 7252
Albuquerque, New Mexico 87185
(505) 844-8745

ABSTRACT

The use of automated inspection tools for certifying complex test equipment can provide the production and quality assurance engineers with a consistent set of metrics. Early identification of probable design faults, in both hardware and software by the use of automated inspection tools, yields a high-quality product in less time and with less direct cost.

INTRODUCTION

This article discusses the use of software tools being developed by Sandia National Laboratories for the certification of Automated Test Suites with embedded software for high risk weapons programs. The certification of automated testers is based primarily upon verification of specifications by "Black-Box" testing and evaluation of embedded software capability. These tools are presently being used throughout the life cycle of High-Risk Automated Testers and can result in the development of a "confidence number" which indicates the tester's conformance to released specifications. The amount of "excess capability" built into the tester with software, which was not tested during "Black-Box" evaluation can be identified.

AUTOMATED TOOL CONCEPTS

The tools discussed in this paper were developed independently as "stand-alone" tools to demonstrate feasibility. Sandia National Laboratories and the U.S. Army Product Assurance Directorate are jointly developing a "Quality Assurance Inspection Program", QAIP, which is an integrated program that allows use of a common data base by tool programs. The installed programs are able to interact with each other via the data base system information file, DBIF. Additional integrated tools, including user independent tools, will be accessible under the common executive program. QAIP is designed to allow diverse

users to use all or part of the tools in multiple operating system environments with multiple languages. QAIP is presently being targeted for the AT&T UNIX version 5 systems, with reduced MS-DOS capabilities. The use of QAIP in an integrated environment allows users to confidently certify both software and hardware high-risk projects in a consistent, reliable, and repeatable manner.

The use of automated tools to facilitate certification of either hardware or software can quickly lead the engineer into a false sense of security followed by total frustration. Automated tools are generally introduced into the work place with little experience as to which metrics are required, how to implement them, or how to evaluate the results. Classroom examples and textbook formulas do not transpose easily to the production environment.

Software Branch Analysis

Sandia National Laboratories Quality Assurance Division started using a manual branch analysis metric for reviewing software in 1983. At that time, there was no testing methodology for using this metric other than that used by Strum and Ward in evaluating linear circuits. T. McCabe of McCabe and Associates was working on the development of an automated branch analysis methodology, but had not implemented the practical usage. The expectations of testing were raised, but practical usage of the metric and consistent interpretation of the manually produced graphs proved difficult.

Three classifications of software testing are: functional testing, structural testing and Code Branch Analysis (walk throughs).

a. Functional testing: Referred to as "Black-Box" testing because the structure of the software is not tested, but rather the conformance of the system to specifications is tested. The software is tested via the hardware on a "stimuli vs response" concept. Two of the most widely accepted techniques for functional testing are boundary value analysis and equivalence class partitioning.

   Boundary value analysis detects errors at or near the boundaries. It is used to assure thorough testing at, above, and below the specification values plus any accuracy determinations.

   Equivalence class partitioning is a technique to partition the input domain into classes. If a test case from the class is executed and fails to find an error then any other test case in that class would also fail to locate an error. The technique is used to reduce the number of tests required.

Dynamic testing takes place once the software has been completed and is integrated into the host equipment. Test cases developed to stress the software are run and the execution observed in real time. The effectiveness of dynamic testing is entirely dependent upon the test cases developed from the specifications and from the static analysis.

One of the major weaknesses of Functional Testing is that there is no way of determining when testing is really completed. 100% testing based upon functional requirements may actually check only a small portion of the embedded software.

b. Structural testing: Referred to as "White-Box" testing, emphasizes the opposite testing approach and stresses the software performance rather than conformance to specifications. This is generally accomplished by monitoring the programs execution and/or by static analysis. Typically, it has been observed that full functional testing results in approximately 30% to 40 % of the software branches being covered. Partial functional testing based upon "best engineering judgement" has resulted in 15% to 18 % of the software branch paths being covered.

One of the most obvious benefits of structural testing is the identification of untested code. This is generally code not identified as a requirement but is implemented as a result of test methodology. Structural testing provides a confidence that the software is "good". In fact, the software could be fully structurally tested and be "good" and still fail to meet the functional requirements. It is for this reason the functional testing and the structural testing must be combined in a cohesive test plan. Structural testing can be further broken down into two phases, static and dynamic.

The static analysis of the software is best accomplished during the design and building of the program. Complex and poorly structured modules can be detected early and corrected. Static testing is a "White Box" evaluation at this level. The number of branches and nodes and the number of linearly independent test paths required to cover all branches can be determined.

Structural dynamic testing of software can be accomplished during development of the software. Dynamic testing can be done on each module or groups of modules using drivers and stubs. The static test cases generated during the static analysis can be used. Dynamic testing at this level is still "White-Box" verification. It is difficult to dynamically test modules against higher level specifications unless extensive stubs and drivers are developed. This often takes more time then the program under test.

c. Branch Complexity: The number of independent test paths required to cover all branches and nodes at least once is defined as the module or program "Branch Complexity Number". An important limitation to static branch analysis is the inability to detect time dependent events. Because the static branch analysis metric has no concept of timing, code which has timing errors may be passed as "acceptable". Figure 1 shows a typical sequence of activities when conducting static branch analysis on software. Figure 2 shows a similar sequence for dynamic analysis of the software.

d. Structured Walkthrough: The code formally is reviewed by a team of trained personnel. This methodology is often referred to as the AT&T or IBM approach. Working from released specifications the team follows a script in checking and evaluating the software. Although this is a labor intensive operation, if it is done correctly, the resultant code demonstrates errors of less than one per five-thousand lines of code.

Neither functional testing, structured testing, nor structured walkthrough used independently is sufficient to ensure satisfactory testing. They must be combined in a systematic method to ensure the greatest coverage with the least number of test cases.

Testing at the system level requires a different strategy. All drivers and stubs are eliminated. This forces the system to be run and tested in a "customer" mode. Branch analysis at this level is dependent upon the time required to complete a single run. To completely specify and run test cases at the system level using dynamic testing may require "hundreds" of test cases. While the testing is slow at this level, test validity is greatest. There are many reasons for this. All of the operational code is installed, all hardware interfaces are in place, tests are being run in a normal manner, and operational documentation is used. Errors encountered during this phase of testing are the most expensive to correct, but have the greatest impact on delivered operability of the system.

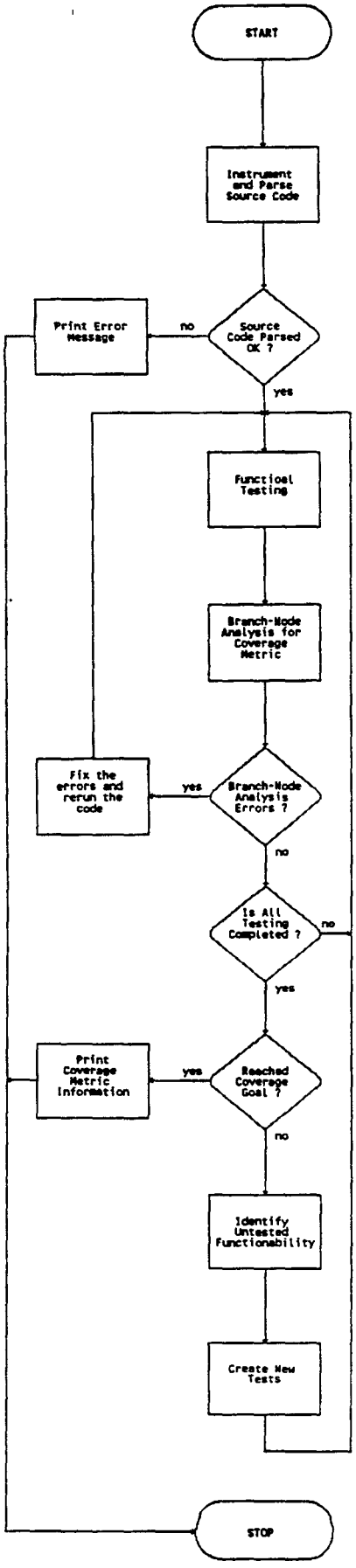

START
Instrument and Parse Source Code
Source Code Parsed OK ?
no
Print Error Message
yes
Functioal Testing
Branch-Node Analysis for Coverage Metric
Branch-Node Analysis Errors ?
yes
Fix the errors and rerun the code
no
Is All Testing Completed ?
no
yes
Reached Coverage Goal ?
yes
Print Coverage Metric Information
no
Identify Untested Functionability
Create New Tests
STOP

Figure 1

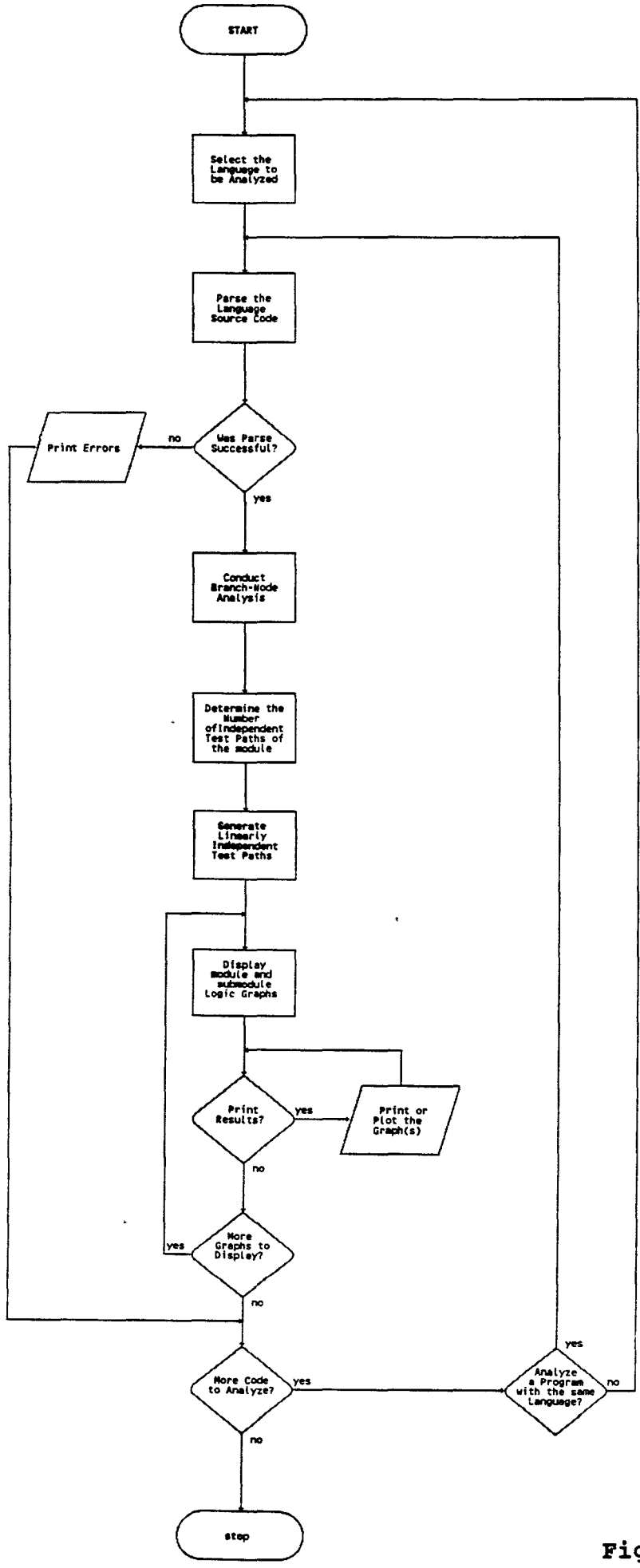

START
Select the Language to be Analyzed
Parse the Language Source Code
Was Parse Successful?
no
Print Errors
yes
Conduct Branch-Node Analysis
Determine the Number ofIndependent Test Paths of the module
Generate Linearly Independent Test Paths
Display module and submodule Logic Graphs
Print Results?
yes
Print or Plot the Graph(s)
no
More Graphs to Display?
yes
no
More Code to Analyze?
yes
Analyze a Program with the same Language?
yes
no
no
Stop

Figure 2

Branch Analysis Errors

One of the major misunderstandings made of the branch analysis was the expectation that software which had low module complexity would have low error rates, and software which had high branch coverage would have a low number of residual errors. Data and experience indicates that there is no inherent direct correlation between high branch coverage and low defect rates, or low module complexity and low defect rates. The branch coverage is meaningless if test cases are not generated properly. Low module or system complexity does not indicate the true state of the testability of the software.

Many of the problems encountered when the branch analysis metric was first implemented were caused by setting unrealistically high branch coverage requirements and low module complexity requirements. There is a proper place for each of these metrics to be used in the lifecycle. At the module level of development the static branch metric, module-complexity, can be used as an indicator of the module. Modules which have high complexity values generally indicate a lack of understanding of the event, or a weakness in the specifications. It would be a simple task to conduct branch analysis on all software at this level. The tester knows the code, and error conditions can be forced. The use of Stubs and Drivers can facilitate execution of the code. Unfortunately testing at this level has a limited value. The tests being conducted are inductive of the software structure and not of any specification functionality. It is often trivial to achieve 100% coverage at this level and develop a false sense of security, with no real test results being generated. The static metric is of value, but should be used with full understanding of the metrics limitations.

QUALITY ASSURANCE CONCEPTS

Quality Assurance is a collection of techniques which seek to assure the hardware and software actually function as defined by the specifications. This type of activity is part of the whole lifecycle, but special methods are employed to focus on the certification of tester hardware and software quality issues.

There are many approaches used to accomplish quality objectives. The automated tools being developed by Sandia National Laboratories stress certification of automated testers against released specifications using demonstrable test cases.

The Quality Assurance Inspection Program, QAIP, being developed by Sandia National Laboratories, is a software executive program which provides user interface to diverse quality assurance programs. There are presently three such programs which can be accessed from the QAIP main menu:

a. "T": An automated and computer-aided test generation tool, developed under an Army contract by Product Environment Inc. (PEI). Specifications are entered in response to prompts, valid as well as invalid test cases are generated for use in "black-box" testing. The test cases generated by "T" confirm that the specifications are implemented by either the hardware design or the software design.

b. "S_PATm": The Software Path Analysis Tool for software modules is a static tool developed for Sandia National Laboratories based upon development work done by Strum and Word of the Naval Post Graduate School, McCabe and Associates, and the U.S. Army Product Assurance Directorate Technology Office. The S_PATm tool parses and analyses the source code for a specified language at the module level and displays a decision logic diagram of the module's structure, from which software or "white-box" test cases can be generated.

c. "TCAT: The Test Complexity Analysis Tool is one of a group of software dynamic tools developed by Software Research Associates (SRA), and modified with Sandia National Laboratories for the Hewlett-Packard Advanced Basic. The TCAT dynamic tool, at the module level, instruments the program and provides dynamic traces at the module level. The TCAT is used with the test cases generated with the "T" tool, S_PATm tool, or other test-case development methods.

PIE: A Peripheral Interface Emulator, is a hardware "smart-box" interface which allows the programmer to control the data returned when a program accesses an external device such as a

digital multimeter. It can be programmed to compare "as-read" values with specification limits. PIE allows partial dynamic checkout of tester software using test cases developed without requiring the actual peripheral devices to be physically present.

The programmer is able to enter the "device" addresses, expected return value and specification limits. When the program is executed, it outputs data to the desired address and receives data. The results of the received data will cause valid as well as invalid test cases to be evaluated. The "PIE" box can store the test, the time of the test, the prompt, the value returned, and the correct limit for later data dump and evaluation.

The use of "PIE" should enable programmers to develop the test software independent of the delivered hardware. Full independent software evaluation is not feasible due to timing constraints and unique "smart-chassis", however PIE should allow early delivery of a higher quality software prior to hardware/software integration.

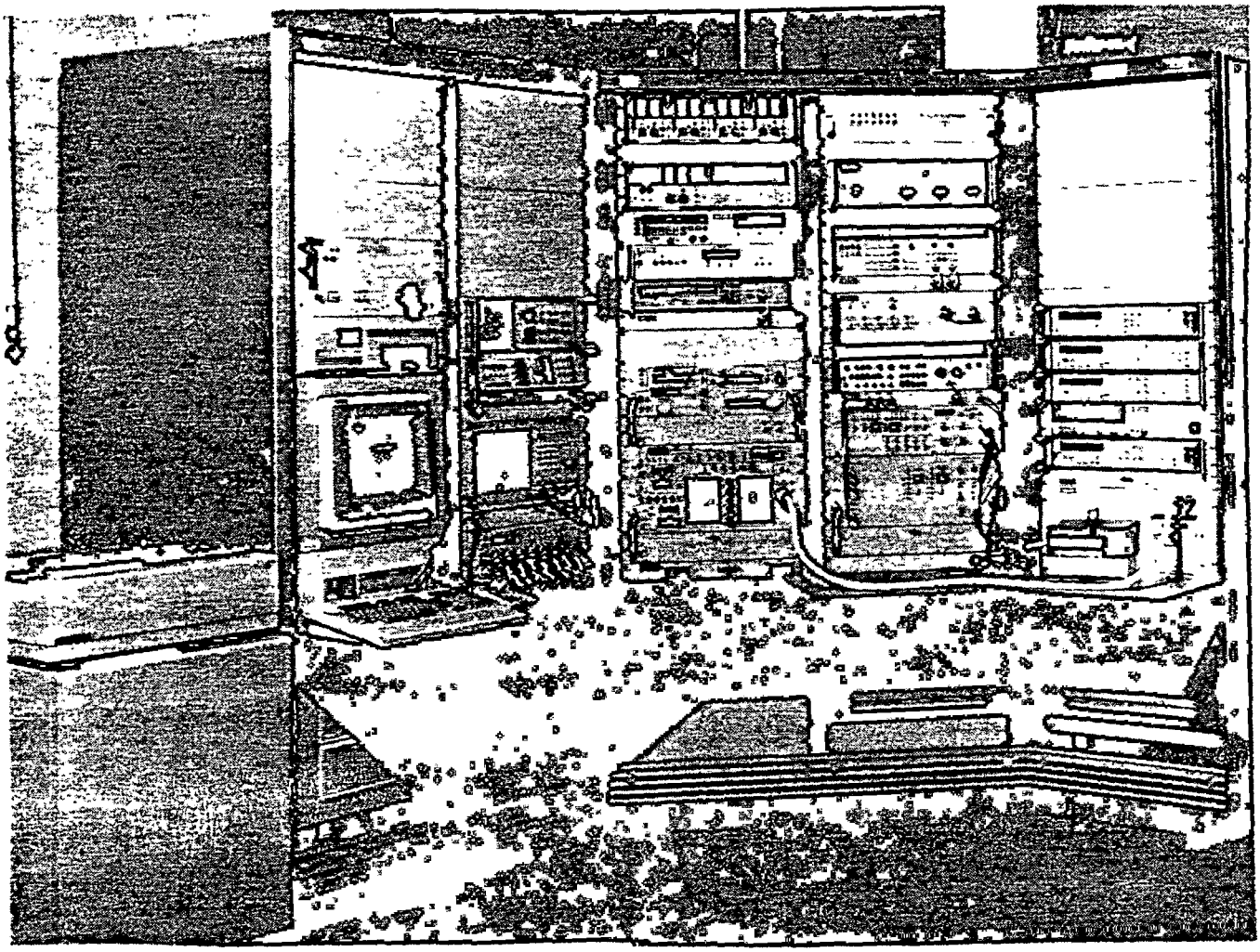


Figure 3

AUTOMATED TEST EQUIPMENT LIFECYCLE

Figure 3 is representative of test systems certified by Sandia National Laboratories for the Department of Energy. In the academic lifecycle of high-risk test suites the development of the hardware and the software occurs only after the specifications have been reviewed, corrected and issued, as shown if Figure 4.

Once the system requirements are released, both the software and the hardware design, test, and review processes begin and continue nearly in parallel. Upon completion of the hardware, the software which is ready to be installed, is installed in the hardware and integration testing commences.

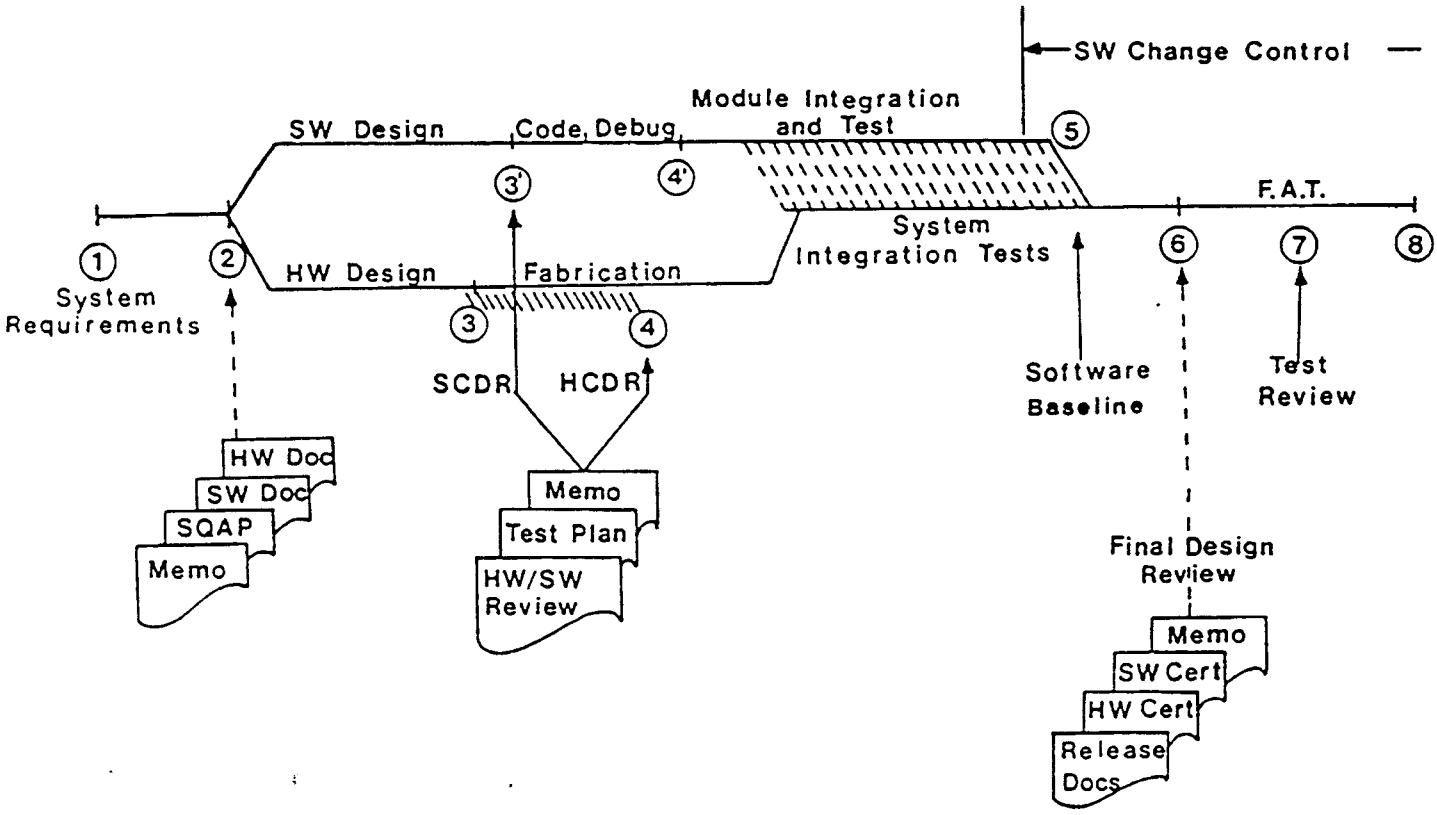


Figure 4

In fact the actual development of the test suite more accurately follows the lifecycle shown in Figure 5.

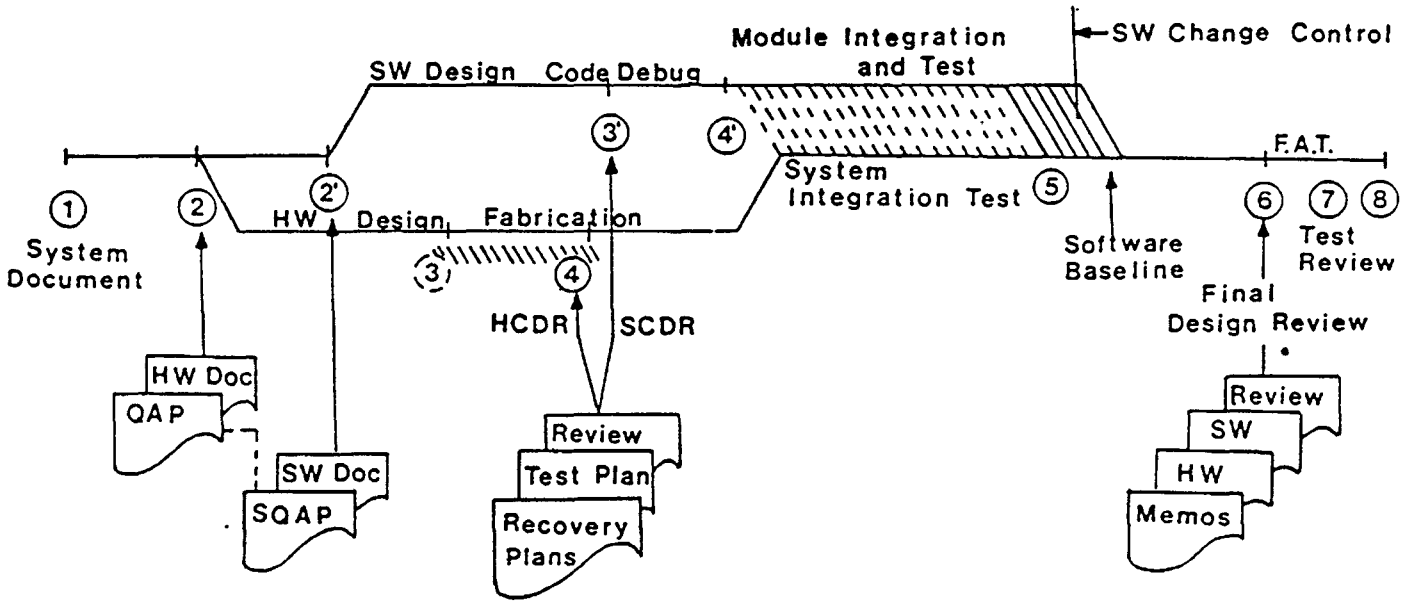


Figure 5

The specifications are "soft" at the beginning of the project, but funding has been allocated and hardware placed on order. The hardware design continues, based upon best estimates, with the expectation that future changes can be incorporated via software. The software is never defined with firm specifications. It remains a "shadow" of the system functional specifications. It is natural therefor for the software development to lag the hardware. Some early software work is done without firm software specifications, automated tools or emulators. Once the hardware completes initial fabrication, both the hardware and the software engineer vie for limited resources. The general result is that the software is integrated into the tester and "debugged" on the fly with the earnest hope that there is enough time.

Sandia National Laboratories has enhanced the Product Quality Assurance Team (PQT) concept, used in certifying hardware and software, for the use of automated inspection tools. Under the PQT concept the Quality Assurance, Design, Test Equipment and Product Engineers form a Product Quality Assurance Team early in the tester life cycle. Figure 6 shows the product lifecycle with the PQT concepts and the automated tools combined.

