CONFERENCE PROCEEDINGS

Quality Week 1990

15-18 May 1990

Golden Gateway Holiday Inn
Van Ness Avenue
San Francisco, California

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

Phone: (415) 957-1441 — FAX: (415) 957-0730

Paper 1-2

SOFTWARE RELIABILITY ENGINEERING — A VISION FOR THE 1990s

Mr. John Musa
AT&T Technologies

John Musa is Supervisor of Software Quality at AT&T Bell Laboratories, Whippany, NJ, where he has managed a wide variety of software projects. His technical background and interests include software reliability, software engineering, and human factors. He has published more than 40 papers and is principal author of the pioneering book "Software Reliability: Measurement, Prediction, Application". He is an international leader in software engineering and is a Fellow of the IEEE, cited for "...contributions to software engineering, particularly software reliability".

SOFTWARE RELIABILITY ENGINEERING — A VISION FOR THE 1990s

JOHN D. MUSA
AT&T BELL LABORATORIES
WHIPPANY, NJ 07981

OUTLINE

→ 1. INCREASING IMPORTANCE

2. BASIC CONCEPTS

3. SOFTWARE RELIABILITY ENGINEERING AND SOFTWARE LIFE CYCLE

4. ACHIEVING THE VISION

WHAT IS SOFTWARE RELIABILITY ENGINEERING (SRE)?

APPLIED SCIENCE OF

- PREDICTING
- MEASURING
- MANAGING

RELIABILITY OF SOFTWARE-BASED SYSTEMS
TO MAXIMIZE CUSTOMER SATISFACTION

EVOLUTION OF SOFTWARE ENGINEERING — STAGES

1. FUNCTIONAL
2. SCHEDULE
3. COST
4. RELIABILITY

WHY IS SRE IMPORTANT?

1. INTENSE AND INCREASING INTERNATIONAL COMPETITION

2. INFORMATION PROCESSSING QUALITY CRITICAL TO COMPETITIVENESS

3. CUSTOMER-PERCEIVED QUALITY AND RETURN ON INVESTMENT
   (TOP THIRD 29%, BOTTOM THIRD 14%)

4. QUALITY MULTIDIMENSIONAL, ONE-DIMENSIONAL CONSERVATISM
   INTOLERABLE

5. MEASUREMENT ESSENTIAL

OUTLINE

1. INCREASING IMPORTANCE
2. → BASIC CONCEPTS
3. SOFTWARE RELIABILITY ENGINEERING AND SOFTWARE LIFE CYCLE
4. ACHIEVING THE VISION

THE LANGUAGE OF SRE

FAILURE: DEPARTURE OF PROGRAM OPERATION FROM *CUSTOMER* REQUIREMENTS

FAULT: DEFECT IN CODE THAT CAUSES FAILURE

EXECUTION TIME: TIME PROCESSOR EXECUTES PROGRAM

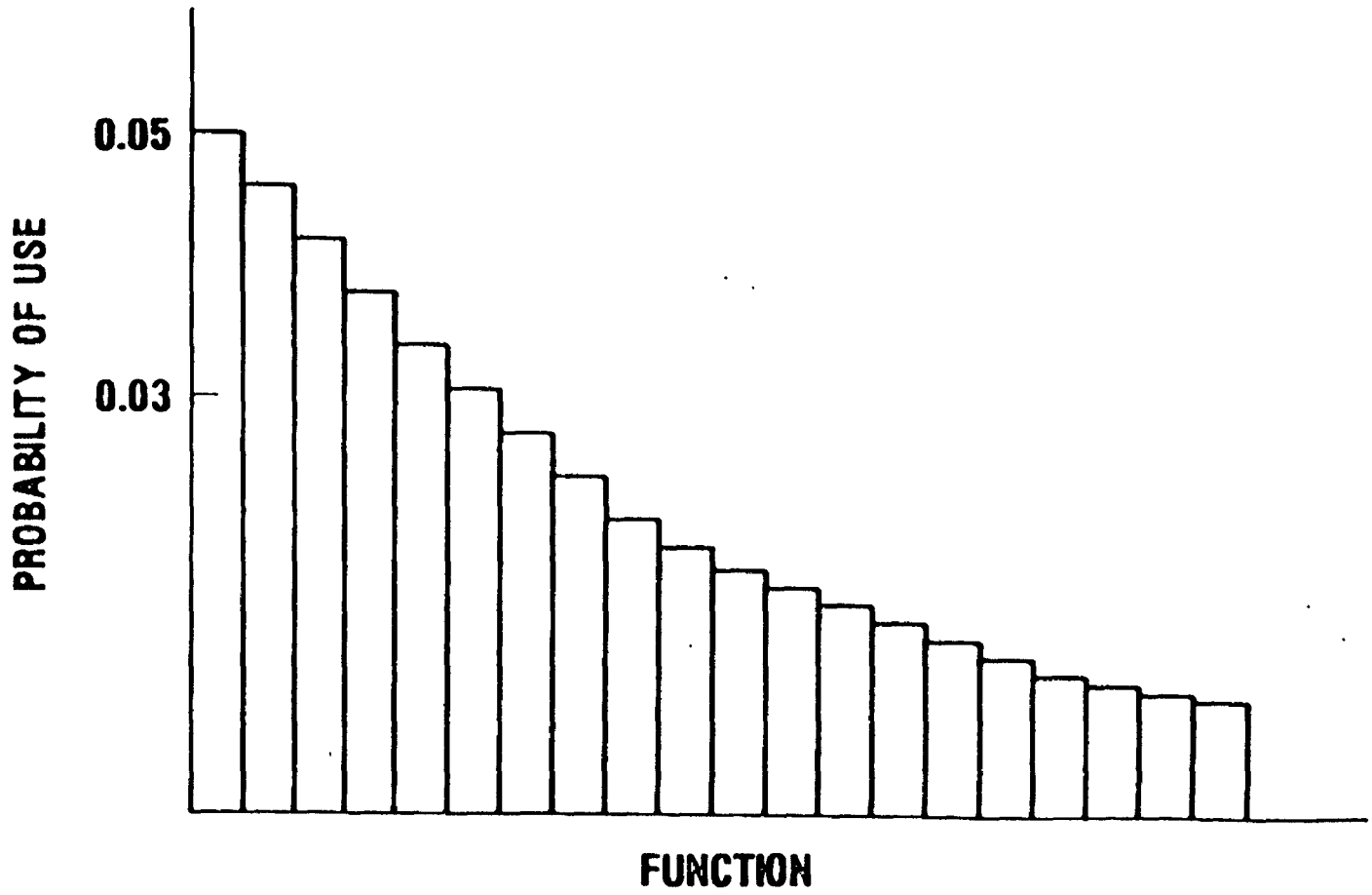
ILLUSTRATION: 1000 HR

FAILURE INTENSITY: FAILURES PER UNIT EXECUTION TIME

ILLUSTRATION: 6 FAILURES/1000 HR

OPERATIONAL PROFILE: SET OF FUNCTIONS AND ASSOCIATED PROBABILITIES OF OCCURENCE

OPERATIONAL PROFILE