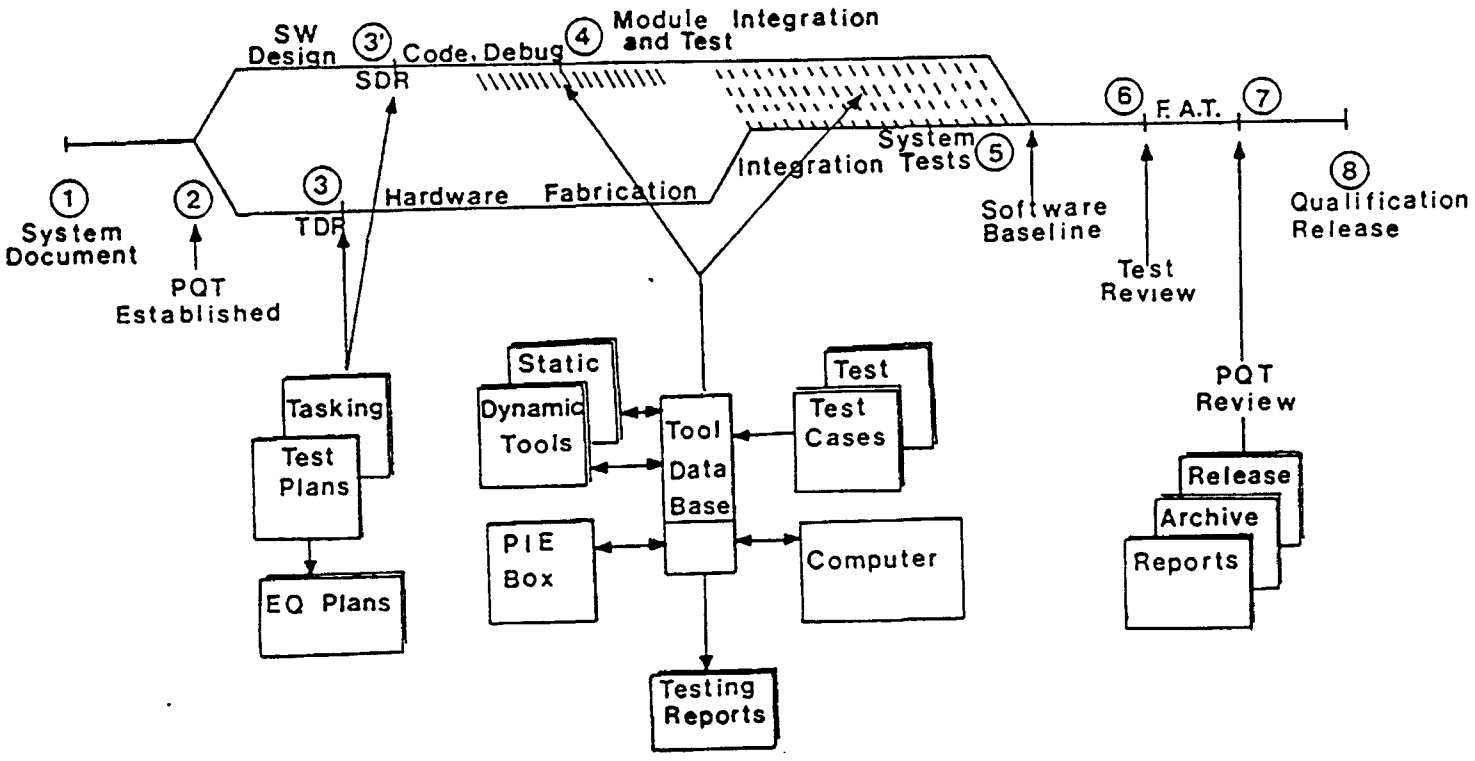


Figure 6

The first unreleased draft of the product specifications which controls the tester design, is processed by the Quality Engineer using the specification program. The first set of test cases generated is voluminous. An example of a single specification illustrates the number of test cases which can be generated:

The flight power pulse shall rise to a voltage of 15 volts +/- 1% in less than 1 millisecond

volts	time	status
15	.75 ms	valid
15.15	.75 ms	valid high bound (hb)
14.85	.75 ms	valid low bound (lb)
0	.75 ms	invalid zero case (ilb)
14.80	.75 ms	invalid low bound minus (ilb-)
15.20	.75 ms	invalid high bound plus (ihb+)
15.0	.99 ms	valid high bound minus (vhb-)
15.0	1.00 ms	invalid high bound plus (ihb+)
15.0	0.00 ms	invalid clock failure

If the test specification were changed to read:

The flight power pulse shall rise to a voltage of 15 volts +/- 1% in less than 1 millisecond with a ripple of less than 10 mv +/- 1%

The resulting test cases would be:

volts	time	ripple	status
15	.75 ms	r< 10 mv	valid
15.15	.75 ms	r< 10 mv	valid high bound
14.85	.75 ms	r< 10 mv	valid low bound
0	.75 ms	r< 10 mv	invalid zero case
14.80	.75 ms	r< 10 mv	invalid low bound minus
15.20	.75 ms	r< 10 mv	invalid high bound plus
15.0	.99 ms	r< 10 mv	valid high bound minus
15.0	1.00 ms	r< 10 mv	invalid high bound plus
15.0	0.00 ms	r< 10 mv	invalid clock failure
15	.75 ms	r= 10.1mv	valid ripple, uhb
15	.75 ms	r= 9.9mv	valid ripple, llb
15	.75 ms	r= 10.2mv	invalid ripple uhb+
15	.75 ms	r= 15 mv	invalid ripple
15.15	.99 ms	r= 10.2mv	valid stress uhb
14.85	.99 ms	r= 10.2mv	valid stress llb

The above example illustrates the rapid growth of test cases if extensive testing were attempted. Based upon the example, there would be 1500 test cases per thousand specifications. To help reduce the test burden the test cases might be partitioned using equivalence class techniques. If the design of the software acceptance and exception handlers were known further test class reductions might be achieved. When the final set of test cases are agreed upon, they represent a set of "critical" test cases. Failure to execute a test in this set fails to test an entire class of tests or software.

The resulting test cases and known certification requirements are listed in a "Product Qualification Plan" (PQP), and formally released. The released PQP provides the tester engineer with the baseline for acceptance testing the test-suite will have to meet.

The PQP serves as a development baseline guide for design and testing of the hardware and for the software modules. The test cases are also placed in a test-file for use during final certification of the tester. Software development continues, independent of the hardware fabrication using the test cases developed by the automated specification program, the Static Path Analyses Tool, and the PIE box. Software modules are checked using S_PATm, and "White-Box" test cases developed. Figure 7 illustrates an "untestable" module while Figure 8 illustrates a "testable" module. In both cases the modules function and appear to satisfy system requirements. In figure 8, the test paths through the module can be determined using S_PATm, but because of the large number of variables required to

execute the test cases, they cannot be physically implemented. Modules of this size and complexity generally indicate poor specifications, poor programmer understanding of the specification, confusion, or all the above. The module may appear to work, but cannot be maintained. Future changes to the module may result in totally unexpected results. Test cases can be easily generated and applied to the program module shown in figure 8, because it is easily understood and testable, therefore it is easily maintained. Figure 9 shows the code logic diagram of figure 7 with four lines of code changed. While the module is still difficult to analyze, it is apparent that additional changes can be made which will allow the module to analyzed, tested, and maintained.

The ability to test the module and compare the results against released specifications enhances the confidence of the module and the maintainability of the program.

The test cases developed during the use of the S_PATm are placed into a test file with the test cases developed from the "T" tool or manually generated specification tests, for combined use during final tester evaluation. When the software is integrated with the hardware and system prove-in begins, the quality assurance engineer begins initial testing using the test cases developed and stored in a data-base file. In addition to using the S_PATm tool, the dynamic run time tool, TCAT, is used to evaluate the software. Using the test cases developed with "T" and S_PATm, the program is run in an emulator mode and the paths through the software are recorded as is the total number of possible test paths. The ratio of test paths executed to the total possible test paths constitutes a metric called coverage. The C1 metric is the coverage of a single test case, while the C2 metric is the summation of linearly independent

MENU
SUB: Menu (A)
Basic
Complexity 47

Upward Flows
Loop Exits
Plain Edges
Tue Mar 29 10: 17

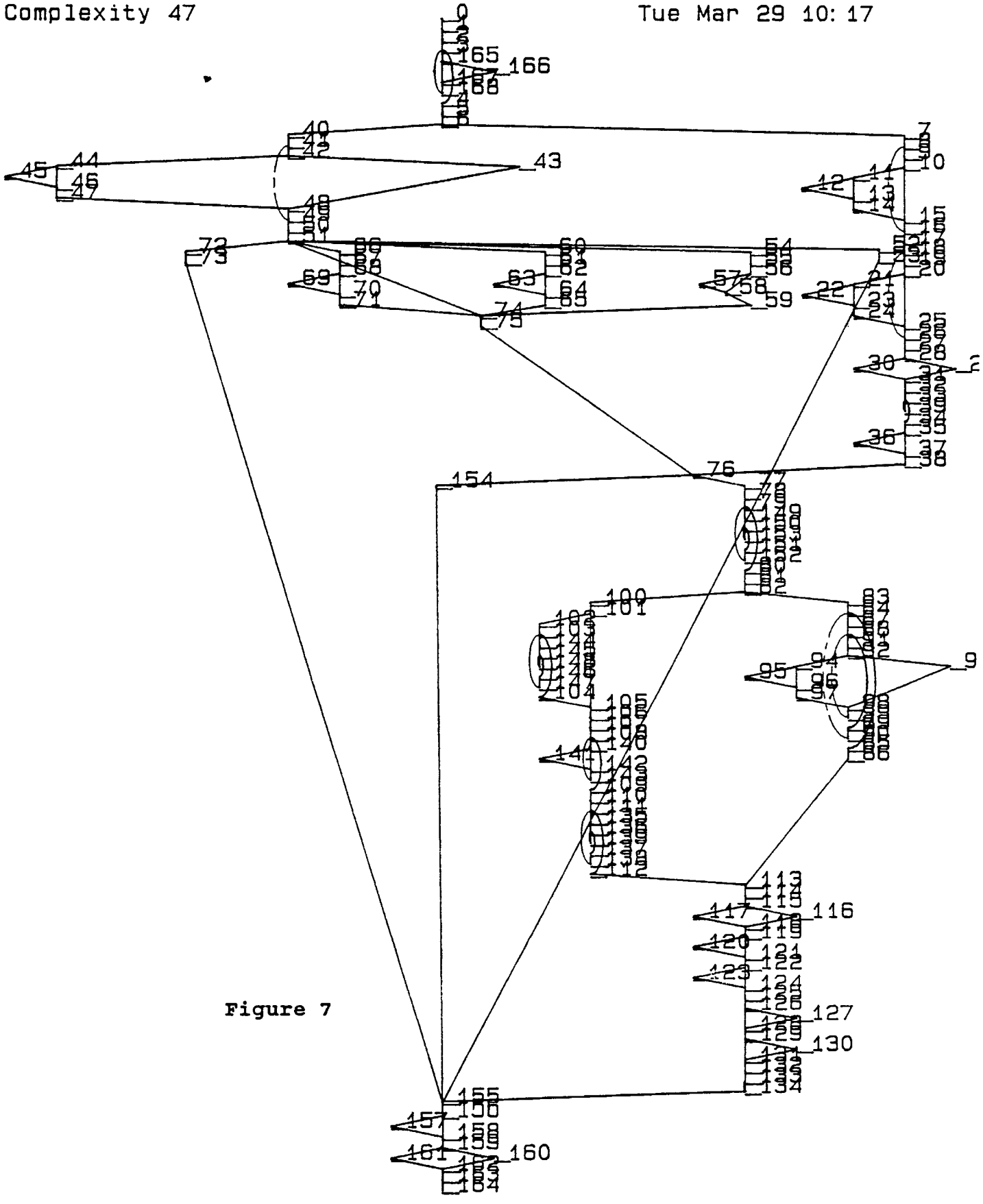


Figure 7

MENU
GOSUB_Print_menu (F)
Basic
Complexity 12

Upward Flows
Loop Exits
Plain Edges
Tue Mar 29 10:34

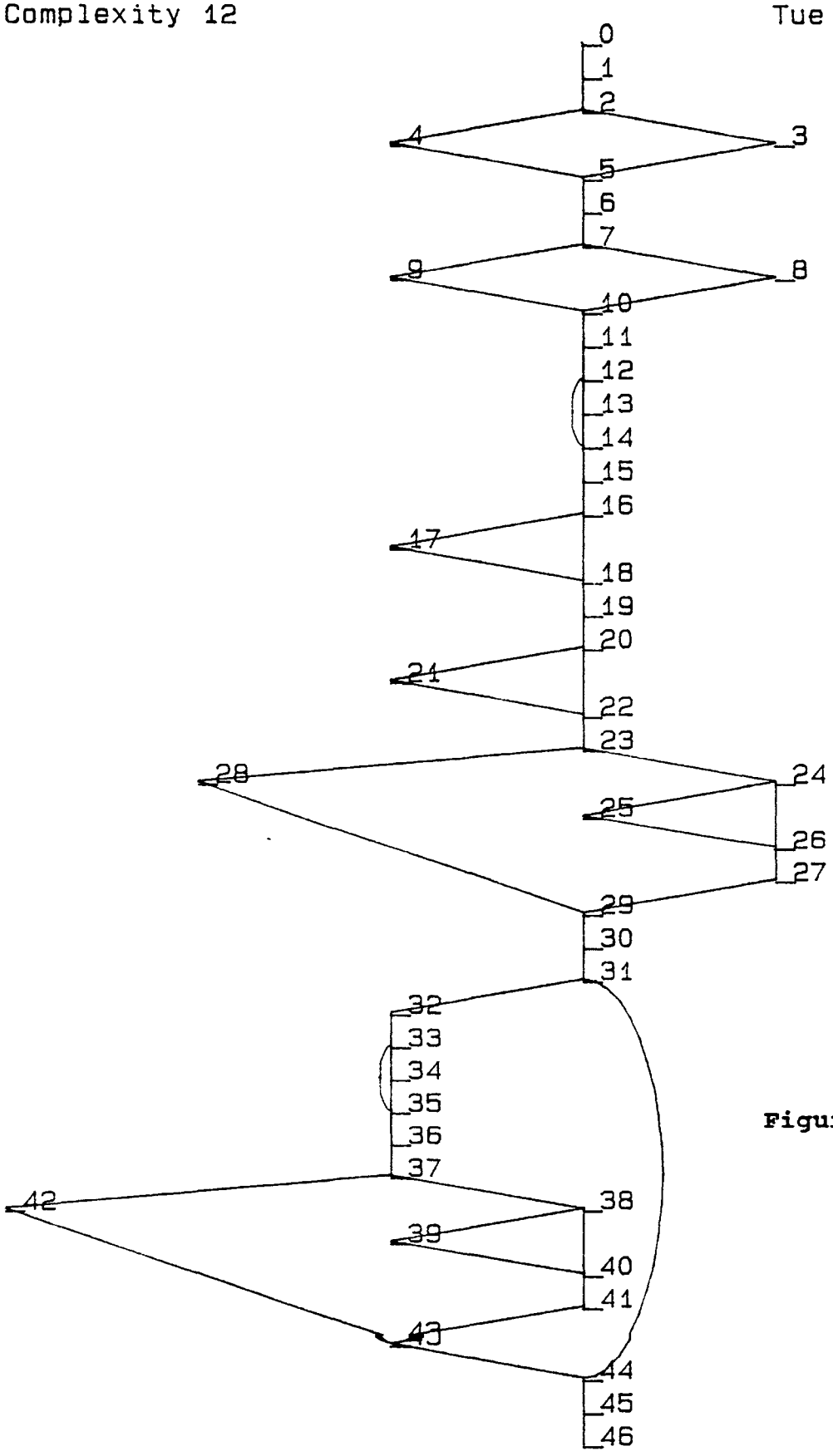


Figure 8

MENU
SUB: Menu (A)
Basic
Complexity 22

Upward Flows
Loop Exits
Plain Edges
Tue Mar 29 10:30

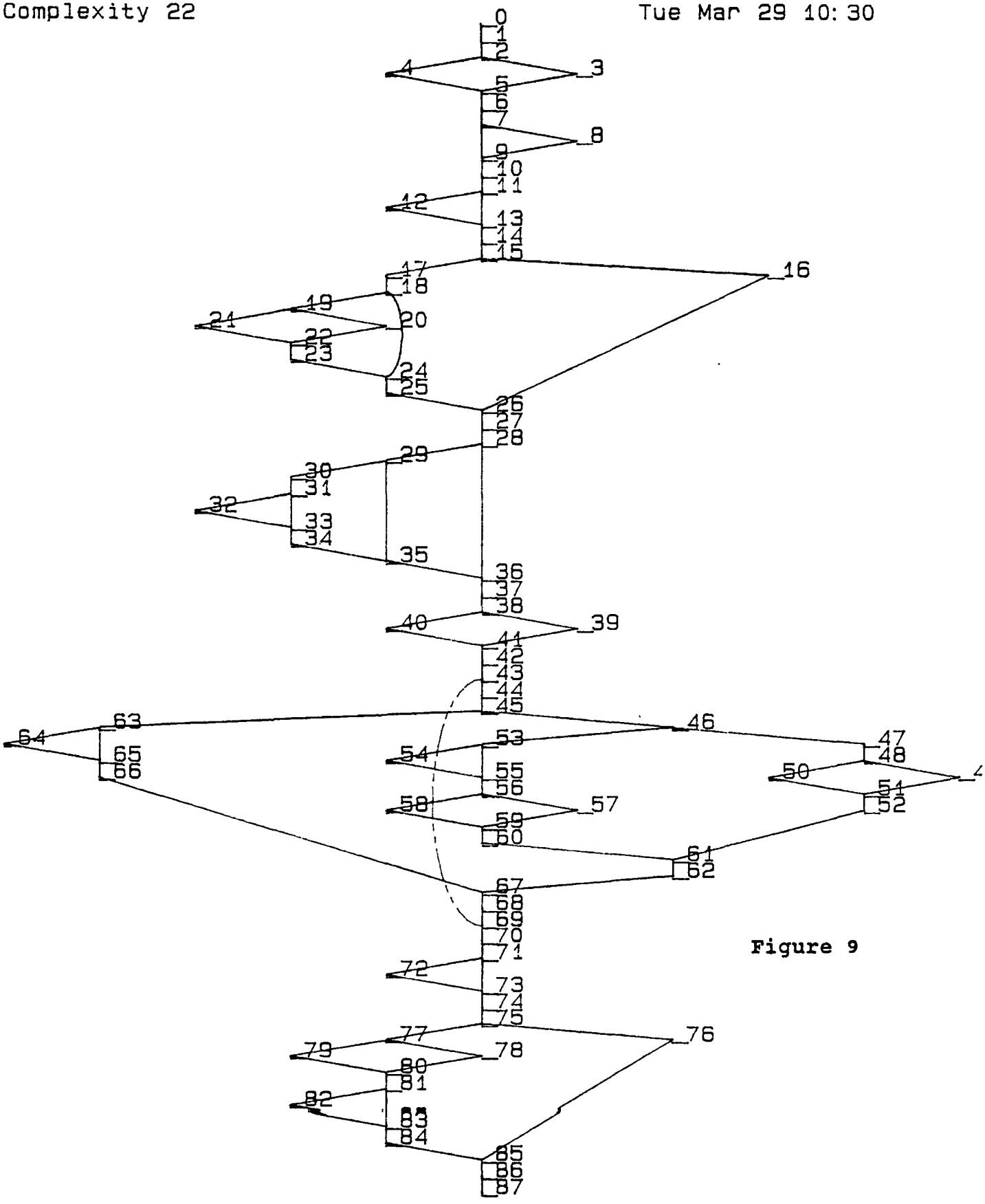


Figure 9

test paths covered during all testing compared to the total independent test paths. Software test paths not tested are identified and evaluated and a decision made as to whether the paths are additional capability, low risk or high risk, specification dependent, and additional tests are required by the PQT.

Upon completion of emulation testing, the Product Quality Team is able to determine whether all specifications have been implemented via software, how the software functions under exception testing, and what excess capability has been built into the tester that was not required by the specifications. Final certification testing of the test suite is accomplished by the PQT using evaluated software and the host system. Using the results of the quality assurance evaluation, the system is tested using released documentation and a second dynamic tool developed by SRA called S_TCAT. S_TCAT is a system-level version of the TCAT and provides coverage analysis of the system path coverage. The coverage metric, C3, provides path analysis of the released specifications developed with the automated specification tool. S_TCAT is presently under development for the Hewlett-Packard Advanced 5.0 Basic.

The verification of the test suite using the automated tools allows the PQT to determine the degree to which the tester meets required specifications. It further identifies excess system capability built into the test suite which was not required by the specifications. The degree of specification verification provides the basis for the "confidence-number" for the test suite. A test suite for which all specifications have been met and tested and no high risk excess capability, would be assigned a confidence-number of 100. The confidence number indicates the capability of the test suite to repeatedly perform all specification testing for the life of the tester. The use of the published test cases and requirements also provides the test equipment engineers with an "acceptance" criteria early in the development cycle.

When the test suite is modified, the original test cases used to verify the specifications and the modified test cases developed are used to recertify the test suite in an easily controlled and documented fashion.

QAIP REDESIGN

The automated tools presently being used are independent. Data transfer between the tools, and interpolation of data is accomplished by the operator. The QAIP tool is presently being redefined jointly by Sandia National Laboratory and the U.S. Army Quality Assurance Directorate to integrate the functions of a group of hardware and software tools. The use of a common data base and emulators provides a tool which can be used by operators with diverse background and needs.

While all functions of the QAIP program are linked via the data base, each capability is modular and can be deinstalled and replaced with no affect to the operation of the remaining modules. The design capabilities of QAIP are:

a. Executive Program: Provides a "user friendly" interface between the operator and the system modules. Printer, Plotter, and Screen types are installed. Data file source and destination are retained for use.

b. Word Processor: A link is established to the local word processor so each user can control the data and the reports generated.

c. Specification Tool: Allows user input of specifications and accuracy requirements and generates test cases required to ensure 100% specification test coverage. Test files generated are maintained in the Data Base Information File and printed as part of a preformatted "Testing Document".

d. Language Parser: A parser for any language which provides the mapping to determine the branch and node interconnections. All parsers conform to the DBIF interface definition. The parser allows parsing of a module, group of modules, or the system.

e. Static Analyzer: The static analyzer uses the information in the DBIF from the parser to:

    1. calculate the minimum number of test paths through the module(s) or system.
    2. generate the test cases required to transverse each test leg for the module and system level.
    3. record the starting and ending line number of each module.
    4. annotates the code listing to correspond to the branch-node graph.
    5. provides for printout to printer, plotter or screen of the branch-node graph.
    6. allows editing of a file and reparsing of the file. The reparsed file is given a new suffix.
    7. generate a "Call-Willcall" map.
    8. generate a cross-reference listing of names, labels, variables, subroutines, functions,
    9.

f. Dynamic Analyzer: The dynamic analyzer instruments the source code and establishes a local data file. The source code instrumentation is linked to the branch-node definition of the static analyzer. The Dynamic analyzer:

1. records the branches and nodes covered as each test case is run.
2. each test case records the individual coverage (C1) and the Cumulative coverage (C2).
3. provides for printing a listing of "branches hit" and "branches-not hit".
4. provides an output to the screen, plotter, or printer of the "branches hit" and /or the "branches not hit" overlaid on the static branch-node graph.
5.

Upon completion of all testing, the Dynamic Analyzer transfers the data files to the DBIF.

g. Statistics: The statistics module provides standard data reduction and evaluation capabilities. Data which is recorded during testing can be retained and analyzed off-line.

h. Graphics Program: The graphics program interfaces directly with the DBIF and provides plotter, printer, and screen outputs. The graphics program is driven by the DBIF data interface specifications and will present the same graphics regardless of the design language being used. This provides the user with a common visual interface.

i. User Tools: The executive program provides for up to six user programs to be linked to the selection menu. The user programs are not linked to the DBIF but can be run from the executive program.

j. Flow Chart: The flow chart program uses the parsed DBIF and produces a module level flow chart of the code. The program can be viewed and edited on the screen. Plotter and printer outputs in a "compressed no Alpha mode" or a "multipage detailed mode".

SUMMARY

The use of automated inspection tools can increase the confidence of the user that test suites are performing per their specifications and will fail in a predictable manner. Where test case generation, software inspection, and dynamic testing are being done manually, the use of automated tools results in a dramatic decrease of time and dollars with a corresponding increase in observed reliability. Where such activities are not being done, the implementation of an automated test and inspection process will increase both time and dollars. But the observed reliability will be decidedly improved.

The implementation of the automated tools must be tailored to each agency's needs. It is possible to "graft" tools from one user to another, but the tailoring is still required. There are no "SILVER BULLETS" in quality assurance.

REFERENCES

1. M. Fagan, "Advances in Software Inspections", IEEE Transactions on Software Engineering, Vol. SE-12, no. 7, July 1986.
2. E. Yourdon," Structured Walkthroughs", Prentice_Hall, Englewood Cliffs, N.J., 1979
3. R. Dunn, "Software Defect Removal", McGraw-Hill Book Company, New York, NY
4. J. Loeckx and K. Sieber, "The Foundations of Program Verification", Johm Wiley & Sons, 1984
5. B. Beizer, "Software System Testing and Quality Assurance", Van Nostrand Reinhold Company, NY 1984
6. B. Beizer, "Software Testing Techniques", Van Nostrand Reinhold Company, NY 1983
7. Chin-Kuei Cho, "An Introduction to Software Quality Control", John Wiley & Sons, NY 1980
8. M. Deutsch, "Software Verification and Validation", Prentice-Hall Inc. 1982
9. Musa, Iannino, Okumoto, "Software Reliability, Measurement, Prediction, Application", McGraw-Hill 1987
10. T. J. McCabe, "Structured Testing", IEEE Computer Society, IEEE Catalog No. EH0200-6, ISBN 0-8186-0452-2, 1983

DARL P. PATRICK

QUALITY ASSURANCE DIVISION 7252

SANDIA NATIONAL LABORATORIES
ALBUQUERQUE, NEW MEXICO 87185
(505) 844-8745

88F7000.06

Sandia National Laboratories

SANDIA NATIONAL LABORATORIES
QUALITY ASSURANCE

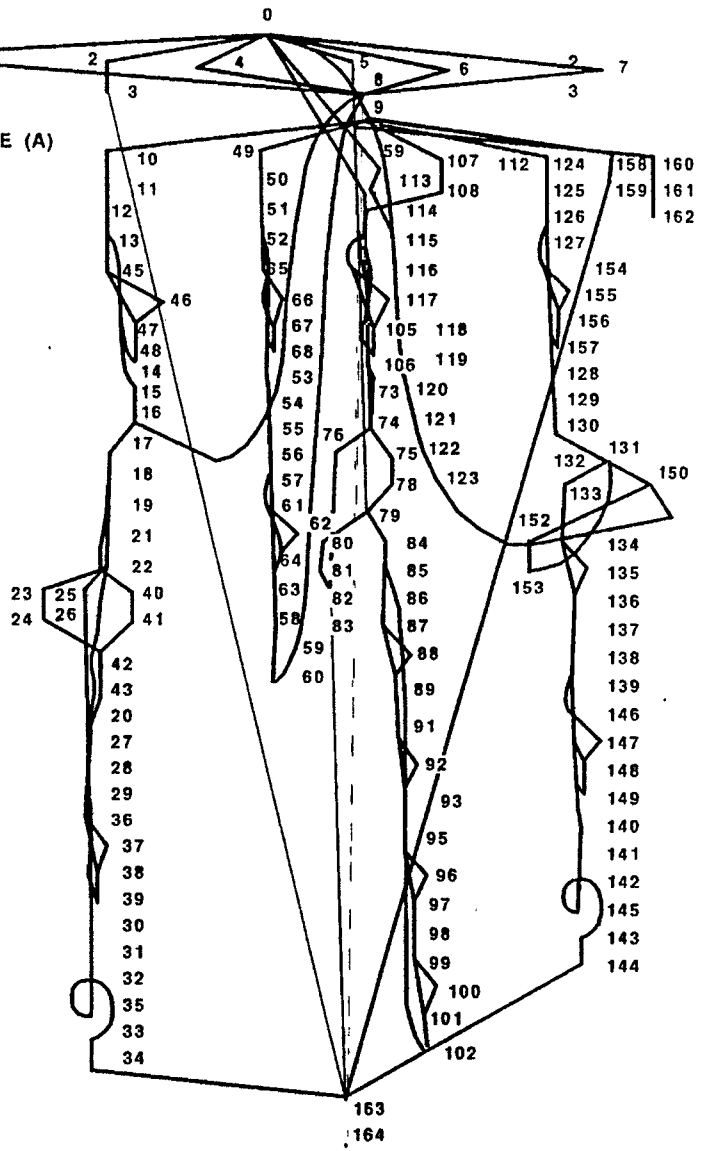
88G7000.59

DATA MODULE

```
CALL INST"CNTR,INIT"
CALL INST"CNTR,HELP"
CALL INST"CNTR,SET,

125     FOR I = 1 TO 200
126             SELECT T__des (I)
127             CASE "TC:"
128     !NIW TEST CODE
129             PRINT
130             PRINT "TEST CODE:
131             CASE "TA:"
132             PRINT
133             PRINT "ADAPTER:"
134             CASE ""
135     !BLANK LINE
136             CASE ELSE
137     !TEST ID
138                 PRINT
139     TAB (4) ; T__des (I) ;TA
140         END SELECT
141     !
142             NEXT I
143     PRINTER IS CRT
144     INPUT "ENTER PASS "
145         IF T< >"PASS" THEN
146         No __ Access
147         OUTPUT 2;K
148         ELSE
149             ASSIGN  @ Ascii
150         END IF
151         ON ERROR GOTO File
152         !
153         !
154     File __ end:
155         ASSIGN  @ Ascii To
156         ON ERROR GOTO Skip
157         PURGE "LIMITS"
158     Skip:
159         OFF ERROR
160         CREATE BDAT "LIMIT
161         !
162     NEW LIMITS FILE
163         ASSIGN  @ BDAT TO
```

augeishamg.bas
SUB __ Geisha __ mgr.EXCE (A)
Basic
Complexity 50



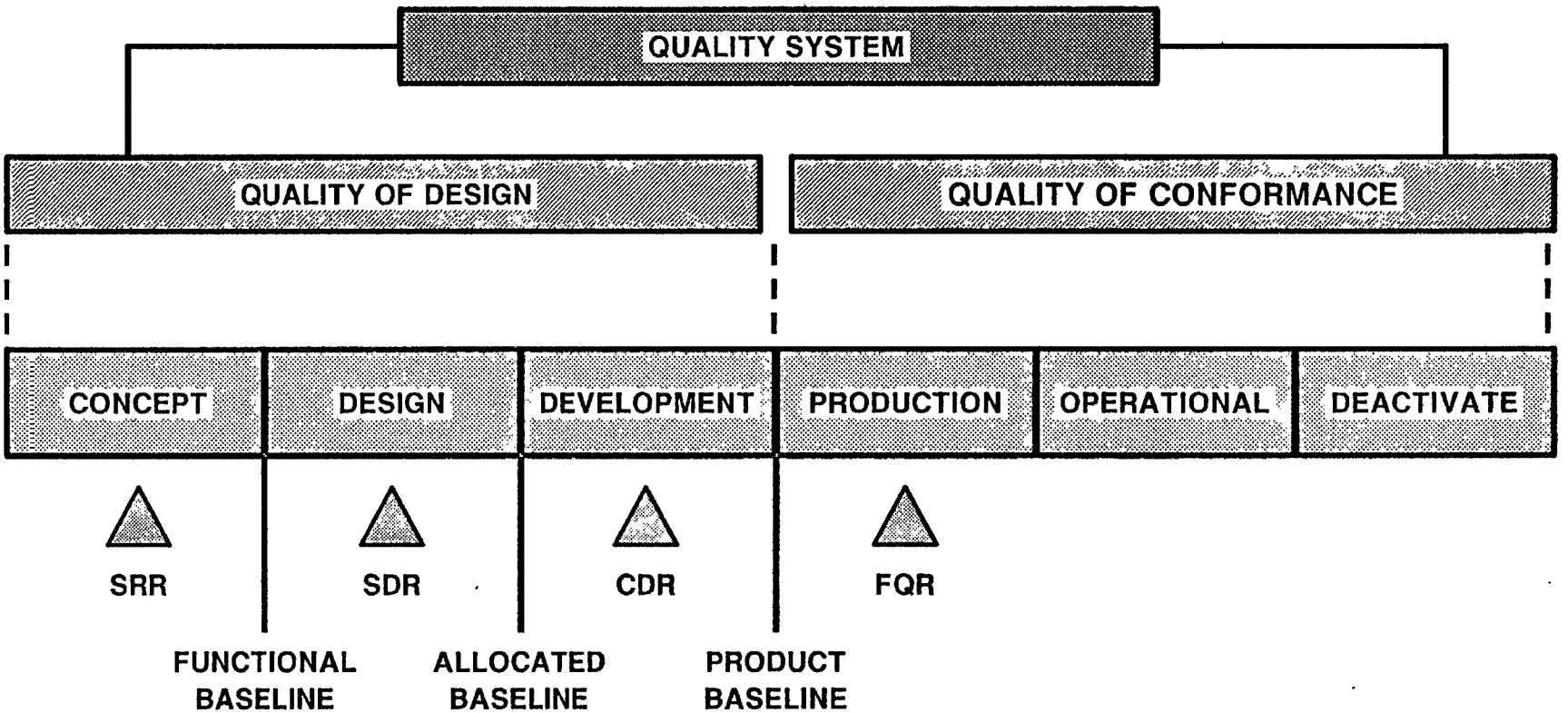
Upward flows
Loop exists
Plain Edges
Sun Jan 11 10:32

Sandia National Laboratories

TYPES OF CERTIFICATION PROCESSES

- EQ OF AUTOMATED TEST EQUIPMENT
- USE OF MINIMUM LEVEL LANGUAGE (BASIC, FORTRAN, ETC.)
  - 50,000 LINES OF CODE
- INSPECTIONS OF AUTOMATED EQUIPMENT
  - PT / TE / DT / COMMERCIAL
  - 50,000 LOC
  - MEDIUM OR HIGH LEVEL LANGUAGES (BASIC, PASCAL)
- INSPECTIONS OF SANDIA PRODUCT CODE
  - 4000 - 100000 LOC
  - ASSEMBLY - UP

QUALITY ASSURANCE



SRR ... SYSTEMS REQUIREMENTS REVIEW

SDR ... SYSTEMS DESIGN REVIEW

CDR ... CRITICAL DESIGN REVIEW

FQR ... FINAL QUALIFICATION REVIEW

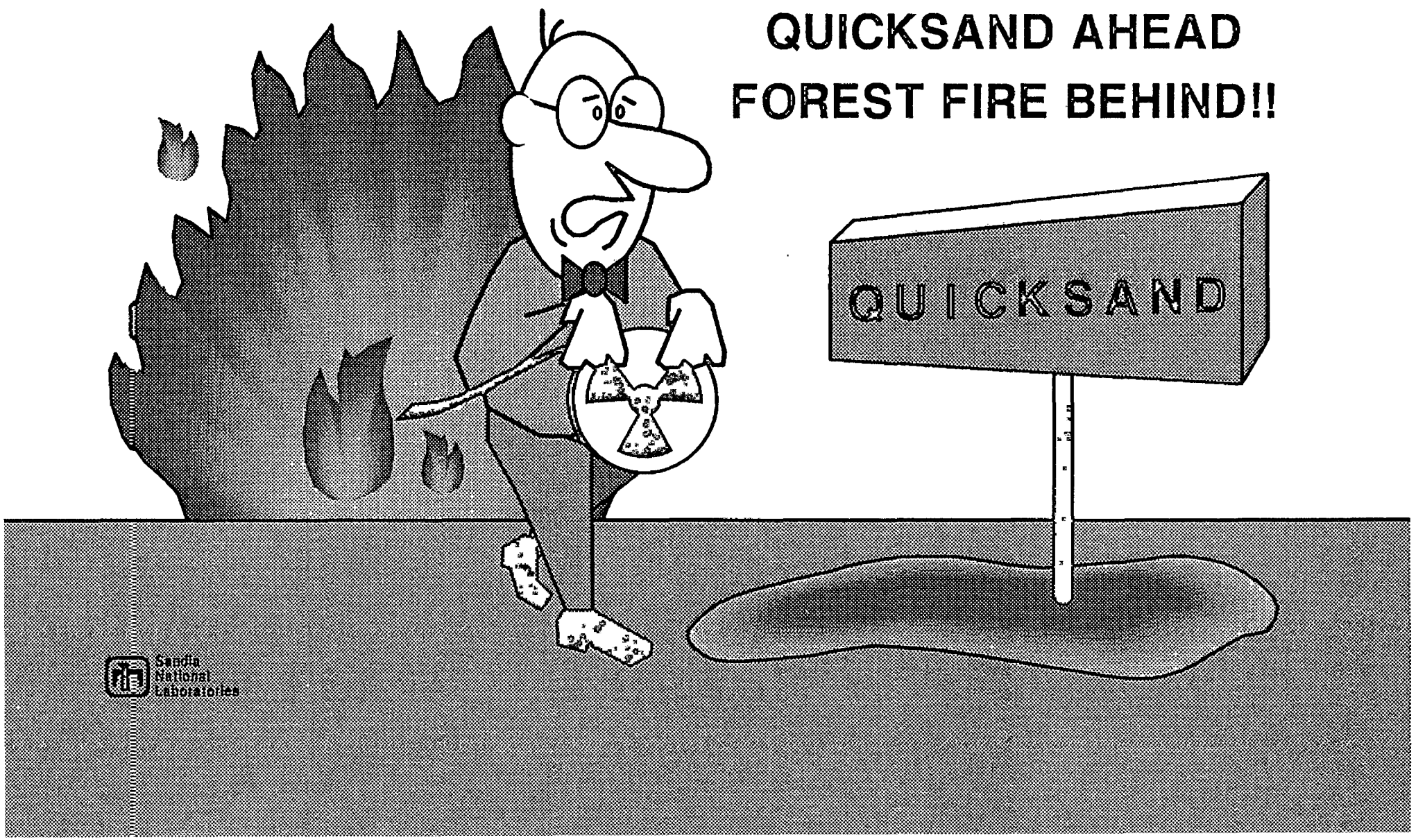
DP P DIV 7252

88F7000.12

Sandia National Laboratories

QUALITY ASSURANCE

88G7000.60



WHAT CONSTITUTES A HOSTILE ENVIRONMENT?

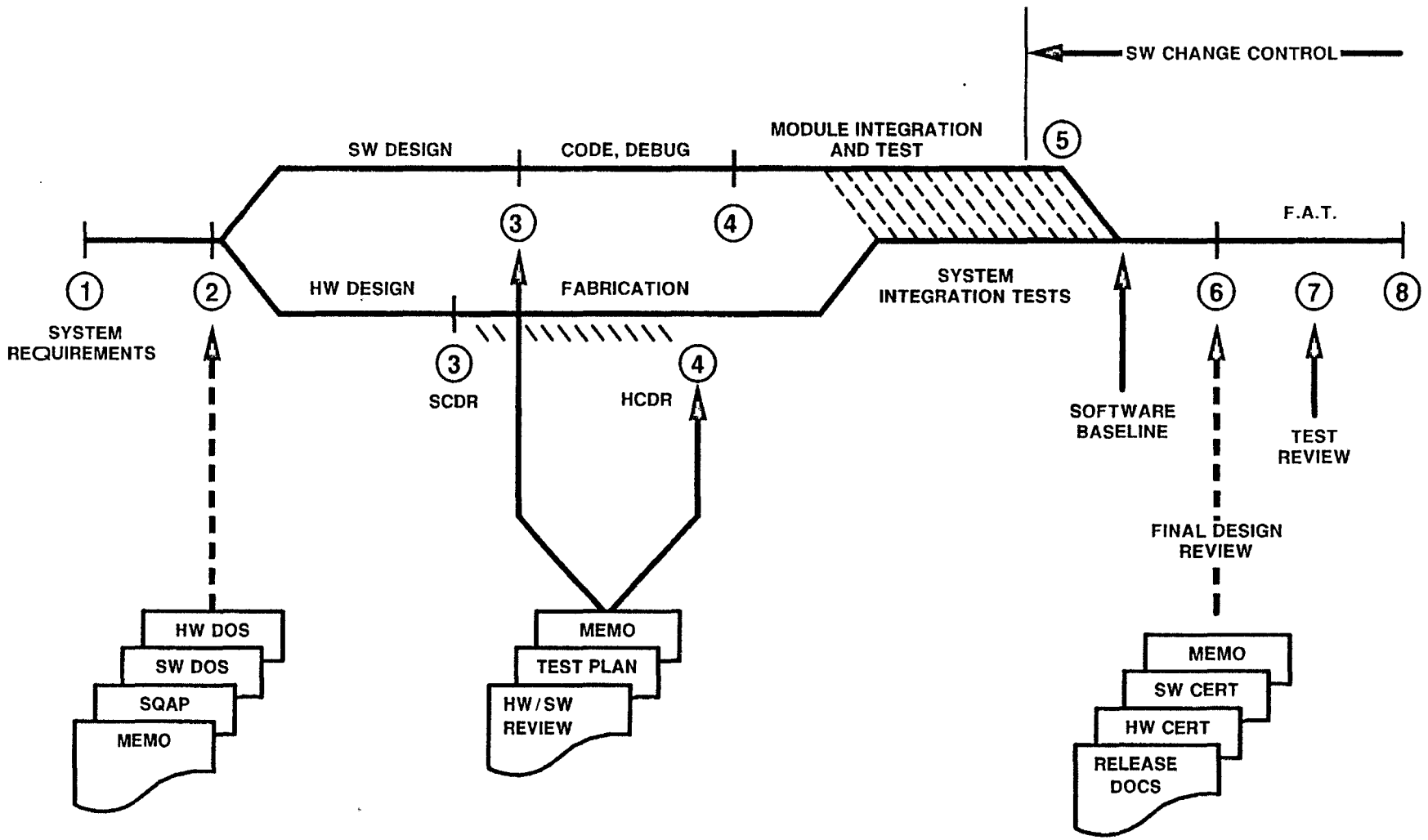
- INABILITY TO CONDUCT FULL HW / SW TESTING
- INABILITY TO DESIGN FROM FIRM SPECIFICATION
- LACK OF DEVELOPMENT TEST DATA
- LACK OF COMPREHENSIVE KNOWLEDGE OF SYNTAX OF LANGUAGE BEING USED
- LACK OF UNDERSTANDING OF THE ROLE RELATIONSHIPS OF HARDWARE AND SOFTWARE
- LACK OF SOFTWARE CHANGE CONTROL
- CONCEPT THAT QUALITY IS GREAT IF IT:
  - A. DOESN'T TAKE ANY EXTRA TIME
  - B. DOESN'T COST ANYTHING
  - C. DOESN'T DELAY PRODUCTION
- LACK OF FIRM REQUIREMENTS FOR HW OR SW

SOME MISCONCEPTIONS CONCERNING TESTING:

- TESTING IS THE PROCESS OF DEMONSTRATING THAT ERRORS ARE NOT PRESENT
- THE PURPOSE OF TESTING IS TO SHOW THAT A PROGRAM PERFORMS ITS INTENDED FUNCTION(S) CORRECTLY
- TESTING IS THE PROCESS OF ESTABLISHING CONFIDENCE THAT A PROGRAM DOES WHAT IT IS SUPPOSED TO DO

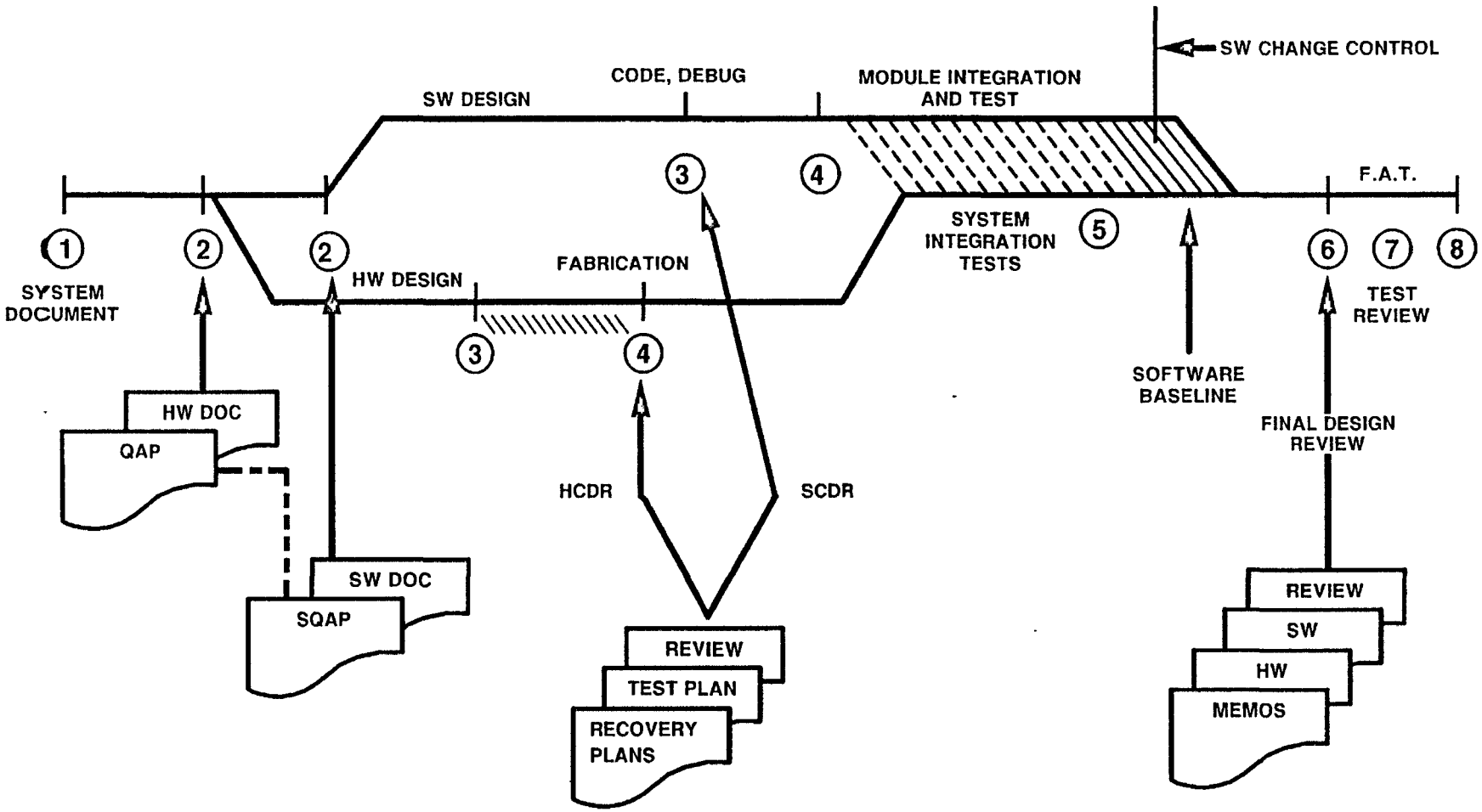
RESULT OF APPLICATION OF THESE MISCONCEPTIONS:

- IF THE GOAL IS TO SHOW NO ERRORS, THAT IS WHAT THE TEST CASES WILL DO
- IF THE GOAL IS TO SHOW OFF THE PROGRAM'S STATED FUNCTIONS, THE TEST CASES WILL SHOW LITTLE ELSE
- IF THE GOAL IS TO PROVIDE CONFIDENCE THAT A PROGRAM IS PERFORMING CORRECTLY, THE LAST THING A TEST CASE WILL DO IS FIND ERRORS



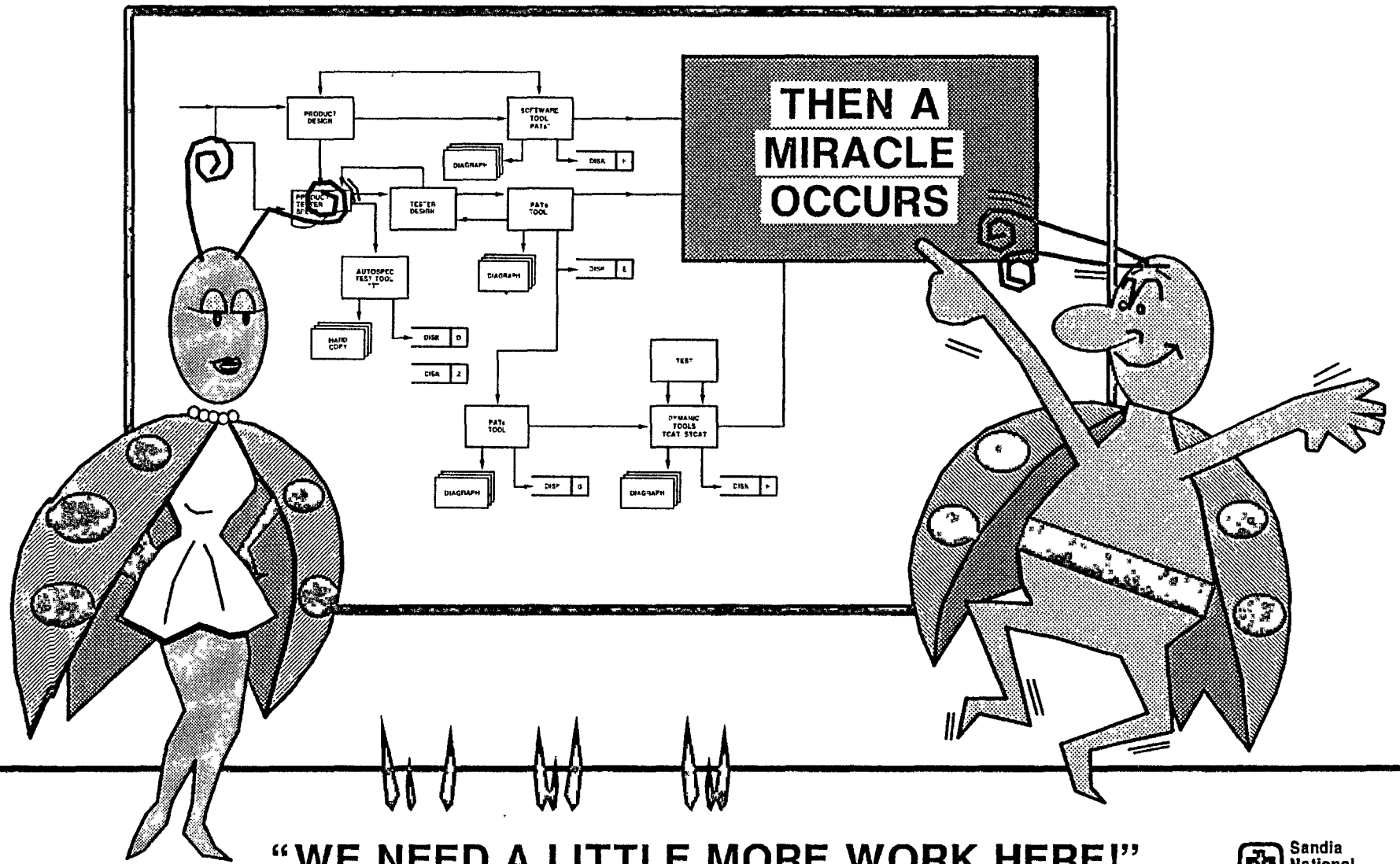
88G7000.18

Sandia National Laboratories



88G7000.28

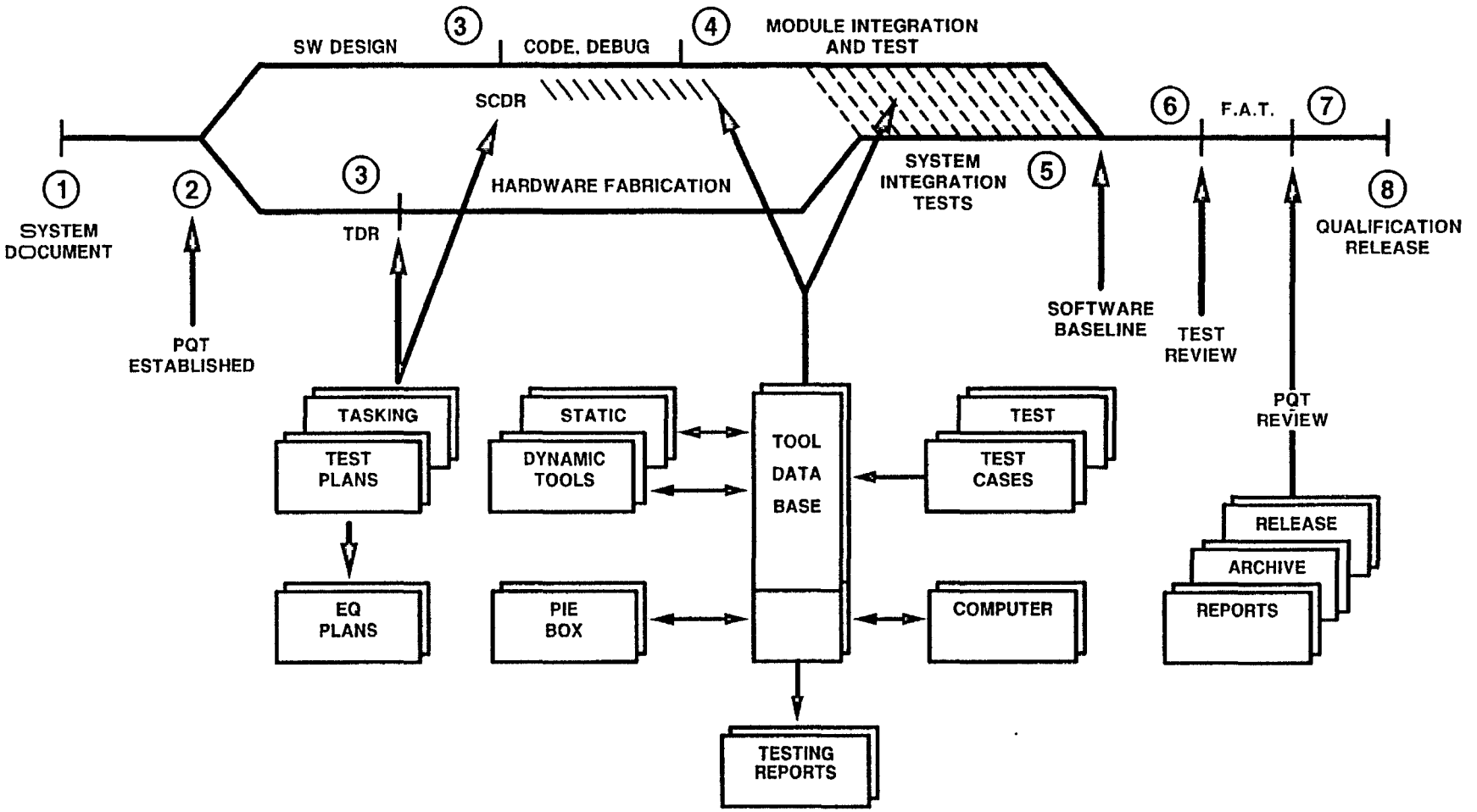
Sandia National Laboratories



"WE NEED A LITTLE MORE WORK HERE!"

88G7000.58

Sandia National Laboratories



88G7000.20

Sandia National Laboratories

SPECIFICATION TEST GENERATION TOOL

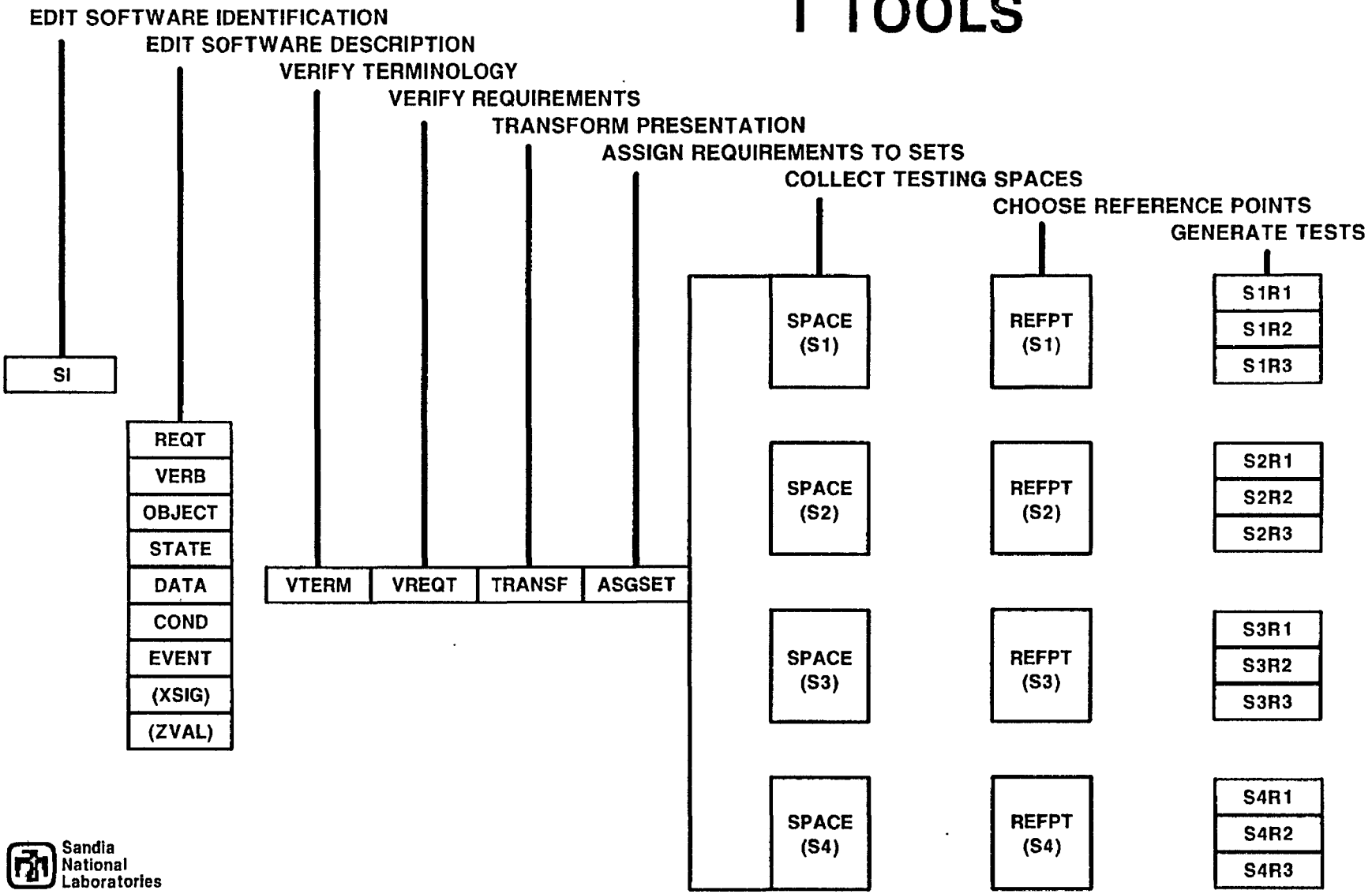
("T")

A TOOL WHICH CONTROLS OPERATOR INPUTS OF SPECIFICATIONS AND GENERATES A SET OF LINERLY INDEPENDENT TEST WHICH ENSURE ALL SPECIFICATIONS ARE COVERED

Sandia National Laboratories

T TOOLS

88G7000.30



Sandia National Laboratories

T Unit: alltimer ver 1 88G7000.32
Report: Software Identification

Testunit is not protected Rev 21:59 08-26-87

Description

Sample testunit. This FORTRAN subroutine will produce an updated data and time, given reference date and time and some number of minutes (positive or negative) by which to shift the reference. It is used for the calculation of radiation exposure in the event of a leak from a nuclear power plant reactor. It should probably be tested rather "thoroughly."

References

Code	Document Names
MS	Module Specification for XXXXXXXX, version 1.2a filed in software library under project NR-MON.

T Unit: alltimer ver 1
Report: Dataitem Listings

28__day__counter - day28 Rev 13:50 06-29-87

Type	integer	Grp -
Unit	day	
Mn/Mx/Rs	1/28/1	
Desc	valid counter for 28-day month	
Tels	v nm 1 @ 7	
	v lb @ 1	
	v hb @ 28	

28__day__month - mon28 Rev 13:50 06-06-87

Type	choice	Grp -
Desc	a month having 28 days (normal year, feb)	
Tels	v nm 1 @ February	

29__day__counter - day29 Rev 13:50 06-29-87

Type	integer	Grp -
Unit	day	
Mn/Mx/Rs	1/29/1	
Desc	valid counter for 29-day month	
Tels	v nm 1 @ 7	
	v lb @ 1	
	v hb @ 29	

29__day__month - mon29 Rev 13:50 06-29-87

Type	choice	Grp -
Desc	a month having 29 days (leap year, feb)	
Tels	v nm 1 @ February	

T Unit: alltimer ver 1
Report: Dataitem Listings

30__day__counter			- day30	Rev 13:50 06-29-87
	Type	integer		Grp -
	Unit	day		
	Mn/Mx/Rs	1/30/1		
	Desc	valid counter for 30-day month		
	Tels	v nm 1	@ 7	
		v lb	@ 1	
		v hb	@ 30	
30__day__month			- mon30	Rev 13:50 06-29-87
	Type	choice		Grp -
	Desc	a month having 30 days		
	Tels	v nm 1	@ April	
		v nm 2	@ June	
		v nm 3	@ September	
		v nm 4	@ November	
31__day__counter			- day31	Rev 13:50 06-29-87
	Type	integer		Grp -
	Unit	day		
	Mn/Mx/Rs	1/31/1		
	Desc	valid counter for 31-day month		
	Tels	v nm 1	@ 7	

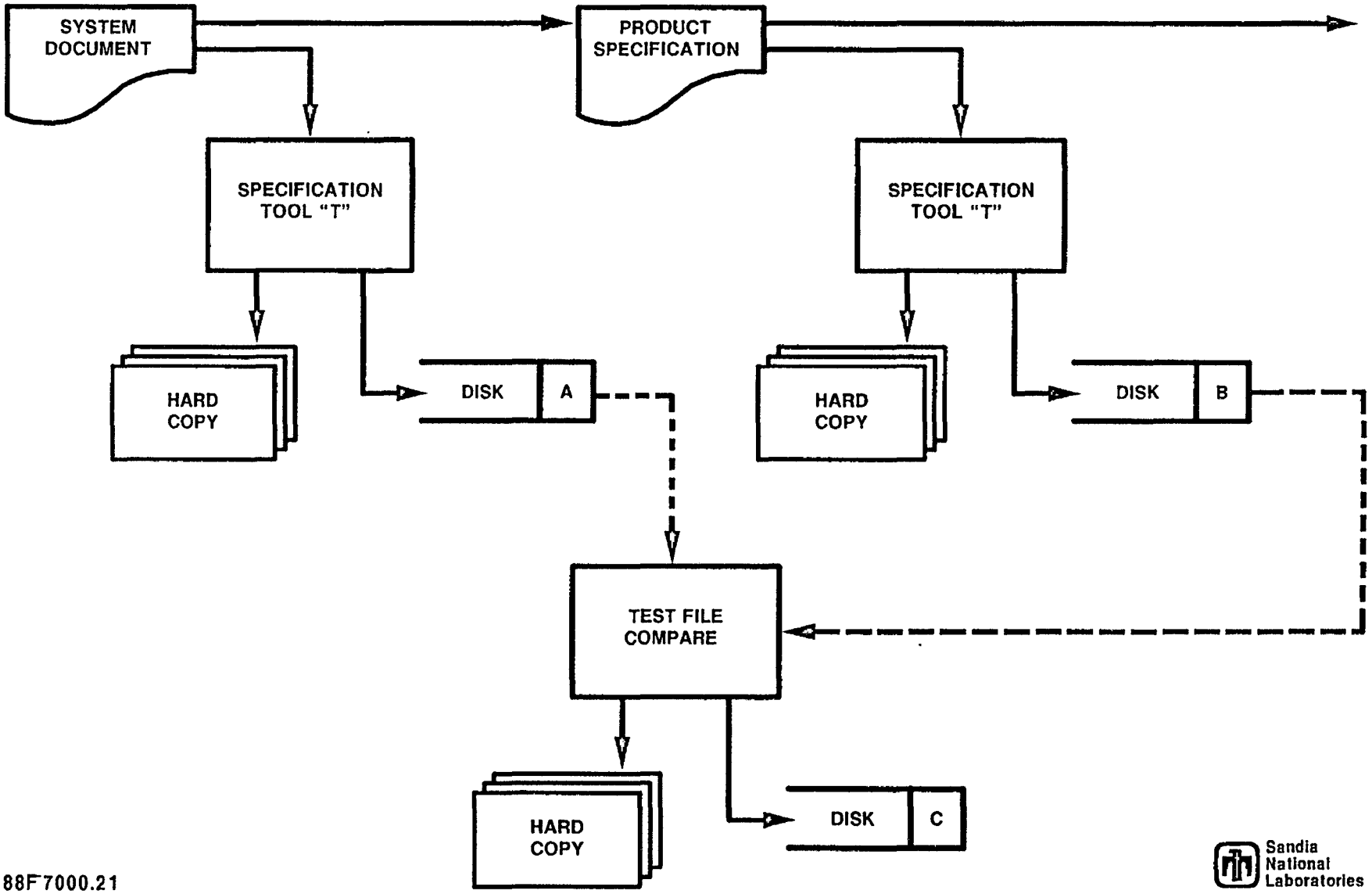
T Unit: alltimer ver 1
Report: Dataitem Listings

88G7000.29

```
                    v tr 1      @ 0
                    v tr 2      @ - 1
                    v tr 3      @ 1
                    v tr 4      @ 1439
                    v tr 5      @ 1440
                    v tr 6      @ 1439
                    v lb        @ - 527040
                    v lb +      @ - 527039
                    v hb-       @ 527039
                    v hb        @ 527040
                    i an 1      @ null
                    i an 2      @ ss
                    i lb -      @ - 527041
                    i hb +      @ 527041

user__entry                                  - uniput      Rev 16:31 06-29-87
          Type   sequence                                  Grp -
          Desc   user entry: date time shift
          Tols   v nm 1  calendar__date
                         + time__of__day
                         + time__to__shift__reference__date/time
```

Sandia National Laboratories



88F7000.21

Sandia National Laboratories

SOFTWARE PATH ANALYSIS TOOLS (PATS)

A SET OF SOFTWARE TOOLS WHICH WILL AUTOMATICALLY "READ" THE HOST SOFTWARE AND DETERMINE METRIC MEASUREMENTS

Sandia National Laboratories

I. CAPABILITIES OF PAT_S

- ABLE TO READ PRODUCTION CODE OFF A PRODUCTION DISK AND ANALYZE THE CODE FOR:
  - SYNTAX ERRORS
  - DECISION NODES
  - LOOPS
  - BRANCHES (LONG AND SHORT)
  - INTERRUPTS
  - COMPLEXITY OF EACH MODULE
  - CRITICAL TEST PATH
- GRAPHS THE LOGIC OF THE MODULE
- PRINTS AND GRAPHS ALL LINERLY INDEPENDENT TEST PATHS OF THE MODULE
- ALLOWS EDITING OF A MODULE

II. INCREASES INSPECTION THROUGHOUT FROM 1000 LOC/WEEK TO = 8000 LOC/WEEK

ANNOTATED SOURCE LISTING

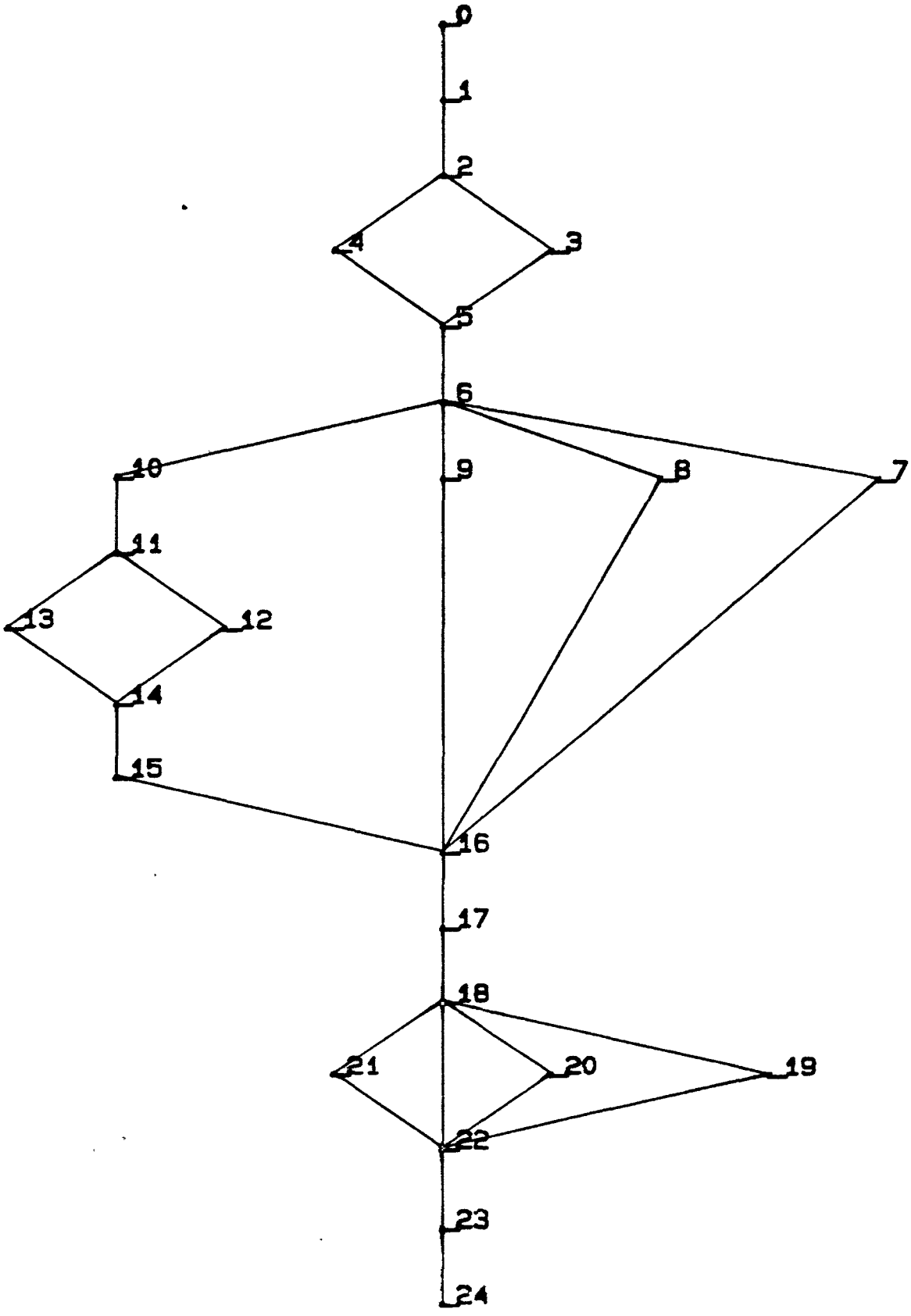
File: CSELECT .BAS
Language: H.P. BASIC

DATE/TIME: WED FEB 11 18:59

MODULE LETTER	MODULE NAME	PREDICTIVE TEST PATHS	STARTING LINE	NUMBER OF LINES
A	SUB _ Initialize	1	62	17
B	SUB _ Rack _ turn _ on	1	82	60
C	SUB _ Graphic _ message	2	145	20
D	SUB _ Chan _ id	1	168	24
E	SUB _ Multiplexchan	1	195	3
F	SUB _ Serialnumber	8	201	230
G	SUB _ Titlepage	4	440	26
H	SUB _ Voltmeter	1	470	4
I	SUB _ Voltmeter _ id	1	477	8
J	SUB _ Drawertest	11	488	155
K	SUB _ Passfail	3	646	14
L	SUB _ Dataheader	1	663	7
M	SUB _ Openlooptest	10	673	56
N	SUB _ inittwopoint	10	733	66
O	SUB _ Repeatability	18	803	164
P	SUB _ Fourpointtumble	14	971	81
Q	SUB _ Pfcolor	2	1056	9
R	SUB _ Push	15	1068	78
S	SUB _ Voltagesource	1	1150	3
T	SUB _ Pushtest	12	1156	71
U	SUB _ Minmax	4	1231	10
V	SUB _ Graph	2	1244	6
W	SUB _ Paplot	3	1253	46
X	SUB _ Labels	2	1302	18
Y	SUB _ Test	1	1323	9
Z	SUB _ Biasplot	3	1335	36
a	SUB _ Alignplot	3	1374	36
b	SUB _ Springplot	6	1413	49
c	SUB _ Curvefit	4	1465	30
d	MAIN	7	1	58

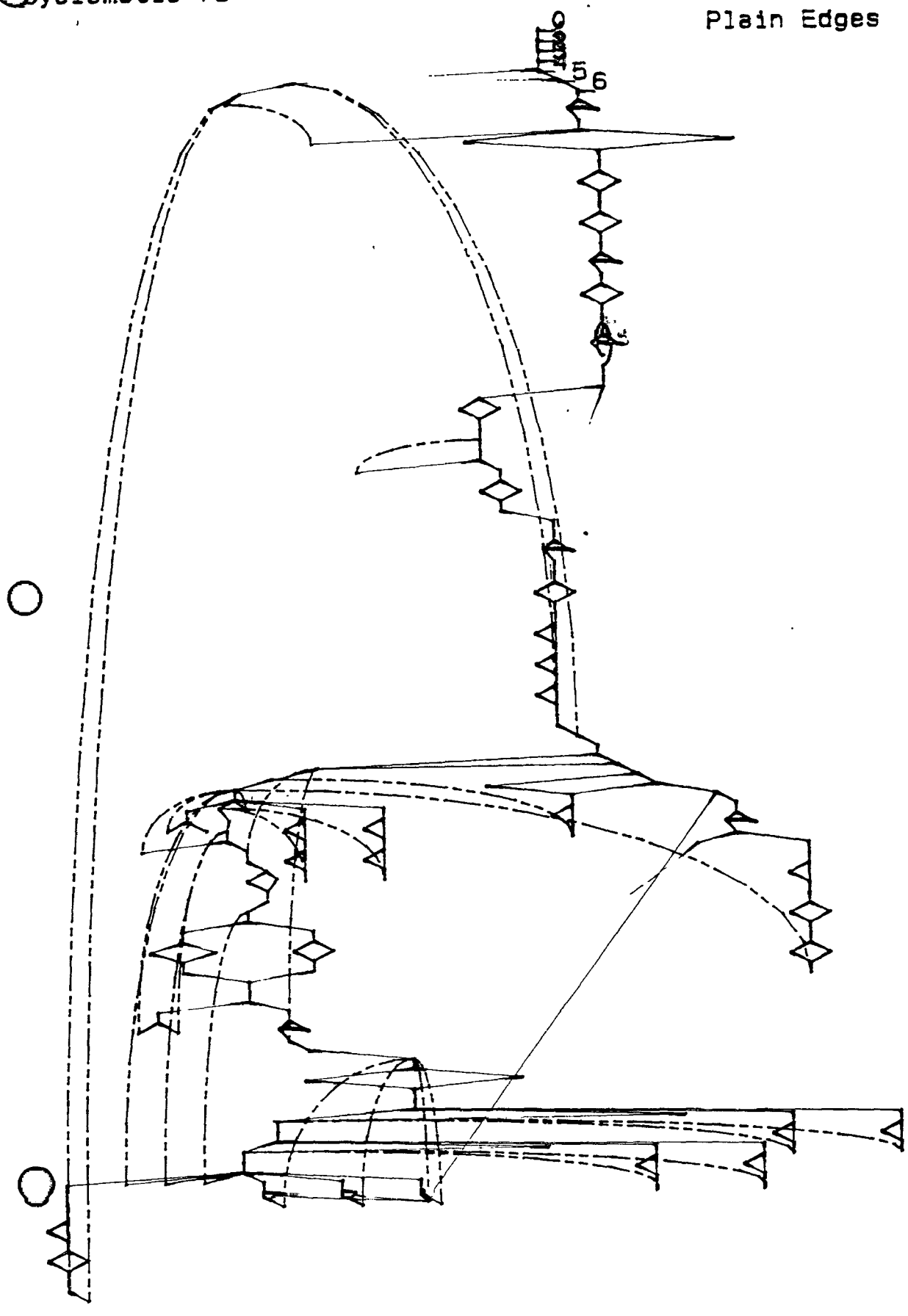
MENUSMAL.BAS
Menu (A)
Cyclomatic 9

Wed Apr 13 14:27
Upward Flows
Loop Exits
Plain Edges



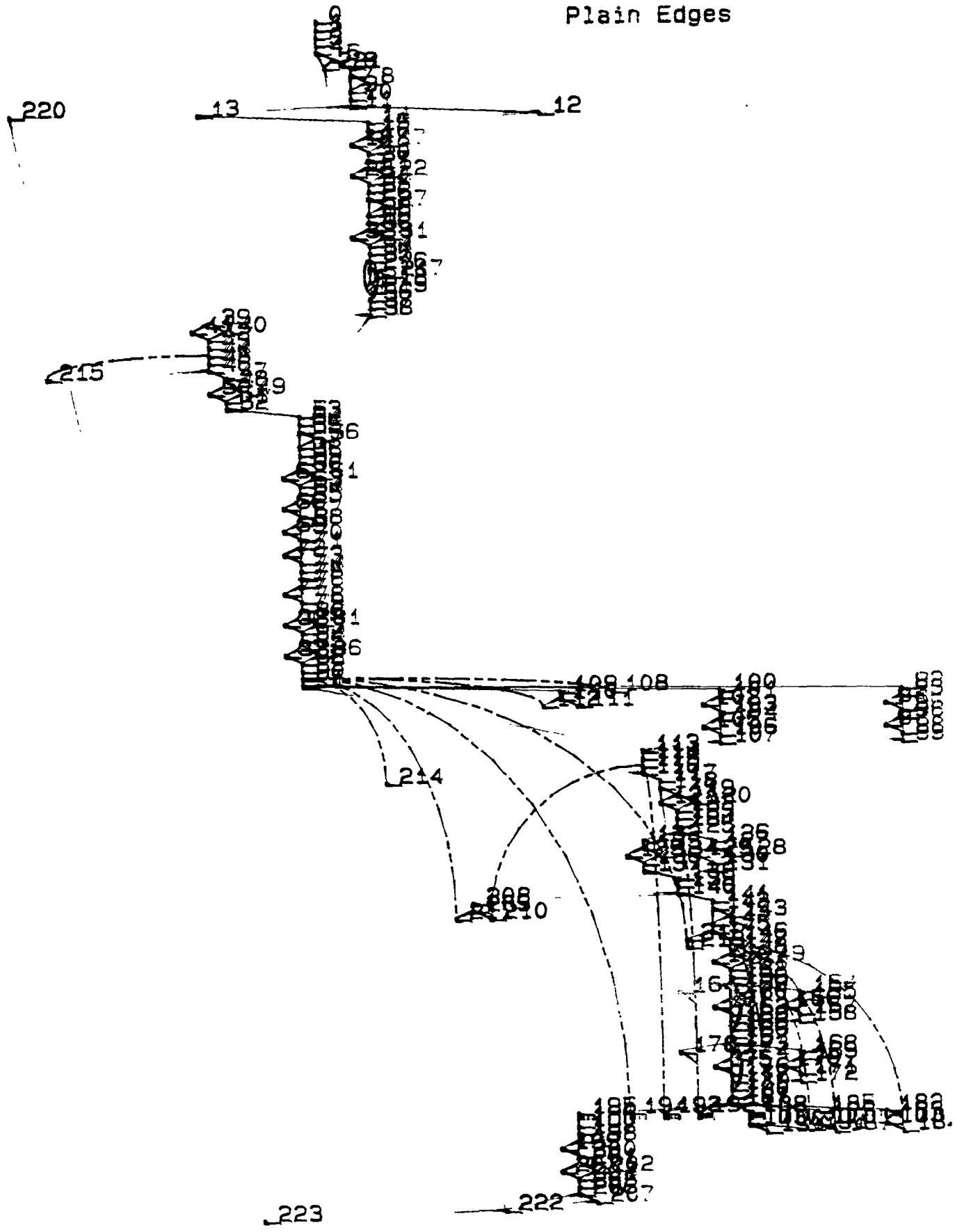
^NTWO.BAS
MAIN (W)
Cyclomatic 75

Tue Apr 12 05:08
Upward Flows
Loop Exits
Plain Edges



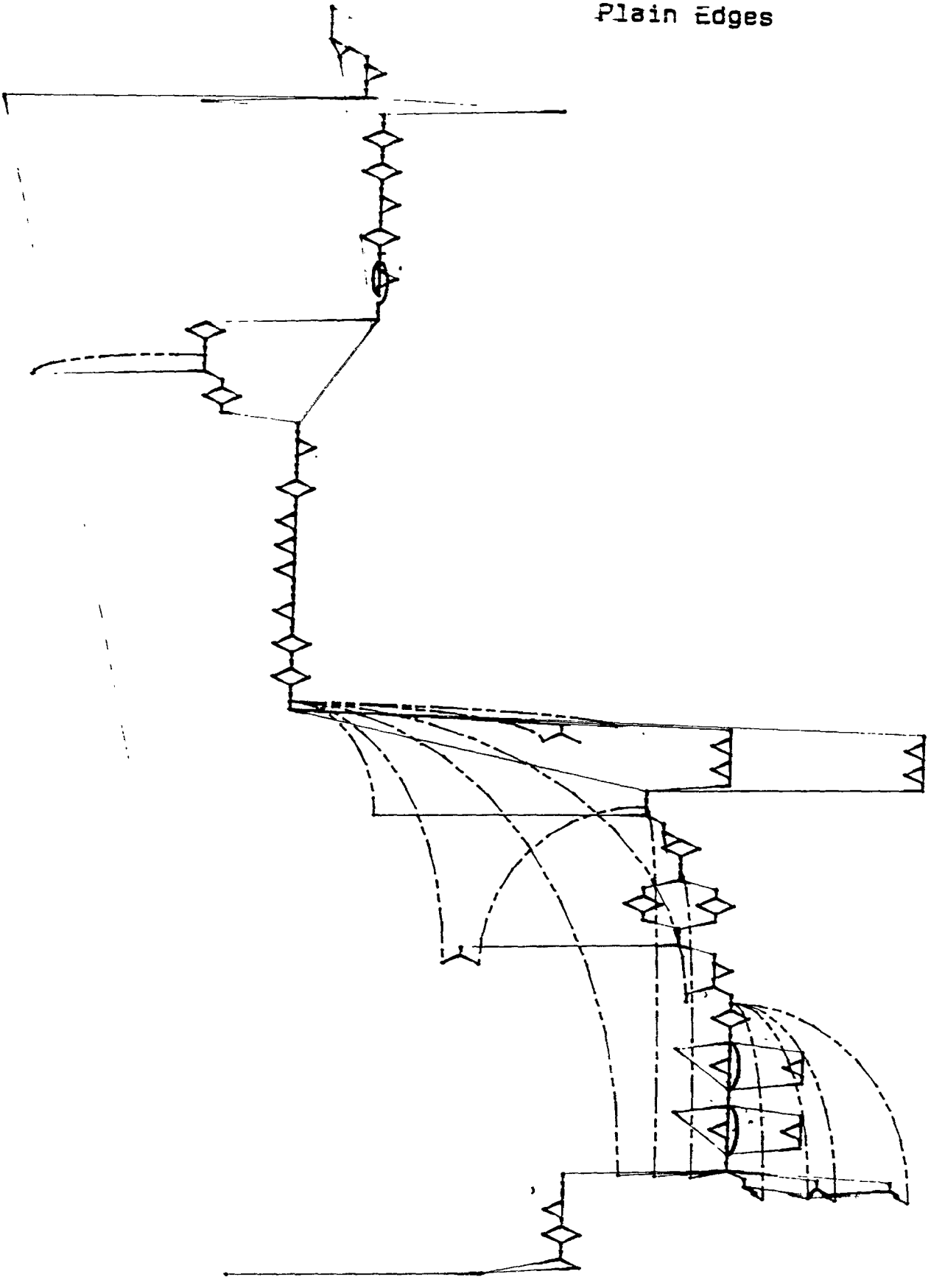
MAIN1.BAS
MAIN (V)
Cyclomatic 65

Tue Apr 12 05:40
Upward Flows
Loop Exits
Plain Edges



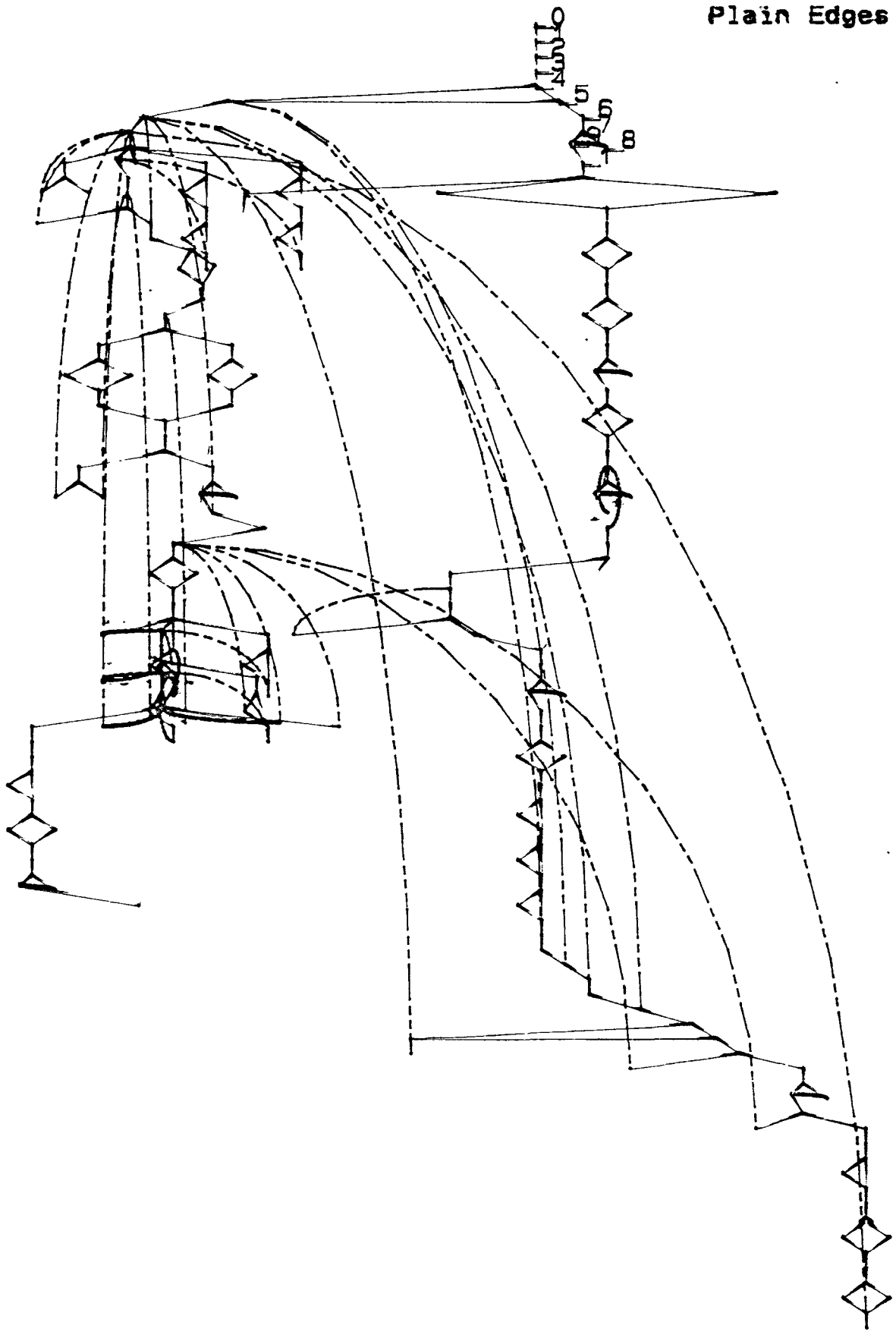
MAIN1.BAS
MAIN (V)
Cyclomatic 65

Tue Apr 12 05:51
Upward Flows
Loop Exits
Plain Edges



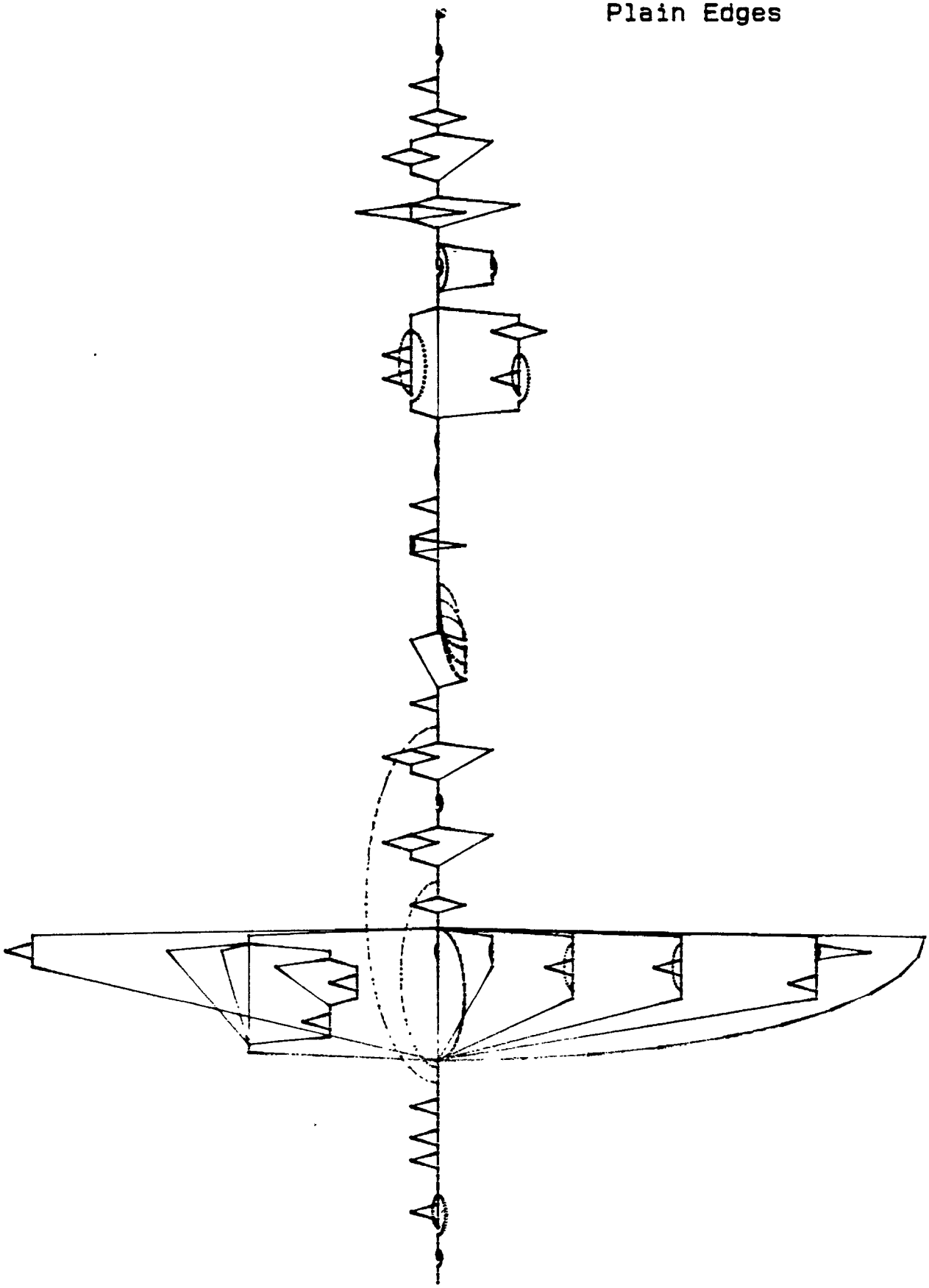
:NONE.BAS
MAIN (W)
Cyclomatic 69

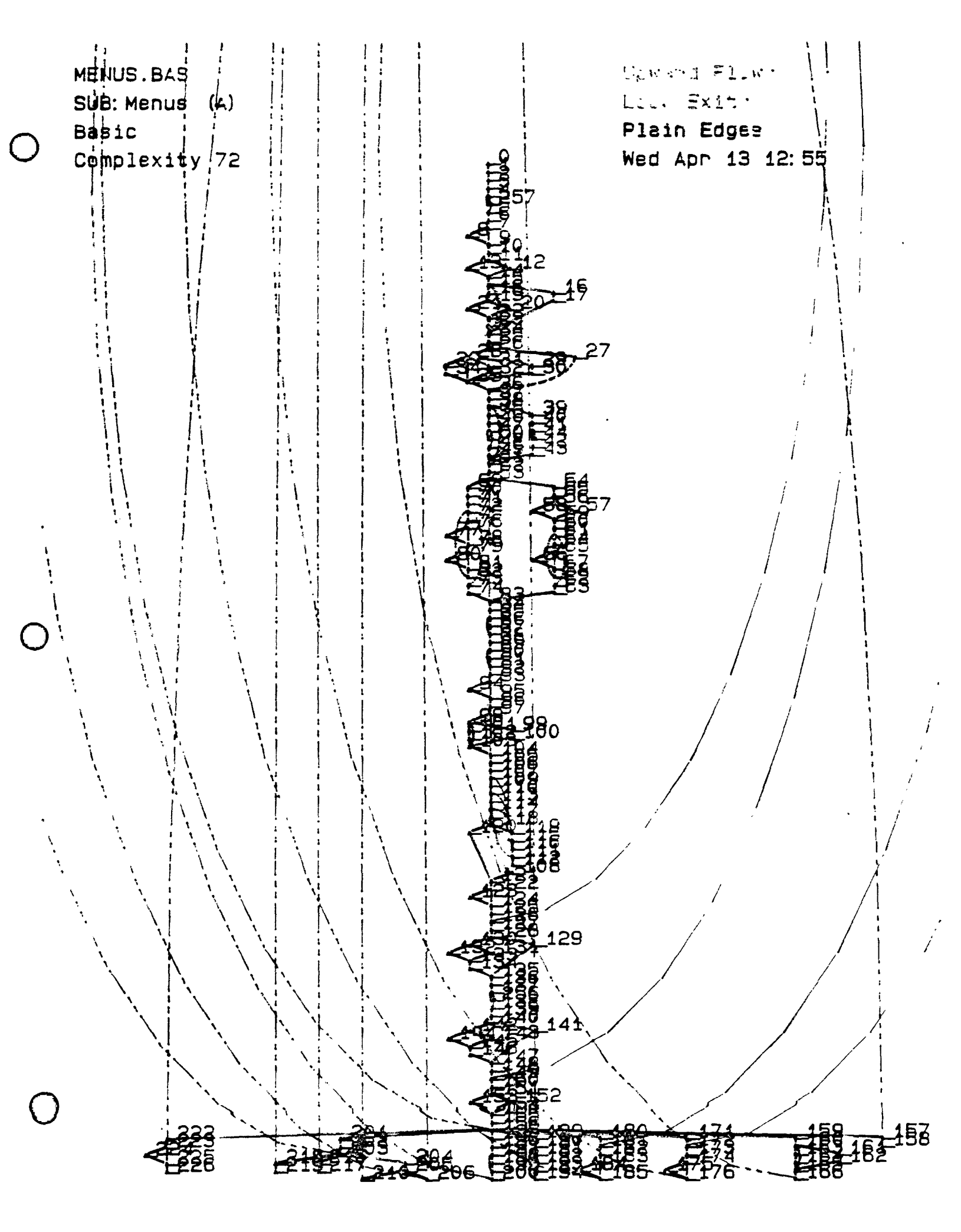
Tue Apr 12 05:23
Upward Flows
Loop Exits
Plain Edges

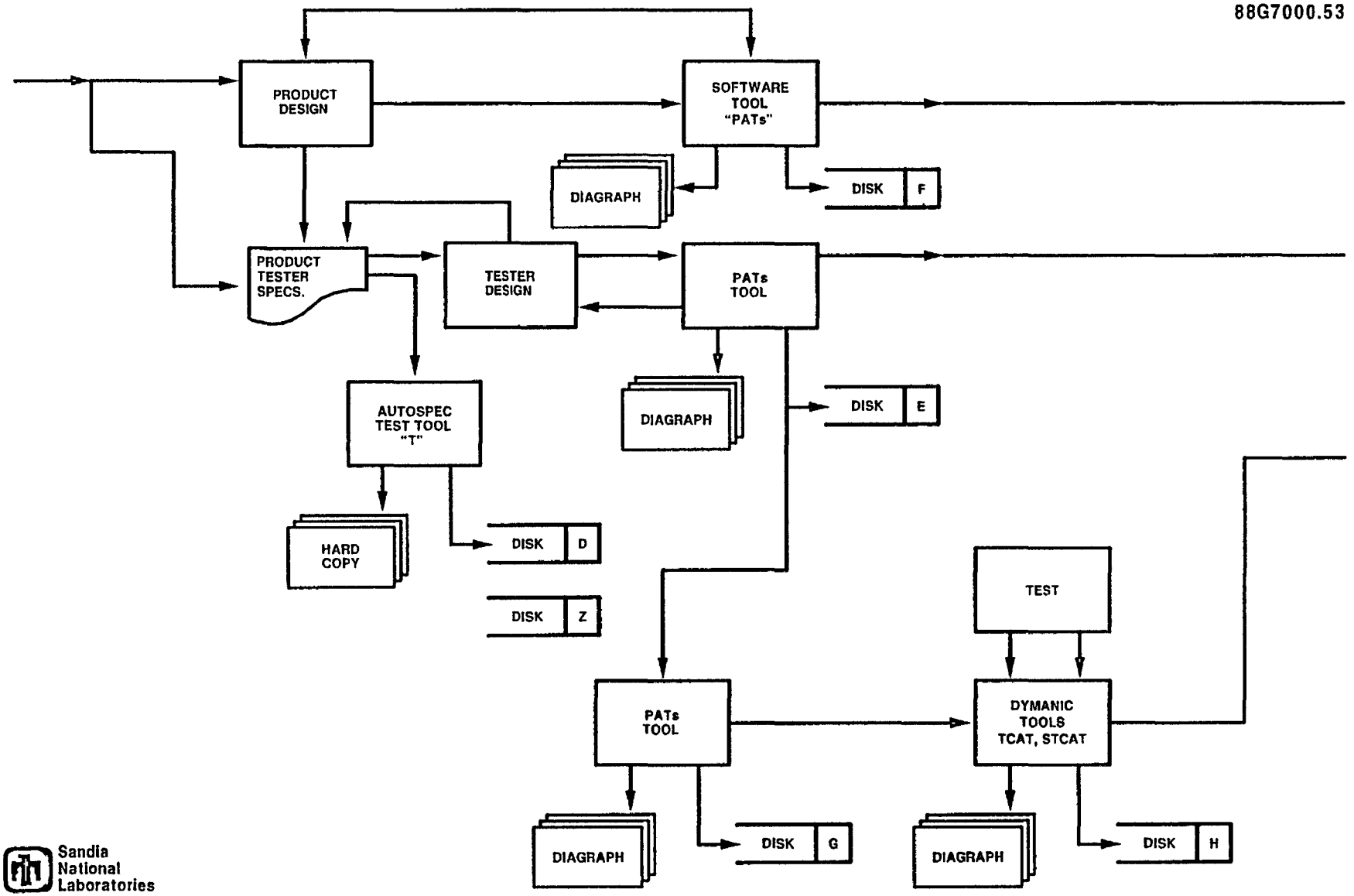


·NUS.BAS
Menus (A)
Cyclomatic 70

Wed Apr 13 14:24
Upward Flows
Loop Exits
Plain Edges



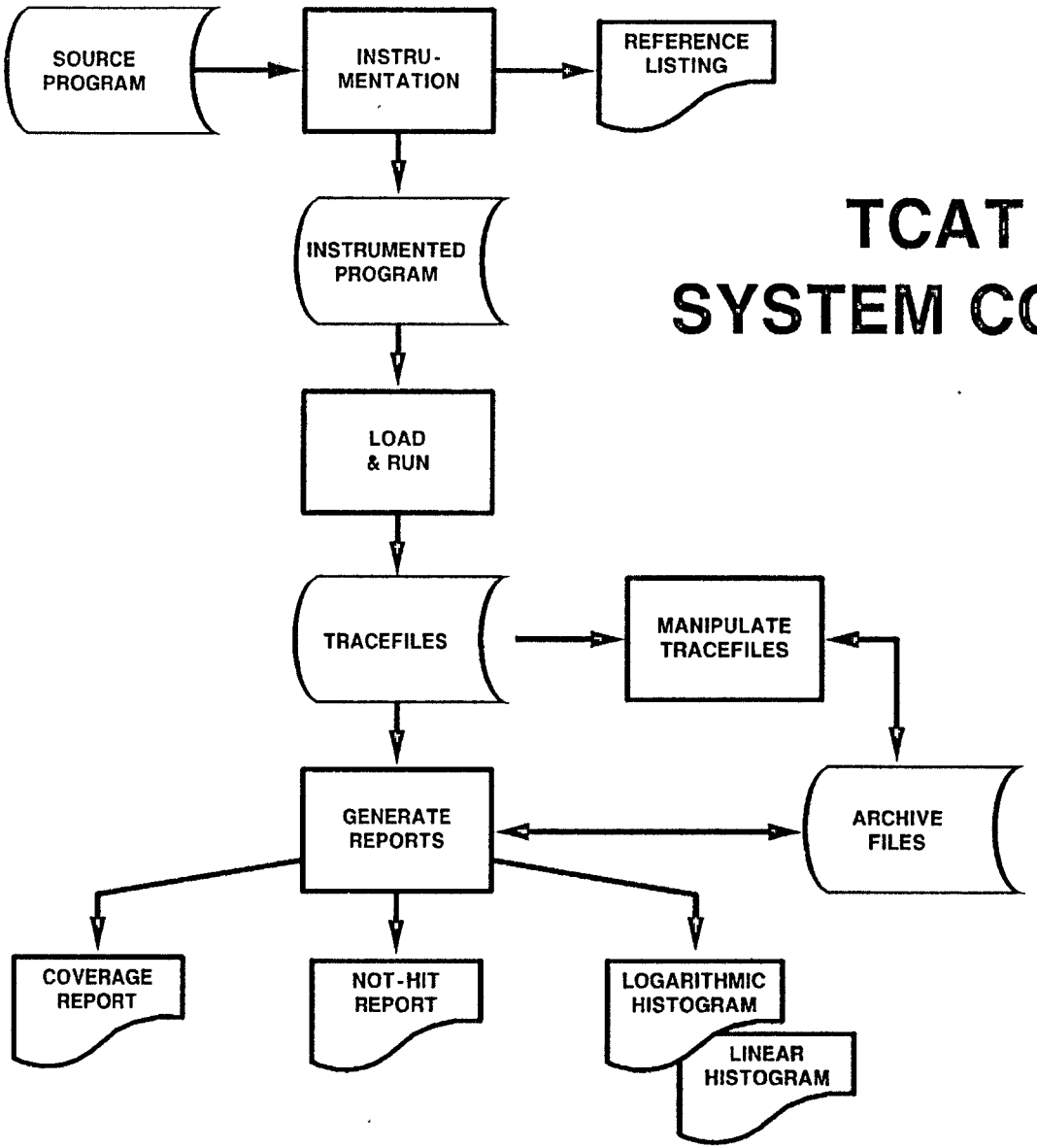

MENUS.BAS
SUB: Menus (4)
Basic
Complexity 72
Plain Edges
Wed Apr 13 12:55


88G7000.53
PRODUCT DESIGN
SOFTWARE TOOL "PATs"
DIAGRAPH
DISK F
PRODUCT TESTER SPECS.
TESTER DESIGN
PATs TOOL
AUTOSPEC TEST TOOL "T"
DIAGRAPH
DISK E
HARD COPY
DISK D
DISK Z
TEST
PATs TOOL
DYMANIC TOOLS TCAT, STCAT
DIAGRAPH
DISK G
DIAGRAPH
DISK H
Sandia National Laboratories

DYNAMIC TRACE TOOLS
(TCAT, S-TCAT)

TOOLS WHICH INSTRUMENT THE OPERATIONAL CODE AND PROVIDE TRACE AND COVERAGE REPORTS BASED UPON THE TEST CASES GENERATED

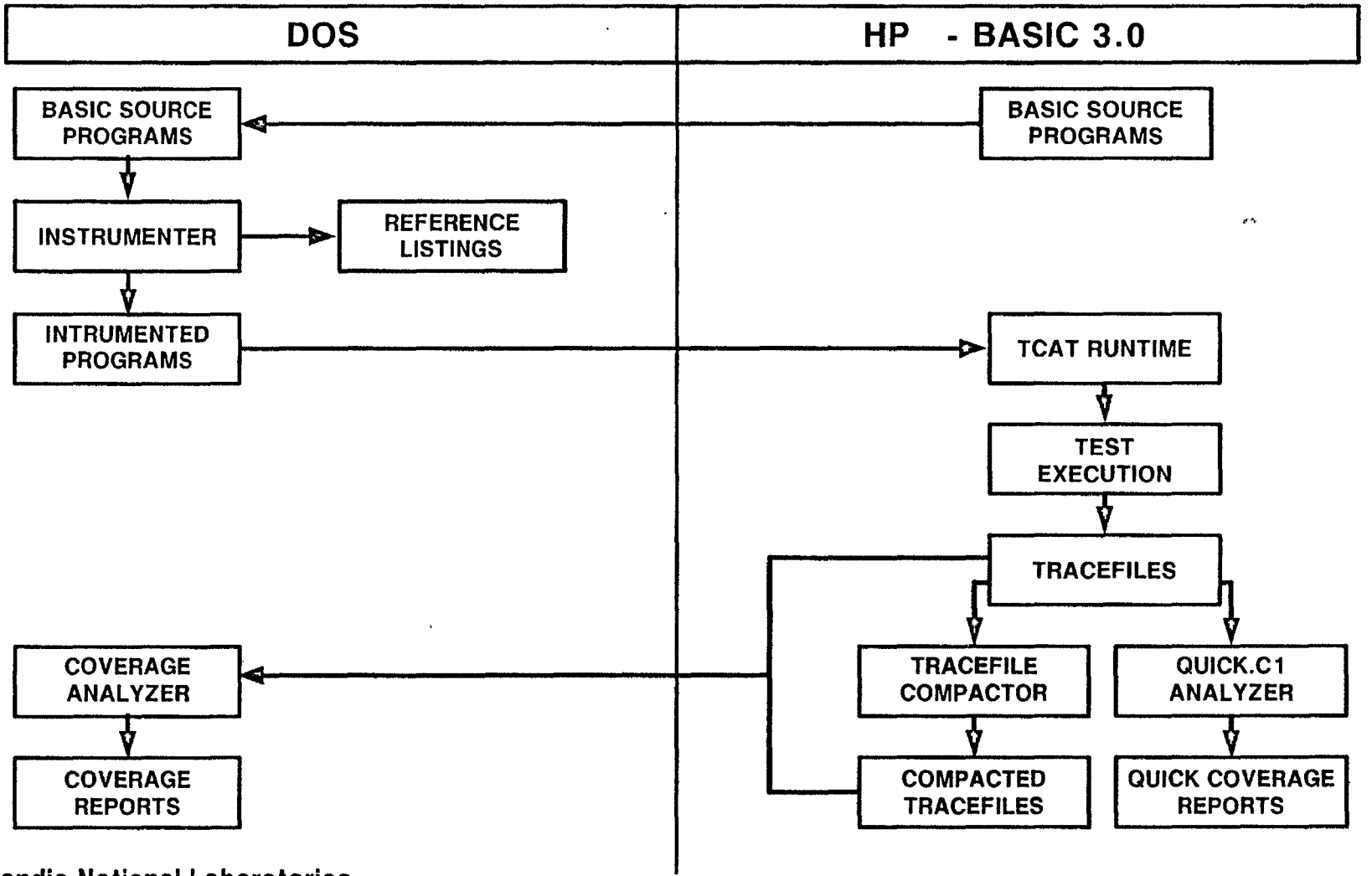
TCAT / BASIC SYSTEM COMPONENTS

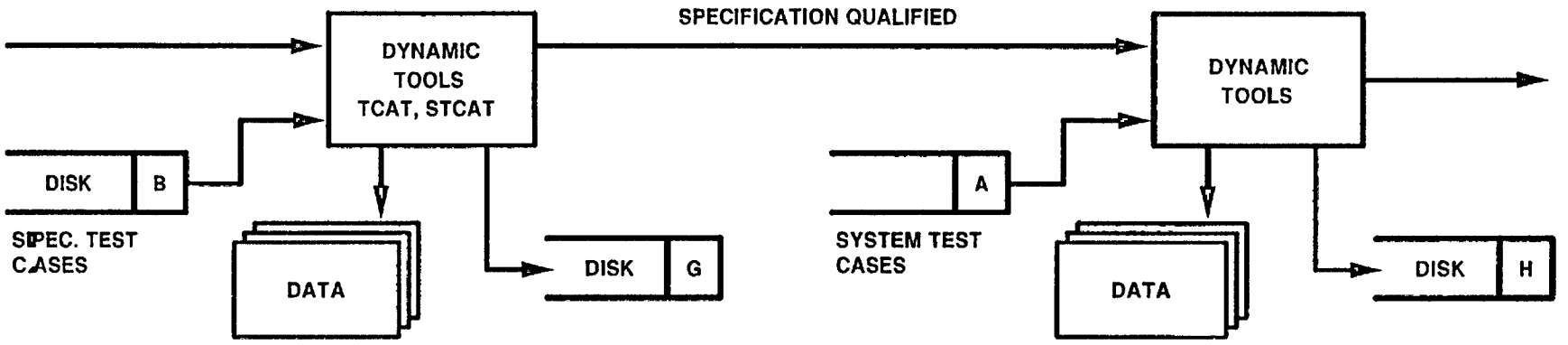


88F7000.20

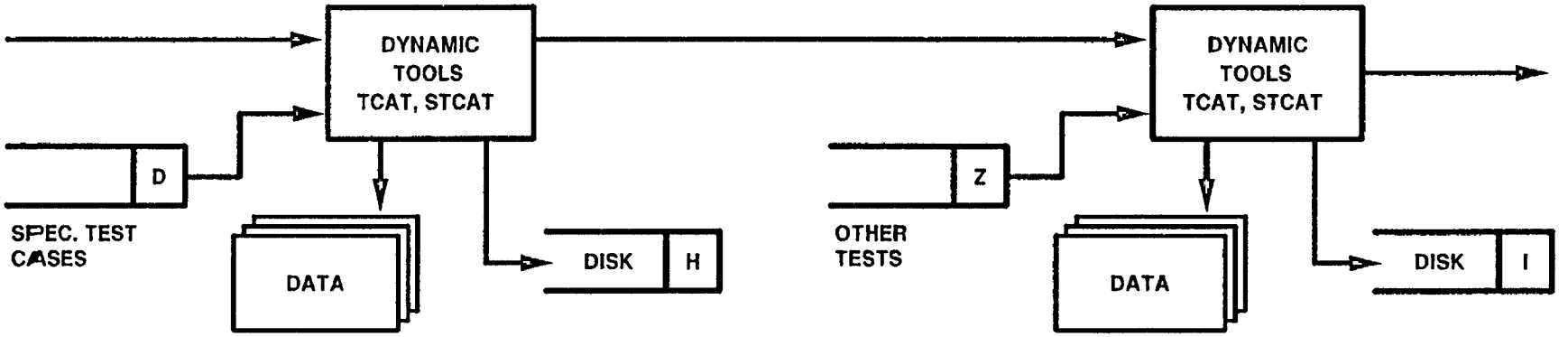
Sandia National Laboratories

OVERVIEW OF TCAT / BASIC FOR HP BASIC 3.0

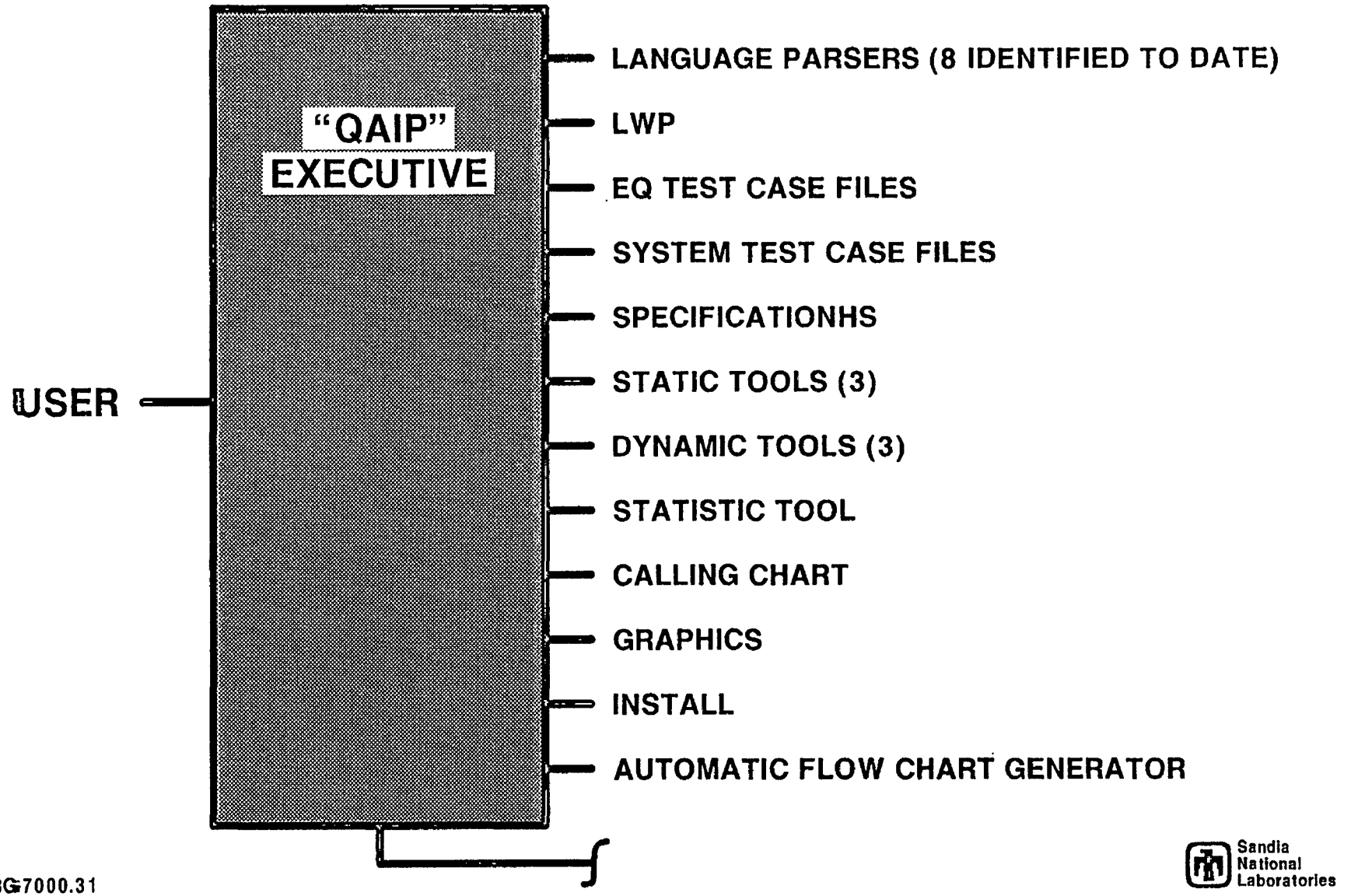




PRODUCT F.A.T.

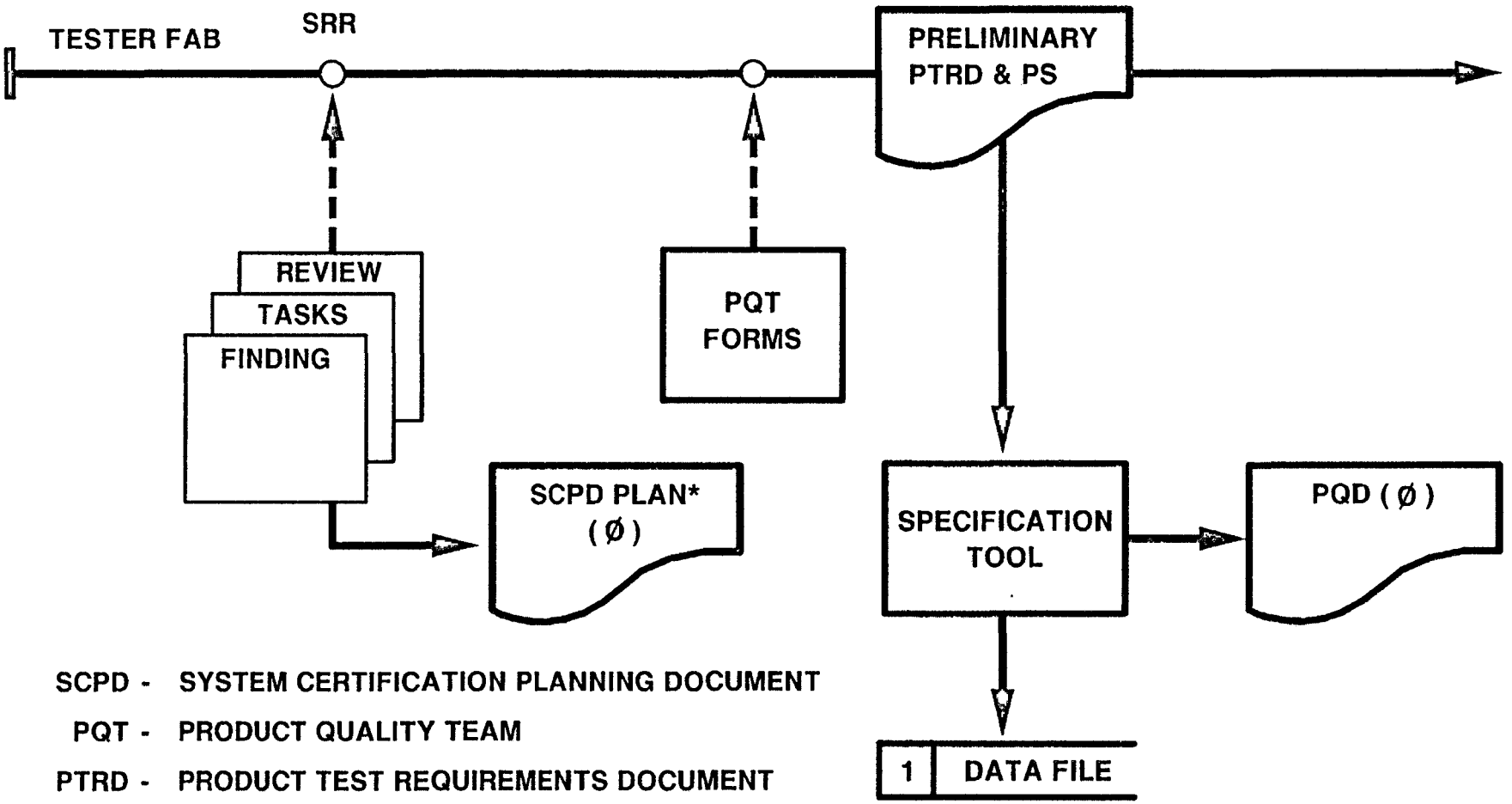


Sandia National Laboratories



88G7000.31

Sandia National Laboratories



SCPD - SYSTEM CERTIFICATION PLANNING DOCUMENT

PQT - PRODUCT QUALITY TEAM

PTRD - PRODUCT TEST REQUIREMENTS DOCUMENT

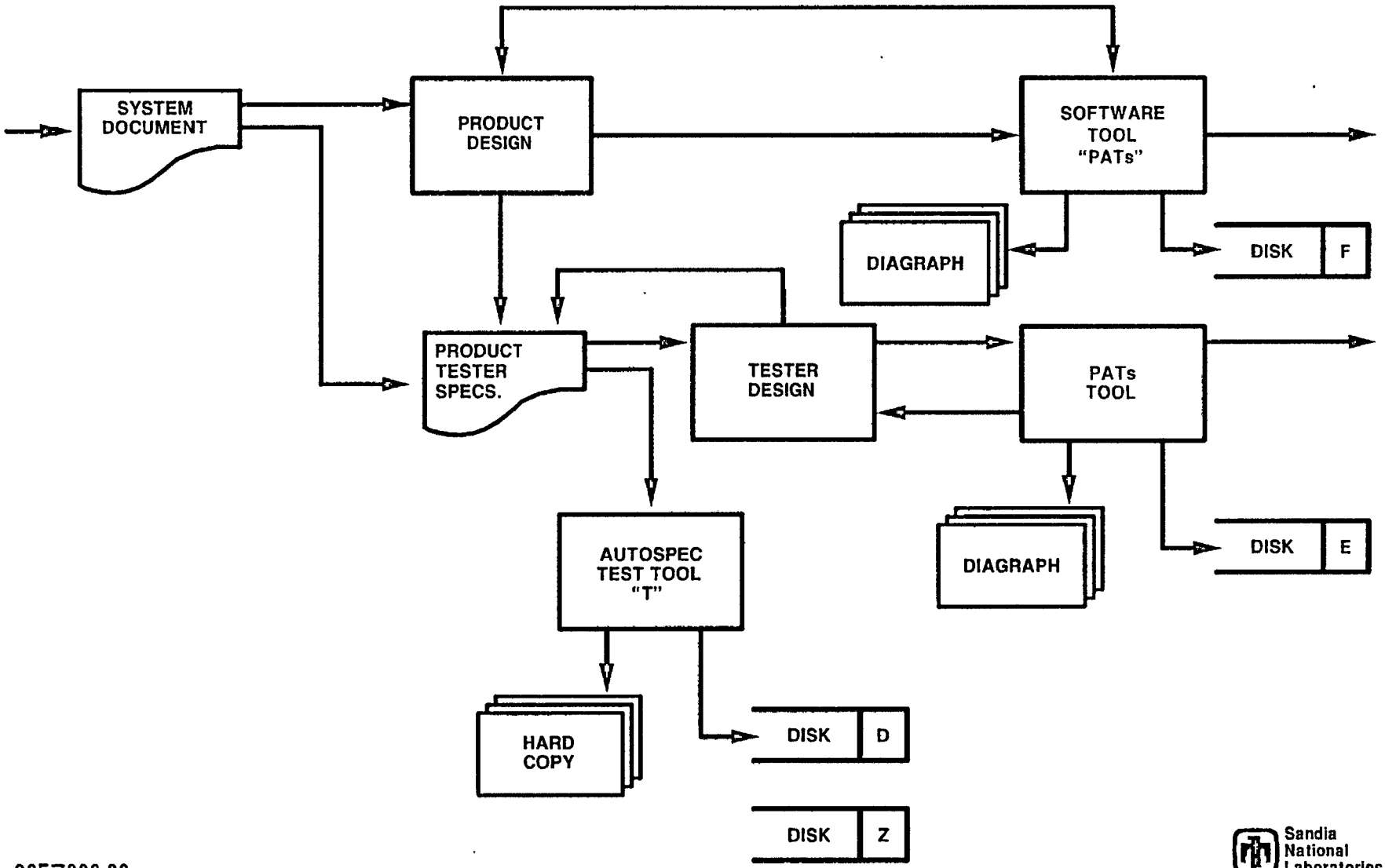
PS - PRODUCT SPECIFICATIONS

PQD - PRODUCT QUALIFICATION DOCUMENT

SRR - SYSTEM REQUIREMENTS REVIEW

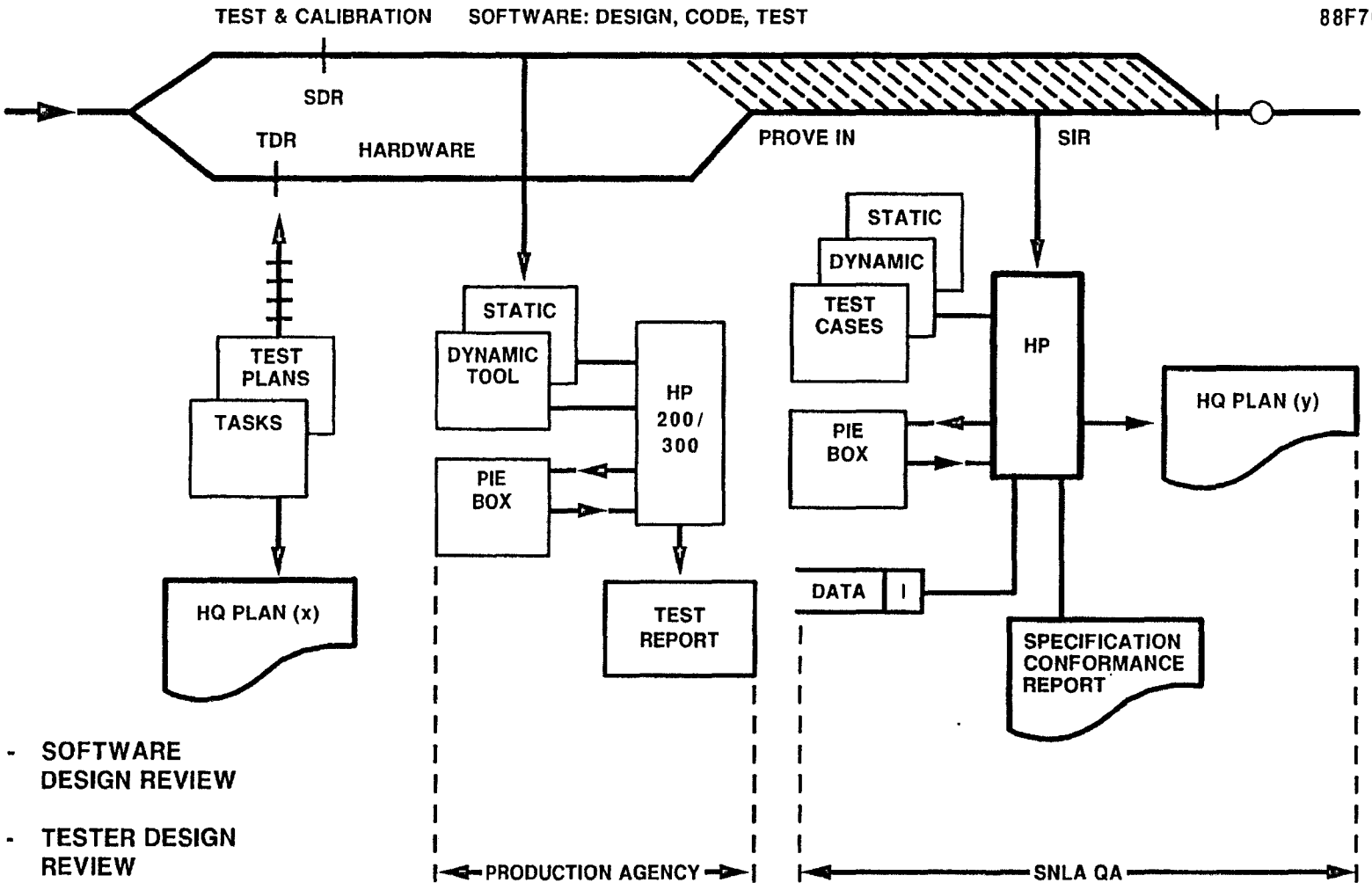
88F7000.40

Sandia National Laboratories



88F7000.22

Sandia National Laboratories

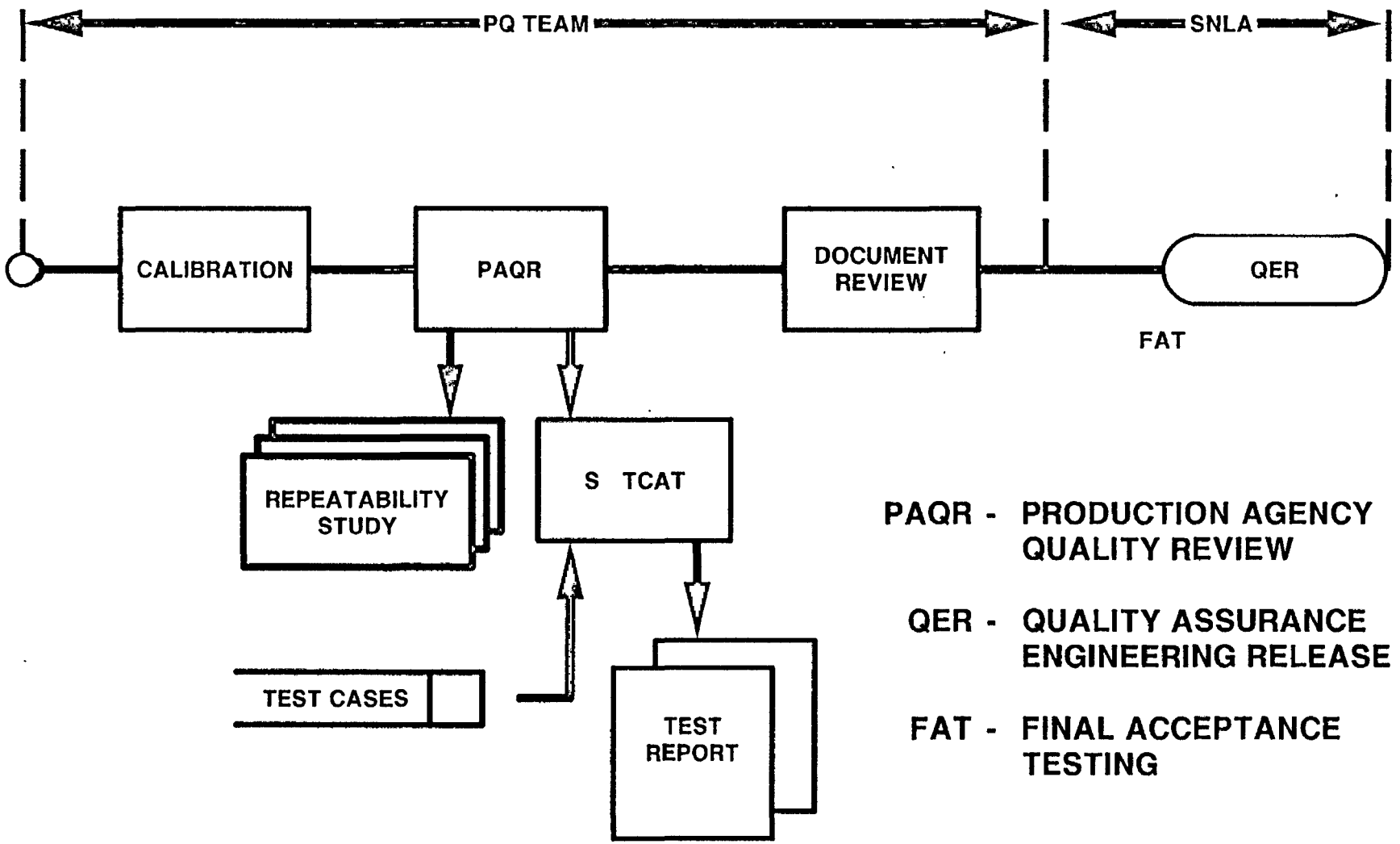


SDR - SOFTWARE DESIGN REVIEW

TDR - TESTER DESIGN REVIEW

SIR - SPECIFICATION IMPLEMENTATION REVIEW

Sandia National Laboratories



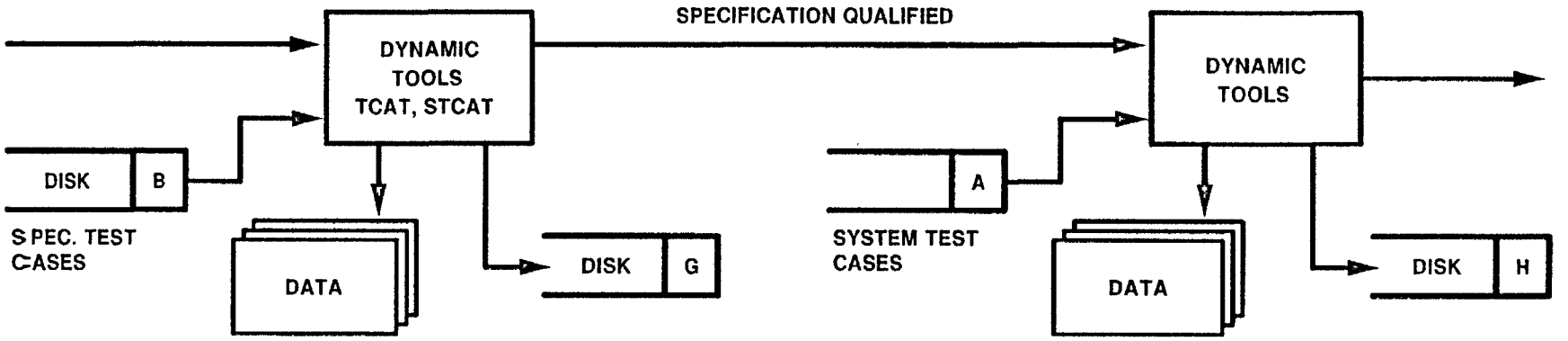
PAQR - PRODUCTION AGENCY QUALITY REVIEW

QER - QUALITY ASSURANCE ENGINEERING RELEASE

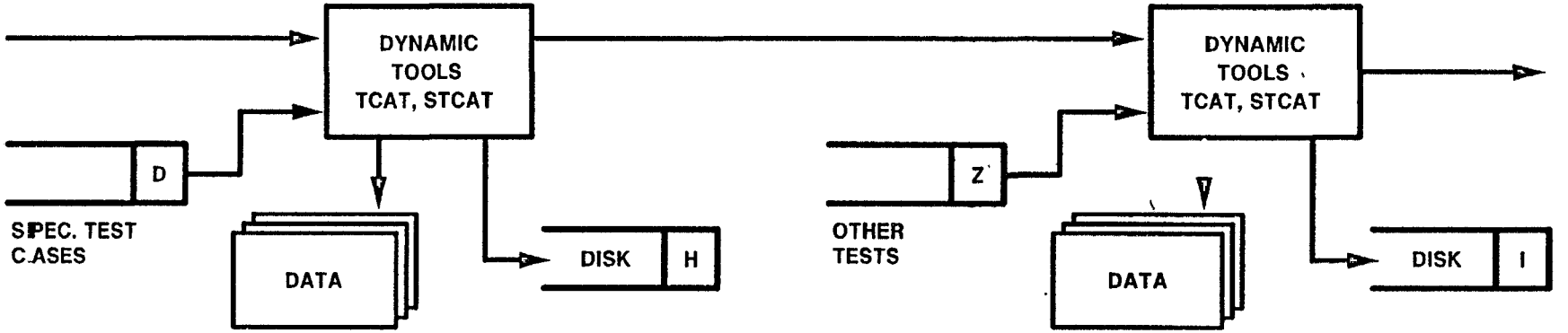
FAT - FINAL ACCEPTANCE TESTING

88F 7000.39

Sandia National Laboratories



PRODUCT F.A.T.



88G-7000.54

Sandia National Laboratories

Edward F. Miller

Software Research, Inc.
San Francisco, California

Topic: ***Starting a Q.A. Testing Group from Scratch***

Edward Miller is President and Technical Director of Software Research, Inc. (SR), San Francisco, California. SR specializes in software quality management and high quality software engineering. Dr. Miller has worked in the software quality management field for 20 years in a variety of capacities. He has been involved in the development of families of automated software and analysis support tools. He was chairman of the 1985 First International Conference on Computer Workstations, and has participated in IEEE conference organizing activities for many years. He is the author of *Software Testing and Validation Techniques (Second Edition)*, an IEEE Computer Society Press tutorial text. Dr. Miller serves as chairman of SR's Quality Week and will present the 1-day seminar: *Advanced Software Test Methods.*

Vern Crandall

Brigham Young University
Provo, Utah

Topic: ***Software Quality: Product Verification, Design Analysis, and Code***

Vern Crandall started the software engineering curriculum at Brigham Young University in 1975. Dr. Crandall currently teaches a novel course in which students test commercial software submitted by major companies nationwide, including NOVELL, WordPerfect, Microsoft, and IBM. For the past ten years, in addition to teaching at BYU, he has taught software design and improved programming technology worldwide for IBM. Dr. Crandall started in the computer business 35 years ago, working in his father's service bureau, and majored in Biomathematics in Medical School.

Some Thoughts on Software Quality

Product Verification Testing

Design Analysis

Code Quality

Dr. Vern J. Crandall

Brigham Young University/Novell Corporation

Quality Week

Software Research, Inc.

San Francisco, California

May 13, 1988

Overview

- Introduction
- Product Verification Testing
- Design Analysis
- Code Quality
- Conclusions

INTRODUCTION

- TESTING--WHILE IT HAS BEEN AROUND SINCE PROGRAMMING CAME INTO BEING--IS STILL A NEW, SOMEWHAT UNDEFINED AREA.

  - IT LACKS CONSISTENT DEFINITIONS.
  - IT LACKS STRATEGIES.
  - IT LACKS METHODOLOGIES.
  - IT LACKS FRAMEWORKS.
  - IT LACKS AUTOMATED TOOLS.
  - IT LACKS MANAGEMENT SUPPORT AND ADEQUATE BUDGET AND TIME.

EVEN THOUGH MANY BOOKS HAVE BEEN WRITTEN ON THE SUBJECT.

INTRODUCTION

- MUCH HAS BEEN WRITTEN AT THE **UNIT TEST LEVEL**.
- SOME HAS BEEN WRITTEN AT THE **INTEGRATION** AND **SYSTEM TEST LEVEL**.
- PRACTICALLY NOTHING HAS BEEN WRITTEN AT THE **DESIGN ANALYSIS LEVEL** OR THE **PRODUCT VERIFICATION LEVEL**.

A MAJOR CONSIDERATION IS THAT IT IS DIFFICULT TO COME UP WITH A FRAMEWORK FOR TESTING AN ENTIRE PRODUCT AND HAVE IT APPLY ACROSS A NUMBER OF DIFFERENT PRODUCTS. EACH PRODUCT SEEMS TO HAVE A DIFFERING SET OF CHARACTERISTICS WHICH MUST BE TESTED IN COMPLETELY DIFFERENT WAYS.

- BECAUSE THE SOURCE CODE AND SOFTWARE ARCHITECTURE DESIGN DOCUMENTATION ARE RARELY AVAILABLE TO THOSE DOING **PRODUCT VERIFICATION TESTING**, TESTING MUST REVOLVE AROUND USER'S MANUALS, SCREENS AND SCREEN DOCUMENTATION, AND HELP FILES.

INTRODUCTION

- SINCE THESE THREE AREAS ARE SELDOM--IF EVER--ADDRESSED, I WILL BRIEFLY DISCUSS:
  - PRODUCT VERIFICATION TESTING
  - DESIGN ANALYSIS
  - CODE QUALITY

FROM A TESTING AND MAINTENANCE POINT-OF-VIEW.

INTRODUCTION

- SOFTWARE DEVELOPMENT HAS THREE GROUPS WITH OFTEN CONFLICTING GOALS:
  - SOFTWARE DEVELOPERS--"CORRECTNESS" AND EFFICIENCY.
  - SOFTWARE TESTERS--EASE OF TESTING.
  - SOFTWARE MAINTENANCE PROGRAMMERS--EASE OF MAINTENANCE.
- USUALLY **SOFTWARE TESTERS** AND **SOFTWARE MAINTENANCE PROGRAMMERS** HAVE SIMILAR--OR THE SAME--GOALS, BUT NOT ALWAYS!
- ALL THREE GROUPS SHOULD BE INVOLVED WITH EACH OTHER ACROSS THE **SOFTWARE DEVELOPMENT LIFE CYCLE** TO PROVIDE FOR THE BEST COMPROMISE AMONG THE COMPETING GOALS.
  - DESIGN REVIEWS.
  - CODE INSPECTIONS.
  - TEST CASE INSPECTIONS.
  - ETC.

PRODUCT VERIFICATION TESTING

DEFINITIONS IN THE TEST ENVIRONMENT:

PRODUCT VERIFICATION TESTING IS LOOKING AT "CORRECTNESS" AND "QUALITY" OF A SOFTWARE PRODUCT FROM THE OUTSIDE-IN. IT DIFFERS FROM STANDARD, WELL-KNOWN TEST PROCEDURES IN THAT ONE DOES NOT LOOK AT THE CODE OR THE SOFTWARE STRUCTURE. RATHER, ONE EXAMINES THE PRODUCT FROM THE **USER'S** PERSPECTIVE.

TYPES OF TESTING PERFORMED IN THIS CONTEXT ARE:

- USABILITY TESTING:
  TEST THE RELATIVE EASE OF USING THE PROGRAM, ITS SCREENS, FUNCTIONS, USER'S MANUALS, ETC.--FROM THE **USER'S** POINT-OF-VIEW. ARE THE PF KEYS CONSISTENT FROM SCREEN TO SCREEN? ARE THE SCREEN FORMATS CONSISTENT, CLEAR, AND EASY TO FOLLOW? ETC.

- FUNCTIONALITY TESTING:
  TEST THAT ALL THE FUNCTIONS **NECESSARY FOR THE OPERATION OF THE PROGRAM FROM THE USER'S POINT-OF-VIEW** ARE PRESENT--AND ARE EASY TO IDENTIFY AND EXECUTE. CAN A USER DO WHAT HE/SHE WANTS TO DO WITH THE PROGRAM?

PRODUCT VERIFICATION TESTING

- DEFINITIONS IN THE TEST ENVIRONMENT:

  - PERFORMANCE TESTING:

    TEST THE PERFORMANCE OF THE PROGRAM. DO ALL THE FUNCTIONS OPERATE FAST ENOUGH? ARE ALL OPERATIONS **CONSISTENT** FROM SCREEN TO SCREEN, POINT TO POINT WITHIN A SCREEN, ETC.? ARE THERE UNNECESSARY KEY STROKES IN GETTING FROM FUNCTION TO FUNCTION, ETC.?

  - RELIABILITY TESTING:

    TEST THE ABILITY OF THE PROGRAM TO PERFORM CONSISTENTLY OVER LONG PERIODS OF TIME--IN CONNECTION WITH THE HARDWARE--WITHOUT PRODUCING SYSTEM CRASHES, DATA (OR CALCULATION) INTEGRITY PROBLEMS, OR I/O ERRORS. [ETC.]

DESIGN ANALYSIS

. ERROR TRACKING

. THE ONLY WAY TO DETERMINE THE COST OF FINDING AND FIXING ERRORS AND DETERMINING WHEN AND WHERE THEY SHOULD HAVE BEEN FOUND IS THROUGH **ERROR TRACKING.**

. A MECHANISM MUST BE PUT IN PLACE TO CAPTURE ROUTINELY INFORMATION ABOUT ERRORS--**INCLUDING** ERRORS WHICH PROGRAMMERS FIND WHILE DESIGNING, PROGRAMMING, CODING, AND COMPILING THEIR PROGRAMS.

. ERRORS SHOULD BE TRACED BACK INTO THE CODE TO THE ACTUAL MODULE IN WHICH THEY OCCURRED; THEY SHOULD BE TRACKED BACK INTO THE LIFE CYCLE TO DETERMINE WHETHER THEY COULD HAVE BEEN CAUGHT EARLIER WITH APPROPRIATE DESIGN REVIEWS, CODE INSPECTIONS, OR TESTING.

CHECKLISTS AND OTHER SAFEGUARDS SHOULD BE PUT IN PLACE EARLY IN THE LIFE CYCLE TO AID IN FINDING THESE ERRORS AT POINTS AND TIMES WHERE THEY ARE RELATIVELY INEXPENSIVE TO FIND AND FIX.

DESIGN ANALYSIS

. DESIGN ANALYSIS USING METRICS

TESTING CAN BE MADE MORE EFFECTIVE BY ANALYZING THE PROGRAM AND TESTING CODE WHICH IS MOST LIKELY TO CONTAIN ERRORS.

CODE LIKELY TO CONTAIN ERRORS CAN SOMETIMES BE DETERMINED THROUGH SEVERAL MEANS:

- **EXPERIENCE:** SOME CODE TENDS TO BE ERROR-PRONE BY ITS VERY NATURE, E.G., ERROR HANDLING ROUTINES. EXPERIENCE ACROSS SEVERAL PROJECTS CAN POINT TOWARD SUCH AREAS. ASSOCIATING ERROR RATES WITH CERTAIN TYPES OF MODULES CAN AID IN IDENTIFYING THEM.

- **DESIGN QUALITY:** WHERE POOR SOFTWARE ARCHITECTURAL DESIGN (MYERS' CONCEPTS) IS PRESENT, ERRORS TEND TO OCCUR.

- **CODE QUALITY:** WHERE CODE IS UNNECESSARILY COMPLEX AND CONTAINS MANY UNSTRUCTURED STATEMENTS, ERRORS TEND TO OCCUR.

DESIGN ANALYSIS

- **DESIGN ANALYSIS USING METRICS**

  - **HALSTED'S SOFTWARE SCIENCE METRIC:** HALSTED'S MEASURE HAS TENDED TO BE A GOOD PREDICTOR OF ERROR RATES IN SOME SOFTWARE PROJECTS. ROUTINELY CALCULATING IT AND RELATING IT TO ERROR RATES CAN VALIDATE ITS USE IN YOUR ORGANIZATION.

  - **MCCABE'S COMPLEXITY MEASURE:** WHEN MCCABE'S MEASURE BECOMES GREATER THAN 10, THERE IS A STEP FUNCTION IN THE ERROR RATE (IN MANY STUDIES). IT CAN BE USED TO REGULATE MODULE SIZE (AS LONG AS IT IS NOT MISUSED) AND TO POINT TO ERROR-PRONE MODULES.

DESIGN ANALYSIS

- **DESIGN ANALYSIS USING METRICS**

  - IMPROVED TESTING STRATEGIES CAN MAKE THE TESTING PROCESS MORE EFFECTIVE:

    - ANALYZE SOFTWARE ARCHITECTURE (DESIGN ANALYSIS):

      - MINIMAL TESTING ON CODE MOST OFTEN USED (LET **ALPHA-TESTING** CATCH MANY OF THESE ERRORS).

      - MAXIMUM TESTING ON CODE WHICH IS DETERMINED TO BE **CRITICAL TO THE OPERATION OF THE SOFTWARE** AND ON CODE WHICH (THROUGH PROPER USE OF METRICS) HAS BEEN DETERMINED TO BE **POTENTIALLY ERROR-PRONE.**

      - MINIMAL TESTING ON CODE WHICH IS NON-CRITICAL (ONLY USED BY "EXPERTS" DOING "NON-STANDARD" FUNCTIONS) OR (ONLY THERE TO HANDLE RARE, BUT NON-STRATEGIC "WHAT IF?" CONDITIONS).

DESIGN ANALYSIS

- DESIGN ANALYSIS USING METRICS

  - FORCE ERROR-PRONE MODULES TO BE RE-DESIGNED AND RE-PROGRAMMED! ON IMS, THEY FOUND THAT 20 % OF THE MODULES PRODUCED 80% OF THE APARS! REPROGRAMMING THEM EARLY-ON WOULD HAVE PRODUCED A SUBSTANTIAL SAVINGS TO THE OVER-ALL COST OF THE PRODUCT.

    WHEN A MODULE CONTAINS TOO MANY ERRORS, HAS A HIGH COMPLEXITY METRIC, HAS TOO MANY VIOLATIONS OF CODE QUALITY OR GOOD PROGRAMMING PRACTICE, OR IS HARD TO READ OR UNDERSTAND, IT SHOULD BE CLASSIFIED AS ERROR-PRONE AND BE RE-DESIGNED AND RE-PROGRAMMED.

MEASURES OF CODE QUALITY

- READABILITY IS MORE IMPORTANT THAN STRUCTURED PROGRAMMING CONSIDERATIONS

- PROPER PROGRAM AND PROPER PROGRAM SEGMENTS
  - SINGLE ENTRY/SINGLE EXIT
  - ALL CODE "REACHABLE" (NO "DEAD" CODE)
  - ALL LOOPS "FINITE" (NO "ETERNAL" LOOPS)

- MAINTAINABILITY

  (USE OF SMALL NUMBER OF PATTERNS AND STRUCTURES)
  - 3 BASIC STRUCTURES
    - SIMPLE SEQUENCE
    - ITERATION

      (DO WHILE STRUCTURE UNLESS USE OF OTHER STRUCTURES JUSTIFIED)
    - SELECTION

      (CASE STRUCTURE UNLESS USE OF OTHER STRUCTURES JUSTIFIED)
  - USE "CORRECT" STRUCTURE
  - DO NOT BRANCH OUT OF LOOPS

    (ALL EXIT CONDITIONS SHOULD BE SPECIFIED IN THE BOOLEAN EXPRESSION OF THE WHILE STATEMENT. (NESTED) IF THEN ELSE'S SHOULD BE USED TO CREATE EXIT CONDITION.)
  - DO THINGS IN THE "SAME WAY" AS MUCH AS POSSIBLE

    (USE "VARIATIONS" ONLY WHEN JUSTIFIED BY SOME CRITERIA)

MEASURES OF CODE QUALITY

- COMPLEXITY
  - NO NESTED IF THEN ELSE's
    (EXCEPT FOR SEQUENTIAL ERROR (OR OTHER TYPE) PROCESSING)
  - USE CASE STRUCTURE (WITH DOCUMENTED BOOLEAN EXPRESSIONS) IN PLACE OF NESTED IF THEN ELSE'S FOR ALTERNATIVE PROCESSING)

- AFFIRMATIVE PROGRAMMING
  (TEST AND DEFINE ALL DEFAULT CONDITIONS. USE CASE STRUCTURE IN PLACE OF IF THEN ELSE'S)

- ROBUSTNESS
  - CODE SHOULD ALWAYS RUN TO COMPLETION
  - AVOID USE OF REPEAT UNTIL STRUCTURE
  - VALIDATE ALL INPUT DATA TO MODULE
    (BUT DO NOT CAUSE "RESTRICTIVE MODULES")

- PROGRAMMER ASSUMPTIONS

- SEGMENT DEPENDENCIES

CONCLUSIONS

- THESE AREAS HAVE BEEN PRESENTED TO RAISE ISSUES RATHER THAN TO PRESENT ANSWERS.

- MORE RESEARCH NEEDS TO BE DONE IN THE TESTING AREA--ESPECIALLY AT THE SOFTWARE ARCHITECTURAL AND PRODUCT LEVELS.

- CODE QUALITY--IN TERMS OF "TESTABILITY" AND "MAINTAINABILITY"--SHOULD BE DESIGNED FOR AND INSISTED ON IN SOFTWARE DEVELOPMENT.

- BETTER DEFINITIONS NEED TO BE CREATED FOR TESTING AT THE SOFTWARE ARCHITECTURAL AND PRODUCT LEVELS.

- MORE AUTOMATED TOOLS NEED TO BE MADE AVAILABLE TO AID IN TESTING AT THE SOFTWARE ARCHITECTURAL AND PRODUCT LEVELS.

- TESTING SHOULD BE TAUGHT AS A DISCIPLINE--AN INTEGRAL PART OF COMPUTER SCIENCE--NOT AS AN AFTERTHOUGHT, BOTH IN THE UNIVERSITY AND IN INDUSTRY.

Mack W. Alford

Ascent Technology
San Jose, California

Topic: ***A Constructive Approach to Test Planning***

Mack Alford has 26 years of experience in software development and research in methods and tools to support specification, design, and testing of realtime embedded distributed software. He has authored more than ten articles on software requirements and design techniques in professional conferences and publications. Mr. Alford is an internationally recognized expert in System and Software Requirements and Design Methods, having presented tutorials in West Germany and Japan.

A REQUIREMENTS DRIVEN DESIGN APPROACH TO TEST PLANNING

MACK ALFORD
ASCENT LOGIC CORPORATION
180 Rose Orchard Way, Suite 200
San Jose, California 95134

1. INTRODUCTION

With the imposition of MIL STD 2167 (and its revisions) on the software development process, a renewed interest has been placed on the issue of test planning for critical software. To ensure that end product software will have the required capabilities, there are now requirements to subject the software to systematic, thorough testing, and that the test plans and test procedures be documented in advance of software coding, and test results be documented when testing is completed.

The purpose of this paper is to identify a current deficiency in the state of the art, i.e., the lack of a constructive method for deriving a test plan from requirements and design information; and to present a constructive test planning approach which exploits the content of the requirements/design information resulting from the use of the **Requirements Driven Design** concepts which underly the Distributed Computing Design System (DCDS) [1]. Section 2 will present a brief overview of the current state of the art of test planning, and note some of its deficiencies. Section 3 then presents an overview of the Requirements Driven Design concepts which underly the DCDS methods and tools, and how the resulting information is used to constructively generate a test plan. Section 4 draws some conclusions from the discussion.

2. AN OVERVIEW OF THE STATE OF THE ART

MIL STD 2167 requires that test plans be written, and requires traceability from any test back to the requirements and design element tested. An overview of the required content of a test plan is presented in Section 2.1. Unfortunately, the method used to systematically identify tests is left to the ingenuity of the developer. Unlike requirements and design methods, where literally hundreds of software requirements/design methods have been published, there is a distinct lack of methods available for the construction of test plans. At best there is a test planning "folklore" of the kinds of testing that are needed to sufficiently exercise software. Some of the content of the folklore is discussed in Section 2.2 below. A discussion with software test professionals will yield the identification of perhaps two methods for test planning -- the document driven method, and the Deutsch method. The deficiencies of these methods are discussed in Sections 2.3 and 2.4.

2.1 The 8 Dimensions of a Test Plan

A test plan identifies all of the tests to be performed on the components of the software. One way to view a test plan is that it must specify 8 different aspects of each test and their interrelationships. A brief description of the 8 aspects is presented below.

1. requirement to be tested -- identifies which requirement from the requirements document is being addressed. Note that the total of all tests should test all requirements.
2. software to be tested -- early software tests usually exercise only a portion of the software modules. This allows some programming and testing to be concurrent.
3. when the test is to occur -- this defines the preconditions for a test (e.g., which tests must have previously been successfully completed, and the expected date on which the test is to occur. The test plan should avoid a test plan with a strict sequence of tests -- if one test fails, and time will be required to fix the software before retest, the testers should be able to move on and execute other tests.
4. hardware required -- some early tests can be performed on the host processor used to develop the code (e.g., unit testing), while other testing requires execution on a fully

populated distributed target processor. A strategy for moving from host to fully populated target architecture may have to take into account limited availability of hardware during early development, problems in scheduling the hardware for test, and take into account the inability to extract information from a target processor.

5. kind of test -- this identifies a unit of software is to be tested with stubs for the procedures it calls, or whether the modules are to be included; whether the test is open loop with the environment, or whether a closed loop simulation of the environment is to be included; whether functionality and accuracy are being tested, or whether execution and/or response times are to be measured.
6. support software required -- the support software varies according to test. Bottoms up unit testing requires some sort of scaffolding to insert data and extract and analyze the results; top down unit testing requires the generation of stubs; closed loop testing with the environment requires an environment simulator of some sort. Tests which verify some level of testing completeness measures (e.g., every branch or path for unit testing, every procedure called for integration testing) require some additional test tools to measure this completeness. This software must be specified, developed, and tested before it can be used to test the target software. Frequently, the specification and testing of environmental simulators may require larger development efforts than for development of the target software, and impose additional constraints on the availability of the target hardware.
7. test case inputs and outputs -- sometimes a set of paths will first be tested with inputs from a small number of objects, and later tested with a maximum legal load set of inputs, and finally with inputs which are supposed to exercise the load-shedding or "graceful degradation" features of the software. Since different input scenarios may stress the software system at different points, any of several test cases may be executed by the software for a given test. The preparation of the different test case inputs may require substantial effort.
8. original vs. regression test -- the criterion for passing an original test is that the outputs be within allowed tolerances of expected outputs. When modifications to the software are made, and results of some previous tests are not supposed to be affected, then a regression test is performed, where the acceptance criterion is that the test outputs of the modified software should match the previous outputs exactly. This occurs frequently when errors are corrected or capabilities are upgraded. A regression test can be totally automated (thereby taking less person time) but require substantial blocks of processor time.

2.2 The Test Planning "Folklore"

If one talks to a practitioner of software testing, one will discover that the state of the practice can be loosely called "folklore". There are no well known methods in use compared to the state of the art in software design, where many designers use one form of Data Flow diagrams or another. However, if one sets in on a test plan review, one will discover that there is a well estabslished folklore on the kinds of testing that must be done:

- Unit Testing - the tests of a unit of code without the lower level units that it uses or calls
- Module testing - the tests of a unit of code which includes the lower level units that it uses or calls
- Interface testing - the tests of the interface mechanisms used by the software to communicate with other system components (e.g., input/output, but no processing)
- thread testing - the tests of a "stimulus-to-response thread" of processing of the software as a whole, verifying that the input plus state yields the correct output plus state updates.
- function testing - the tests of a "function" of the software, to demonstrate that the software behaves correctly to the input of a sequence of items(e.g., testing the "tracking" function by insertion of multiple track returns)

- object testing - the tests of the software demonstrating that a single object makes all of the required transitions between functions (e.g., demonstrating that an object is first detected, then tracked)
- multi-object testing - the tests of the software to demonstrate that multiple objects are handled correctly, including some "load testing" to demonstrate that the software handles the required full load and has the required "graceful degradation" properties if more than the maximum load is input
- exception testing - the test of the software to demonstrate that the software responds correctly to various "exceptions" (e.g., hardware exceptions, communications failures, algorithm failures)

There are schools of thought on how the testing should be accomplished, sometimes resulting in heated discussions on the relative merits and disadvantages of different test strategies. For example:

- every one agrees that the "big bang, ship on first normal termination" mode of testing, in which all modules are combined for the first time and tested, is insufficient to achieve anywhere near the desired software reliability. However, it is pointed out that the MILSTD 2167 requirements were developed as a protection against this kind of thinking.
- the "bottoms up" school of testing states that the lowest level modules should be unit tested, then combined into higher level modules, and so on until the software as a whole can be tested. This method has admitted strengths (i.e., much of the lower level testing can be perfomed independently by many developers) and weaknesses (i.e., it is not until the highest level modules are merged that interface problems are detected, requiring substantial re-work and schedule slippages).
- the "top down" school of testing states that one should first test the Job Control Language, then add the top level program module, and then add the modules one at a time from the top down. This has admitted advantages (i.e., interfaces are tested at the same time as the transformations during unit testing), but also admitted disadvantages (i.e., if the software is complex or large, it is difficult to find the particular set of inputs which will force a specific path 10 or 15 layers down in the subroutine hierarchy -- this has sometimes been referred to as "pushing with a rope").
- the "sideways in" school of testing, in which the top down and bottoms up methods are mixed together.

Unfortunately, none of this discussion addresses the critical problem of how one identifies the tests.

2.3 The Document Driven Method

The document driven method of test planning is to take each paragraph of the requirements document and attempt to construct tests which would demonstrate the compliance of the software with the requirements. There are several problems with this approach.

First, the requirements documents paragraphs are usually not testable as they stand. To address thie problem, the analysis phase of test planning extracts a set of "capabilities" from the requirements document which are either explicit or implied. The test plan then addresses the testing of these "capabilities". The definition of a "capability" is largely subjective.

A second problem is that the usual mapping of requirements onto the design elements is expressed via a matrix or table of paragraph vs. design element. Analysis of such matrices demonstrates that the content of the traceability matrices is subject to a great deal of interpretation -- thus the

identification of the modules of software necessary to test a specific requirement or capability may require a great deal of anlaysis.

Finally, even if each requirement is associated with a specific subset of the design, the document driven method does not address the remainder of the critical test planning issues -- the strategy for combining and testing modules in a time sequence, the strategy for dealing with the host/target dichotomy, the requirements for the environmental simulator, etc.

2.4 The Deutsch Method

The method presented by Deutsch [2] is the first published approach known to the author to integrate the representation of requirements, design, and test planning. Figure 1 presents an overview of the approach.

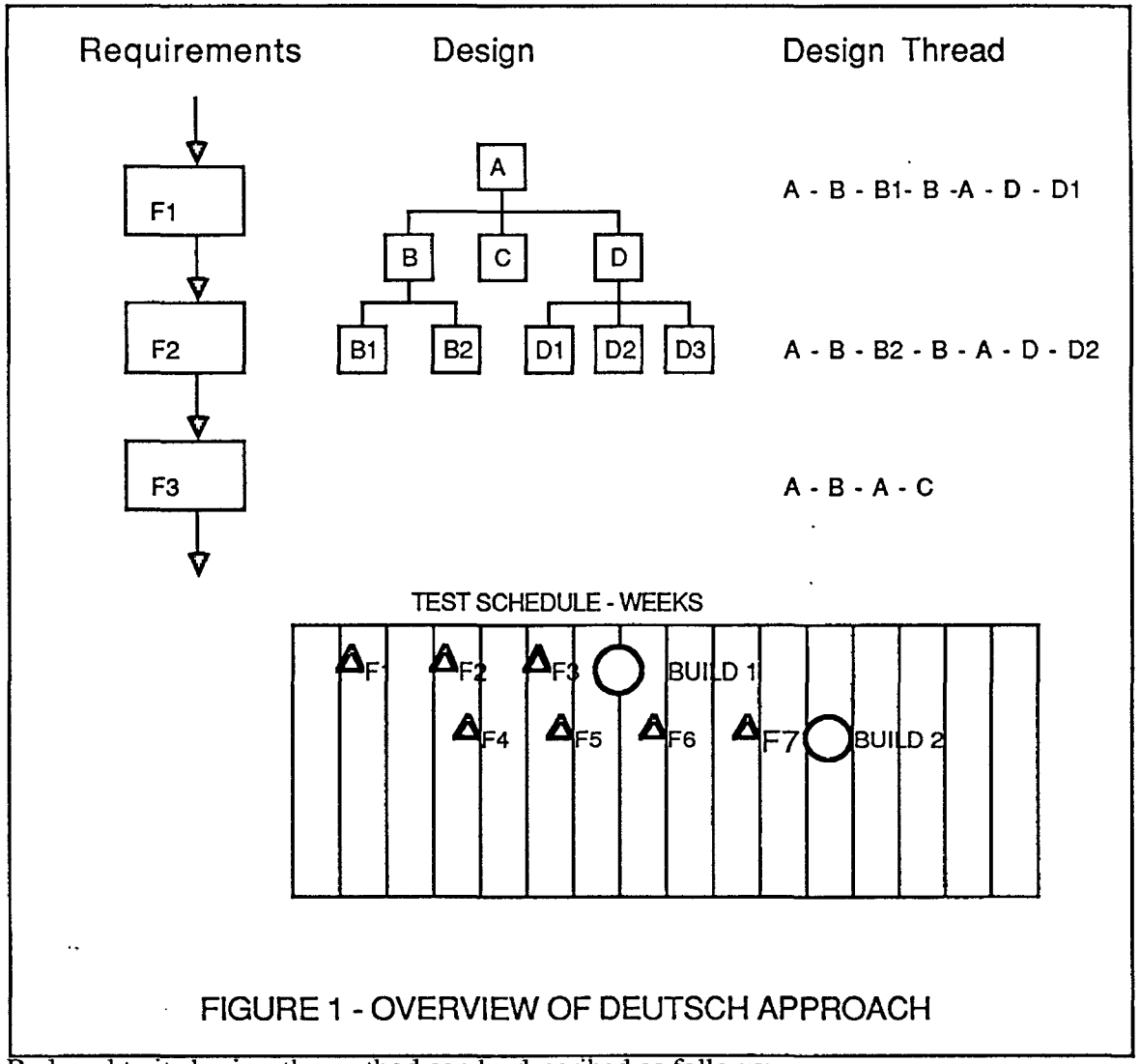


FIGURE 1 - OVERVIEW OF DEUTSCH APPROACH

Reduced to its basics, the method can be described as follows:

- express the requirements as a state machine; for each discrete function of the state machine, identify inputs and state condition, outputs and state transitions, and the traceability back to the requirements document paragraphs. This will also serve to uncover inconsistencies and incomplete requirements, and thus can serve as a verification method

- when design is complete, identify for each discrete function of the state machine a "thread" of module invocations required to implement the function. This will also serve to "verify" the consistency and completeness of the design
- for each thread, there should be one or more "tests" which will verify that the requirements are satisfied by the design elements on the thread. Construct a test plan by defining the sequence in which the threads are tested. It is recommended that the test plan use a top-down threaded approach, i.e., top down but adding modules one at a time to achieve a given thread.
- These threads should be combined into "builds" which are incrementally developed, tested, and released to the customer for early testing in order to reduce the risk of having incorrectly translated customer intent into operating software.

For a compete description of the method, readers are encouraged to read [2].

The benefits of this approach are substantial.

- it is constructive -- for each discrete function identified at the requirements level, a collection of design components are identified, and a specific test must be generated.
- the ambiguities of the requirements document are isolated to the mapping onto the state machine; the mapping of the state machine to design and test is clean.
- it lends itself easily to an incremental development approach, which reduces the risk of delivering the wrong product.

Unfortunately, this approach also has a number of distinct limitations, listed below:

- Because it is based on the model of a state machine for the requirements, it is subject to the limitations of state machines
  1) a state machine defines conditional sequences of actions, and does not allow the expression of concurrency -- this alone limits its application to embedded systems. Attempts to extend the state machine model to include concepts of concurrency destroy the underlying foundation of the model. For example, use of the state machine model to describe the behavior of 13 elevators which stop at 30 floors in response to buttons in each elevator and on each floor can become overwhelming.
  2) if a state machine description becomes larger than 50 to 100 states, its behavior becomes unwieldy to draw and very difficult to understand, because the state machine model does not provide the concept of a "hierarchy" to support a "divide and conquer" strategy. Again, refer to the elevator problem.
  3) if a discrete function (i.e., node of the state machine graph) becomes too complex, it cannot be simply characterized by its outputs and state updates. For example, if one input can yield some combination of outputs and state updates from 6 concurrent decisions, each of which has 3 choices, then the "thread" might not be the correct level of requirement to drive the testing process (e.g., see [1]). A better unit of requirement might be the "path" resulting from one of the concurrent decisions — it is easier to test 15 paths than it is to test (3x3x3x3x3x3 = 729) threads.
  4) some non-functional requirements are not directly associated with any thread of processing (e.g., safety requirements), and others are not even visible at the requirements level (e.g., if the hardware selected is subject to frequent parity errors, the software may have a feature to checkpoint and restart the software on detection of a parity error in a way which is transparent to the overall input/ouput level of processing).
- There is an implicit assumption that the target software architecture is that of a program, rather than a set of concurrent tasks needed for a real time implementation.
- Deutsch recommends a pure top down approach. It is not clear that a sideways in kind of approach might not be better for large complex software.
- No approach for developing the environmental simulator is provided.
- No approach for dividing up the testing between host processor and target processor is provided.

- The approach is not currently supported by commercially available tools, which could reduce substantially the effort required to use it on larger projects.

Thus the Deutsch approach provides some substantial advantages over the document driven method, but is subject to a number of limitations which hamper its application to larger projects.

2.5 Discussion

It appears that the lack of a constructive test planning approach can be traced back to the deficiencies of the models used to describe requirements and design. The document driven model is subject to the deficiencies of requirements specified in textual form. The Deutsch model is subject to the deficiencies of the state machine for the statement of requirements. Thus the search for a constructive test planning approach must start with a more robust representation of requirements and design.

3.0 A CONSTRUCTIVE REQUIREMENTS DRIVEN APPROACH

The constructive approach to test planning presented below is a logical consequence of using the methods of representing requirements and design described in [1], which can be summarized a **Requirements Driven Design**. The discussion starts with an overview of the methods for representing requirements, how designs are represented, and finally how the test planning approach takes advantage of this information. Figure 2 presents a cartoon overview of the approach similar to that of Figure 1.

3.1 The Requirements Model

The essence of the Requirements Driven Design approach to the representation of requirements is as follows: first define the desired system behavior; then decompose this desired behavior and allocate the functions onto the design elements; and finally add functions to detect and recover from exceptions. This can occur at a number of levels of design: allocation of system level functions between the environment and a black box system; allocation of black box system functions to components or subsystems (e.g., a data processor subsystem); allocation of data processing functions onto software design elements; or decomposition and allocation of an algorithm to units of code which will implement it. In this paper, we will focus on the problem of representing data processor level requirements and allocating them onto design elements.

The basic building block of the description is the ***discrete function*** which accepts a discrete input, generates one or more discrete outputs (including state information), and transitions to a new state to receive the next input. When a number of discrete functions are connected by a graph which defines conditional sequencing, you have by definition a ***state machine***. This part of the technology is not new -- a number of different researchers use the concept of state machines in order to represent sequencing conditions (e.g., to represent communication protocols, to represent actions of robots, and even to describe sequencing conditions in some data flow requirements/design approaches). This part of the representation method is essentially the same as that of Deutsch.

To overcome the limitations of the state machine model, the Requirements Driven Design approach provides the ability to aggregate a graph of discrete functions into a new (larger?) building block called the ***time function***. By definition, a time function accepts a structure of inputs over some finite period of time, and generates some structure of outputs during that period of time, until some ***completion condition*** is satisfied. In the same manner, a sequence of inputs or outputs can be aggregated into a new (larger?) building block called an ***item stream***. Much larger behaviors can then be represented using graphs whose nodes are time functions which input and output item streams. Graphs of time functions can be further aggregated into higher level time functions, to an arbitrary number of levels.

In addition to representing conditional sequences of functions, the Requirements Driven Design graphs provide the ability to represent various types of concurrency of functions and/or items:

- concurrent interleaved streams of input items, specifying both partial sequencing and concurrency (e.g., input from each user arrive in sequence, but may be arbitrarily interleaved between users)
- independent concurrent functions, with no interactions -- this provides the ability to define independent state machines
- interdependent concurrent functions, requiring coordination -- this provides the ability to describe the desired behavior of concurrent state machines with constraints
- replicated concurrent functions, requiring coordination (e.g., processing inputs from a number of users) -- this provides the ability to describe the behavior of many identical concurrent state machines with constraints

Such graphs of items and functions can be used to express arbitrarily complex system behaviors in a hierarchical manner which are more understandable than if the behavior was represented at a state machine. For example, the behavior of a set of elevators would be described as the behavior of replicated elevators, where each elevator responded to its button inputs and inputs from a coordination function.

If data processing functions are being described, and the discrete functions are complex, then they can be further decomposed into a stimulus-response level of description. Figure 2 indicates that a function has been decomposed to a stimulus-response description. The paths can be mapped directly onto a task, or the stimulus-response can be subdivided and mapped onto multiple paths if needed to satisfy response time requirements.

In short, the Requirements Driven Design requirements approach provides mechanisms for overcoming the basic limitations of the Deutsch requirements model:

- explicit representation of concurrency, including replications of identical functions
- explicit representation of decomposition to provide a hierarchy of functions for complex problems
- explicit decomposition of a complex discrete function into a stimulus-response representation of the concurrent paths of processing
- explicit representation of functions allocated to the environment or other subsystems, thus providing the definition of the transformation to be implemented by simulators for closed loop testing.

3.2 The Design Model

The Requirements Driven Design approach to design is to explicitly allocate required processing and data onto design elements. Three kinds of design elements are used:

- Modules, which are decomposed into algorithms which accomplish the specified transformations
- Tasks, i.e., logically concurrent units of code, whose executions are serialized by the operating system scheduler; and
- Data objects, i.e., object which encapsulate state data

Module Definition

When the discrete functions of the required processing are decomposed down to the stimulus-response level, the nodes of the graphs represent memoryless transforms (called ALPHAs in the terminology of SREM) with known data input and output. These transforms can be allocated to Modules. The Module required input and output data are allocated to variables in a programming language. Algorithms are developed in the usual top down fashion to accomplish these required transformations. In Figure 2, the module decomposition is displayed in a fashion similar to that of Figure 1.

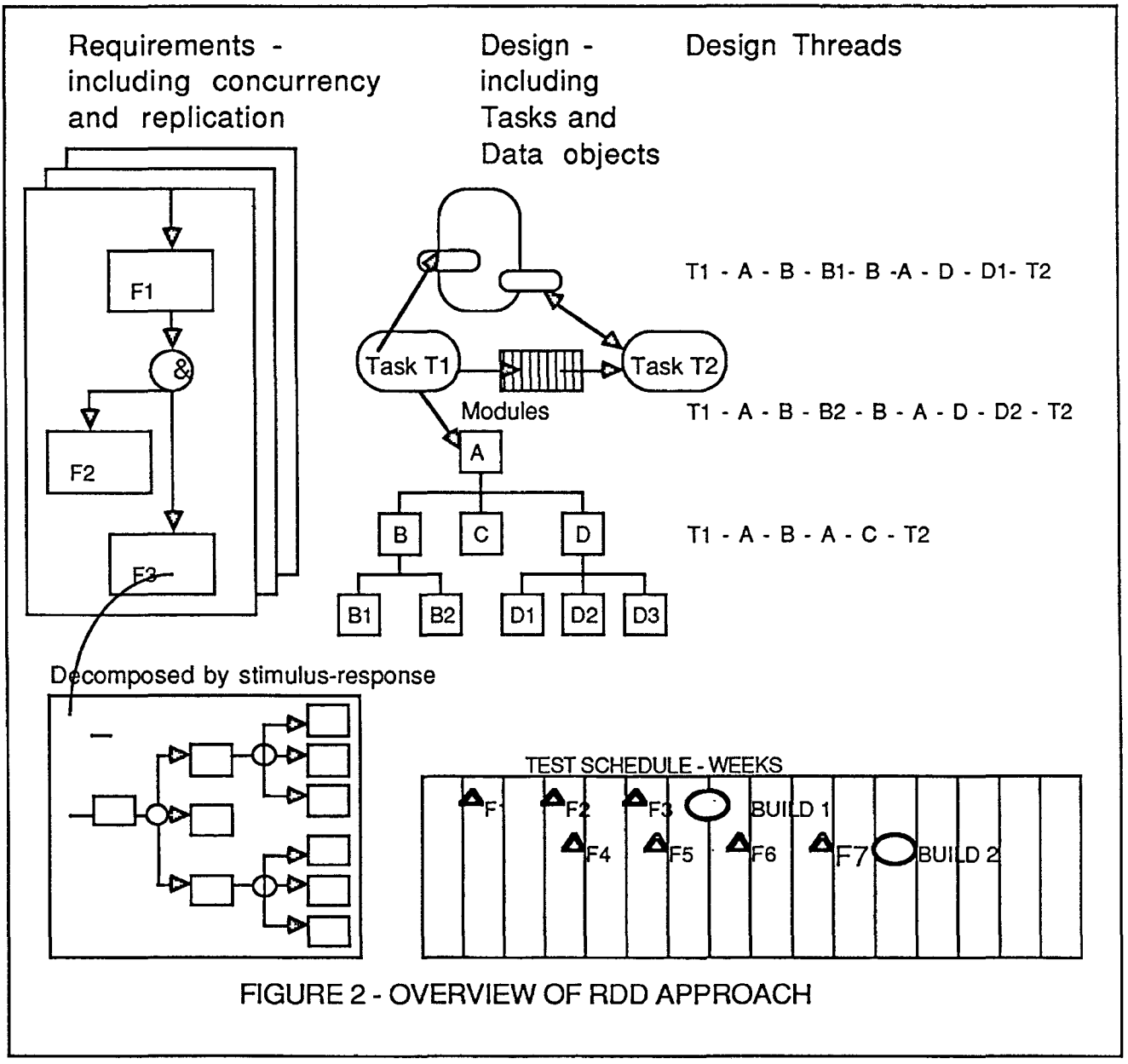


FIGURE 2 - OVERVIEW OF RDD APPROACH

Task Definition

This phase of design addresses the problem of mapping the required processing onto the tasks, i.e., the units of code scheduled by the operating system. If there is only a single task, then all of the concurrency exposed in the requirements would have to be serialized into a single program. This could certainly be done, but then the programmer would have to generate the code which polls the input lines periodically and call procedures to handle the inputs in a manner which ensured that the required processing load could be sustained. If tasking constructs are available at either the coding language level (e.g.tasks in C or Ada), the the run time system can provide the (reusable!) code to perform the serialization of the concurrency in a flexible fashion. Figure 2 illustrates that required processing has been mapped onto tasks, and that one of the tasks uses a module which is decomposed into a hierarchy of modules.

Data Object Definition

Just as processing must be allocated to tasks, state information must be allocated to data objects. In this context, a **data object** is defined as a combination of data structures and methods used by other processing elements to create, access, manipulate, update, and destroy data contents in the data structure. Recall that Booch [3] describes the defining characteristic of an object as an entity which encapsulates state information. Since the requirements portion of the Requirements Driven Design approach defines all of the required sequencing of the system and thus all of the state information required to support that sequencing, then without loss of generality, any object oriented design can be described in terms of allocation of required state information onto data objects. Note that in Figure 2 the state information has been allocated onto a data object with two methods - one for insertion, and one for extraction. A buffer linking Tasks 1 and 2 is just a standard predefined data object.

Fault Detection and Recovery

After the required processing has been allocated to the design objects, a Failure Modes Effects Analysis is carried out to identify potential faults, and identify the impact of those faults on the overall operation of the software. If it is decided to deal with a fault, functions may be added to detect and recover from the fault. These functions may then be decomposed and allocated onto design elements in order to complete the design. Thus there is a systematic method for identifying and implementing the design decisions which address the non-functional requirements (e.g., safety, availability, reliability, resource constraints).

The Mapping of Requirements to Design

The above mappings of required processing and data onto design elements is made in a fashion which explicitly preserves both the required sequencing and the data flow of the requirements. This means that any path of processing in the requirements will map into some sequence of task and module invocations in the design, as depicted on the right hand side of Figure 2. Similarly, any sequence of time functions in the requirements will be decomposed and allocated onto a set of tasks, modules, and data objects which implement it. Thus the effort required by the Deutsch approach for finding the mapping from discrete functions in the requirements onto a design thread is eliminated because of the method of constructing the design using the Requirements Driven Design approach.

3.3 The Test Planning Approach

Now we reap what has been planted during the requirements/design phases of development. Following the discussion in Section 2, one can simply read off of the diagrams all of the tests which are performed in the state of the practice:

- Unit level tests are obvious -- they apply to each of the modules and tasks.
- Module integration tests can be performed meaningfully at the top module level, at the level where a task incorporates its modules, and for all of the methods supporting a specific data object.
- Path and thread tests are equally obvious. For every path of the requirements there is a corresponding path of task and module invocations, as depicted on the right hand side of Figure 2. Thus the path (or, in simple cases, the thread) serves the same purpose as the basic unit of integration testing for this approach as the thread served as the basic unit of test planning for the Deutsch approach.
- The function tests, in which a time function accepts a time sequence of inputs, can be associated with each time function.
- The single object tests must test each of the state transitions (or paths of state transitions) identified in the graph of functions. This must occur for all of the state transitions for each object (e.g., see Figure 2).
- The multiple object tests will force all of the state transitions in the coordination function.

- The exception testing tests all of the branches of processing which were added to detect and recover from the exceptions identified during the latter stages of the design phase.

The following strategy for test sequencing is suggested:

1) Modules which perform the required transformations, and methods used to access each of the data objects can be tested either top down or bottoms up, according to the preferences of the testers.
2) Modules and data objects can be tested concurrently with the top down testing of the tasks. These test should be performed first on the host, where a friendly debugging environment exists, and then moved to the target machine where the testing is repeated.
3) To test the tasks on the host, a special test tool called a Functional Operating System (FOS) will be required. The FOS is a program on the Host machine which provides all of the operating system services supplied by the target operating system.
4) The final load tests will have to be performed on the Target architecture only, as the Host processor does not usually have the capacity to perform them.
5) Increments are defined for development. Each increment will go through the same sequence of module and task development on the host, integration testing on first the host and then the target architecture. The definition of the increments must trade off two points of view: do first things first, so as to minimize the required scaffolding software; and early resolution of the critical issues (e.g., if track processing has been identified as risky, then it should be done early so there is time to fix it if needed).

It is noted that the methods described above can be used effectively as a requirements/design verification and validation method for software not developed using the Requirements Driven Design approach. This is a consequence of the fact that the methods describe invariants of the system (i.e., behavior which must be satisfied regardless of design implementation, and allocation of required processing and data onto design elements). In fact, it is recommended that the Requirements Driven Design methods and tools be used in an Verification role previous to first use as the requirements/design methods for a critical software project in order for the personnel to gain confidence in the methods and tools.

4.0 CONCLUSIONS

It would not be totally unfair to characterize the approach presented above as merely:
- adapting the approach documented by Deutsch to apply to an improved requirements/design model, and thus to eliminate its deficiencies;
- extending the approach to address host/target testing problems; and
- incorporating the "folklore" from the state of the practice.

All of the deficiencies of the Deutsch method identified in Section 2.4 have been addressed except one -- the lack of tools. The methods described above are currently supported by tools. The DCDS tools developed by TRW under contract to the U.S. Army capture the requirements/design information, and perform substantial consistency/completeness analysis on this information. A Test Specification Language is used to document the information developed using the methods described above. A second set of tools to support this approach is currently in development at the **Ascent Logic Corporation**. Readers desiring more information about these tools should contact the author.

The expected results of using the methods described above are substantial. First, the process of defining a test plan traceable to requirements and design is now constructive (i.e., paths of the system behavior graphs are mapped onto tests) and understandable, rather than the product of "black art". Second, use of the methods are expected to increase productivity (i.e., the current effort to extract thread and object test cases from the requirements will be eliminated).

REFERENCES

1. M. Alford, "SREM at the Age of Eight", IEEE **Computer**, April 1985

2. M. Deutsch, **Verification and Validation: a Practical Approach**, J. Wiley and Sons, 1979

3. G. Booch, **Software Engineering with Ada,** The Bejamin/Cummings Publishing Company, Inc., 1983

AUTOMATED SOFTWARE TESTING — CASE STUDIES

E. Uren, E. Miller, J. Irwin

Software Research, Inc.
625 Third Street
San Francisco, CA 94107-1997

Abstract: Typical Software Research projects are described and numerical results from these projects are given. Levels of productivity are very high provided that a significant level of mechanization can be obtained. SR's use of specialized software tools is described in detail.

Confidentiality Note: We have to keep the names of clients confidential — this is often a main condition of our work — and have disguised the project summaries extensively. However, in all cases the statistics and the effort levels are reported accurately, as is the general type of product.

INTRODUCTION

We have found four major patterns in the work we are called on to do for clients at the "hi-tech" end of our business. The patterns repeat often enough that we think it will be interesting to current and potential clients to see what the numbers are, so that they can compare themselves with others.

In addition to these patterns, there is substantial common ground across these project types.

The typical situation is that a vendor has a product such as a compiler or operating system under development. The vendor is interested both in detecting errors in the current release or version of the product and in having a procedure for detecting errors. The procedure should be mechanized and should be as simple as possible so that when errors are repaired, the entire product may be retested economically, (this latter procedure is called regression). This will enable the user to verify that the errors have indeed been corrected and that no new errors have been introduced during the repair process.

To detect the errors, a test suite is constructed and since the customer is very eager to see the results of the testing the customer expects the test suite to be applied to the product under study during development of the suite itself. Typically, SR will agree to do this and also to accomodate their eagerness, SR usually sets up electronic mail so that they may get "instant access" to the latest "news " about tests applied and errors detected.

Development of the mechanized procedure for running the test suites was considered to be a process which was unique to each project because environments and test objects appear, at first glance, to be so different. However, as experience with the projects increased, however, it became clear that a general purpose tool could (and should) be constructed.

Project Classes

Test Suite Development: In this category, our purpose is to build the customer a suite for a fixed product. In early discussions with the potential customer, a decision is usually made whether there should be full validation or whether a touch test suite will suffice. A touch test suite consists of a set of programs that collectively exercise all language at least once. Smaller than a full validation suite, it is also far cheaper. It is a compromise between size and required complexity (that one might expect in a full validation), and thoroughness.

Comprehensive Product Testing: In this category, the purpose is to build and apply a complicated mechanized set of tests. The thrust of these projects is to develop a set of tests which provide as complete functional coverage as possible of the product under test. Since the product tends to be complex, the advantages of mechanizing the process of applying the tests become more significant. Consequently, there is considerable effort devoted to constructing the mechanical methods of application. The mechanization process goes hand-in-hand with organizing the tests themselves. This organization in and of itself proves to be a very powerful tool for analysis of the weak and strong components of the product. Generally, the grouping of individual test cases in the suite is oriented towards major functions of the product, and an accumulation of failed test cases in a group will provide a clue to product developers as to how to repair the errors so detected.

Detailed Technical Testing: In this category, we classify compiler or operating system testing. Compilers and operating systems are the foundation upon which which most development work is constructed. Typically, they are widely distributed with the computer hardware, and probably to most programmers, are seen as part of a complete package which happens to comprise both hardware and

software. The acceptance of the hardware by the user is indeed "masked" by the "appearance" and performance of the software. Consequently, the vendor considers it crucial to be as fully informed as possible about what these systems can and can not do.

Validation Testing: In this category, we have another type of critical product. Some software products have a particularly important dimension of criticality since they control medical devices which themselves have important impact on the management of health care. They can be of different orders of complexity. At various levels, they provide data used to analyze an individual's state of health and treatment can be prescribed on the basis of results of these systems. Not only do these systems affect human life and the quality of human life, they are also subject to regulation by the Federal Government and they operate in a domain wherein liability assumes a greater and greater importance.

TEST SUITE DEVELOPMENT

Environment Test Suite

Under contract to a foreign company, which was in turn under contract to a (foreign) government agency, SR developed a comprehensive validation suite for substantial extensions to the Unix System V validation suite. This test suite was the first to address validation of an environment, and was targeted to an environment designed for portable common tools.

SR developed the following during about 4 effort-months:

- 157 self-checking test programs.
- 471 tests of 175 commands, calls, drivers and functions.
- Special control program.

The special control program, in addition to executing the tests, reports incrementally on the progress of a group of tests in terms of the pass/fail ratio.

SR provided onsite installation support.

The initial application uncovered 24 errors.

PL/I Touch Test Suite Development

SR developed a touch test suite for the LPI/PL-1 subset G compiler for a major US vendor. This suite tests 204 features of the language. The touch test suite consists of 11,167 lines of PL/I code in 165 programs (and two auxiliary files for one of the programs), 26 scripts, two automated test scripts and 164 baseline files which have been validated manually.

Included in the suite (but not necessary for "functional coverage") were 8 programs from a widely available PL/I G-subset textbook.

An important feature of the suite was that it was set under SMARTS control (Software Maintenance and Regression System -see Reference 2), so that regression could be just about as automatic as one wished using the two automated test scripts. This is the generalized regression control system referred to previously. The package also included scripts to simplify compiling, loading and executing the test cases should one not wish to use SMARTS.

The programs in the test suite were not self-checking. Instead, SR used another approach. First, all the test output results were accumulated in "baseline" files. Then SR validated the contents of the files to ensure that the contents were correct for the test object in its current state of development. Thus while most of the output was correct because the test object had no errors, in some (perhaps many) instances, the test results were the results of errors. This output was still incorporated into the baseline files. Output from subsequent executions of the test suite during regression could then be compared with the baseline files quite simply using "diff". Clearly, any differences reflected changes in behavior of the test object, which is exactly what the tester was looking for in a regression situation. The goal of the regression test is that correct output remain unchanged and incorrect output be changed, presumably for the better. Under SMARTS, this is always the approach taken.

During development of the suite, SR discovered thirty-one problems serious enough to warrant reporting in formal error reports. The cost of discovery of each error was $500 alone, IGNORING the fact that a test suite, a set of baseline cases and a regression system were delivered.

COMPREHENSIVE PRODUCT TESTING

Assembler Test Suite and Control Program

In this project for a major US vendor, the purpose was to develop both a comprehensive test suite for a new macro assembler and an automated way to apply it, and also to apply the suite to the assembler using the automated procedure. The automated procedure was to allow the user to "browse" through the test set, run individual tests or groups of tests, compare results of runs with previous results, and maintain statistics.

610 test programs were developed and applied, detecting 160 defects. After the client made some revisions to the macro assembler, the tests were re-applied.

The automated procedure developed in this project can be used in other regression situations on other projects. Thus the client has, as a by-product, a new universal tool for regression. Should the client develop another product which requires regression, it is only necessary to

define the structure of the tests in a control file, construct the tests and validate the output of the first application in what are called baseline files.

From SR's point of view, development of the automated procedure led SR to completely generalize the process of automated control and produce the SMARTS package. SR had developed so many control programs from scratch that the need was evident. The general purpose qualities of this program were the final step.

To develop the tests and the regression system took one effort-year. The regression system contains 5,400 lines of control file.

System Test Mechanization Project

A major portion of this project was to develop an automated regression system for a client with a quite large, extremely sophisticated, highly user-interactive, product. The product runs on Sun workstations and was designed using object-oriented principles. Interaction with the system used a keyboard as one might expect, but far more emphasis and use was focussed on the use of a mouse. The client had invested substantially (more than 8 figures) in developing the system. There have been many releases and a few versions have been in beta-test for about a year.

Working with the client's programming staff, SR developed a system, integrated into the client's program, which captured key-strokes and mouse movements. Tests may be captured during their first application and played back. This together with the regression system allows the client to automate most of the testing procedure.

Thus the client has an accurate detailed record of what functions the test performed, and there is also a procedure for modifying these test playback files so that test variants may be constructed economically. Performed under control of the regression system, SMARTS, comparisons with the results of prior tests may be made and statistics maintained.

SR developed 210 tests in the process. There were about 650 sub-tests included. In the process of constructing and applying the tests, SR discovered 22 errors.

DETAILED TECHNICAL TESTING

Unix System Testing

The purpose of this project was to apply previously constructed touch tests for Unix utilities, to extend touch tests for the system interface, construct library function tests and to develop and apply tests to assess the computer's kernel-level stability.

The client was a major U.S. computer manufacturer whose new computer model was at about the final stages of hardware testing.

To assess the computer's kernel-level stability, SR developed a suite of self-checking tests of CPU, memory, disk I/O, serial communications and a CapBak(tm) session simulating a "typical" terminal user. The tests in the stability test suite were parameterized as to size and could be executed in different mixes. Thus instability in terms of test failure or degraded response time was observed in terms of the size and mix of the load on the machine.

There were 141 tests of utility functions testing 665 switches and combinations of 195 Unix base commands, 66 tests of the system interface (with 182 sub-tests) and 87 tests of library functions (with 150 sub-tests). Fifty-one anomalies were detected of which thirty-one proved to be errors in the software.

The stability tests consisted of load tests for the CPU, Disk, communication channel, keyboard and memory. These parameterized tests could be run independently or as a mixture. Test run times ranged from 8 seconds through 37 hours. Five anomalies were detected in this portion.

The control program for the stability tests contained enough general characteristics to be considered the "seed" for SMARTS.

Xenix Touch Testing

Xenix V Software System Test Project

The purpose of this project, for a large U.S. computer manufacturer, was to validate the operation of Xenix V on a variety of the manufacturer's machines. The test suite developed was to be applied to a number and variety of machines in single-user mode, linked together, and to different versions of the operating system.

SR developed touch tests for all XENIX utilities including base commands, software development system commands, and text processing commands. SR also planned and developed full validation tests for the following software device drivers:

Memory Managment Unit
80287 Co-processor
CPU
Serial Port (including Multiport)
Parallel Port
Console
Clock
Timer

The statistics for these tests were too voluminous themselves for this document. Suffice to say there were considerably more than 1000 tests. Another point to note is that this project was the last that SR had do without the benefit of either a control program or a regression system. The lack of mechanization meant that running the tests and documenting them accurately consumed substantial

manual resources.

SR detected 94 errors in the first application of the tests and 50 in the first of three regressions.

VALIDATION TESTING

Patient Data Management System QC

The purpose of this project was to test a system which permitted medical patients to accumulate periodic readings of certain biological variables without visiting a medical facility. The patient used a portable recording device for this. At fairly regular intervals, the patients' records could be offloaded to a personal computer software system for analysis and storage. The client was a major US supplier of medical equipment. SR developed a system for maintaining the system under strict configuration control; and developed a system for testing new releases thoroughly and economically before distribution to the client's customers. Thus a new release must proceed both under careful configuration control and under examination under the same test situations as previous versions.

To accomplish this, SR developed a set of tests under automatic keystroke capture and playback conditions (using SR's CapBak(tm) system — Reference 1), developed a further set of functional tests, established a Software Incident Reporting System for tracking errors, placed master copies of the code under Unix SCCS control and systematized procedures for making changes to the code smoothly. SR performed detailed coverage analysis on each version of the code using the automated test suite to ensure that the test suite tested the code thoroughly. A variety of errors and anomalies were discovered and repaired as part of the project effort.

Quality Control Printer Testing

This client was under contract to a major U.S. medical equipment supplier to produce hardware and software which would permit the use of a printer as a Quality Control device by producing reports derived from data accumulated in some medical equipment. This medical equipment is, ultimately, the equipment whose operation needs to be checked periodically. When not performing this function, the printer would serve as a printer.

SR's task was to test the software which checked the operation of the medical equipment. SR tested it in a number of ways.

First, SR performed a formal inspection and review of the code, finding 71 anomalies at the modular level and 65 at the system level. Second, it developed a set of 38 functional tests which were applied to instrumented versions of the code compiled on a PC and determined that the coverage levels reached very high levels for both branch coverage and system coverage. SR used TCAT/C and STCAT/C (References 3 and 4), another set of standard tools for this step of coverage analysis.

Third SR produced another 27 tests and modified the original 38 to have increased numbers of readings. This process required that SR also develop a method for generating test cases.

Fourth, SR developed a validation system for test cases. This system consisted of three parts. One part used the same input as the code under test and computed results and the coordinates of where the results should be plotted on graphs which could be part of the output. Another part extracted the results of the test code's output and presented this data in the same format as the first program's output. A third program could compare the output of the first two parts. That this process worked correctly was formally validated on the output of a sample.

Next, SR placed the test suite under SMARTS control for regression purposes. Regression could not be as automatic as one would like due to fact that the code under test, requires that switches be set and a button pushed before the code executes using the data. Nevertheless, the baseline cases were validated.

Finally, SR applied the test cases to two releases of the software.

Another twenty-two error reports were written; 14 of these were judged serious.

By the end of the project, the defect rate on the latest version was 11 or about 4/KLOC. Of these 11, about 5 were still serious or 2/KLOC. Thus, the complete error detection process reduced the error detection rate by a decimal order of magnitude.

REFERENCES

1) D. Casey, L. Ceguerra, C. Cox, J. Irwin and M. Morrison, "User's Manual for Capbak, Keystroke Capture and Playback System", Release 2.0.9, Technical Note TN-1075, Software Research, Inc, San Francisco, Ca., May 1987.

2) D. Casey, C. Cox, J. Irwin and E. Qualls, "User's Manual for SMARTS, Software Maintenance and Regression Test System", Release 4.1, Teechnical Note TN-1281, Software Research, Inc., San Francisco Ca 94107, May 1987.

3) R.W. Erickson, H. Nguyen, E. Miller, J. Irwin, D. Casey and L. Ling, "User's Manual for TCAT/C (PC Version)", Technical Note RM-1100/2, Software Research Associates, San Francisco, Ca., September 1984.

4) H. Nguyen, "User's Manual for S-TCAT/C PC/DOS Version", Release 4.7, Technical Note RM-1266/1, Software Research Associates, San Francisco, Ca., August 1986.

EXPERT SYSTEM VALIDATION : ISSUES AND APPROACHES

Edward F. Miller
Software Research, Inc
625 Third Street
San Francisco, California 94107-1997
USA
Tel. : (1) 415 957 1441

Abstract : As expert system (ESs) technology matures, and as more and more ES software appears on the market, the time to ask hard questions comes nearer and nearer : What about the quality of an ES ? How does one know that the result produced by an ES is correct ? What indications are there that an ES might be giving incorrect information ? What can make an ES fail ? What can be done to certify ES software ?

PARALLELS WITH THE PAST

There can be little doubt that ES's represent a tremendous advance in software technology. The ES approach, combined with the unique set of applications of ES's - which have very great technical appeal - virtually assures that the 1990's will be an ES decade. There are, however, close parallels between the current situation in ES technology and the early 1970's love-affair with software engineering (SE). Many of the rosy promises that were being made by SE technologists of that era concerning formal validation procedures (to take one example) have a resonant similarity with some of the promises - and expectations- of ES technologists of today (Ref 1,2).

Great difficulty can ensue if the ES community too quickly concludes that ability to build a system that appears to have quality behavior on a few cases means that it will have quality behavior in all instances.

FAILURE MODE ANALYSIS

Here is a preliminary analysis of the possible failure modes in an ES. Software Research's (SR's) approach to ES validation (ESV) is described (Ref. 4).

o **Incorrect user input ("Pilot Error")** : The user thinks he said X and he really said Y ; he believes the opposite but he is wrong. The ES-produced answer is wrong not because of a fault in it but because of a fault or inconsistency on the part of the user.

Full validation of the user's input state is the solution, but the overhead for this computation - let alone what is involved in it - is not well understood.

SR's ESV method : little possible protection, except for careful user testing for resistance to error inputs. SR's early experiments indicate that the use of mechanically generated test suites, combined with an automated session driver (i.e. the equivalent of a software test bed), is effective.

o **Incorrect rule in the ruleset** : In this case the "expert" has declared an incorrect fact - probably an intermediate one - and doesn't know it. The ES works correctly but gives the wrong (but 100% self-consistent) answer. The problem is, the rule is invalid with respect to "outside real-world truth".

SR's ESV method : development of suites of automated test sessions generate, as much as possible, one each of every possible equivalence class of output result. Our early experience suggests that, with care in choosing limit and unusual cases, the most common access to an (the original ?) expert is essential to create the ES test baseline.

o **Incorrectly stated rule in ruleset** : Here there are three cases : missing rule, extra rule, and wrong rule. Coverage analysis (percent of rules touched) can detect most of the wrong rule cases. If the input state sequences are sophisticated enough, they can also detect 50%-70% of the missing rule cases. Extra rules are noted by their non-use after coverage checking.

Preliminary estimates suggest life-cycle defect rates in the range 20/kRules to 30/KRules.

SR's ESV method : Measurement of the LRI (Logical Rule) for popular ES languages such as Lisp or Prolog appears to be straightforward, although some coercion of certain interactive features is sometimes problematical. Some ruleset bugs are very difficult to find and careful inspection methods must be used.

o **Rule reduction problem** : The error is in the ES shell or other system software component. Conventional rules of SE production apply : 50 defects/KLOC, detectable inspection, functional test, and structural (unit and system level) test.

SR's ESV method : Use conventional convergence testing methods based on comprehensive functional and structural exercise, but only if the source versions of the ES system component is available. In the future, SR advocates using standardized test suites for the popular logic programming languages.

o **Validation of outputs** : The issue is similar to that in proving complex programs - ones with many and complex paths, not necessarily those with many levels necessary to check that combinations are tried. But combinatorics prevents this unless ways to factor rulesets along structural lines can be found.

SR's ESV method : SR estimates that about a 100:1 reduction in overall path-oriented complexity is possible by linearizing methods. How to accomplish full validation for a complex, real-world system is not fully understood. Preliminary application of SR's **MetaTest** system suggests there is some hope of unraveling the logic flows and factoring them efficiently.

COMMERCIAL ES VALIDATION SERVICE (ESVS)

Many modern approaches to conventional software testing are nearly directly applicable to ES quality certification. The parallels have permitted release of SR's new **Expert System Validation/Testing Service**, based in part on use of advanced software validation methods.

SR's **CapBak(tm)** session capture and playback software, combined with the **SMART** regression test system, allow each ES test case to be run, with test results comparaison, nearly 100% automatically. Regression testing is fully automatic.

The completeness of the tests can be assessed with SR's new **TCAT/Lisp** and **TCAT/Prolog** test completeness assessor systems. (SR's new ES test tools will be available in late 1987).

The costs of SE validation services (SEVS) are $15K-$30k/KLOC. Because of the increased complexity and subtleness of the defects, ESVS costs are presently over $100K/Krules. After SR gains further experience, we estimate the costs to be in the $65K-$85K/KRule.

SUMMARY

From the QC perspective and from the experience of applying effective QC methods to a wide range of software, ESs are, ultimately, built out of software - the same kind of software with which we are all generally familiar. While ESVS costs are significantly higher than for SEVS, they clearly indicate ESVS' viability.

REFERENCES

1. D.G. Bobrow, S. Mittal & M.J. Stefik : "Expert Systems : Perils and Promises", Communications of the ACM, September 1986

2. D. Barstow : "Artificial Intelligence and Software Engineering", Proc. ICSE 9, Monterey, California, 1987.

3. R.L. Enfield : "The Limits of Software Reliability", MIT Technology Review, April 1987.

4. E.F. Miller & W. Howden : Software Testing and Validation Techniques, 2nd Edition, IEEE Computer Society Press, 1984.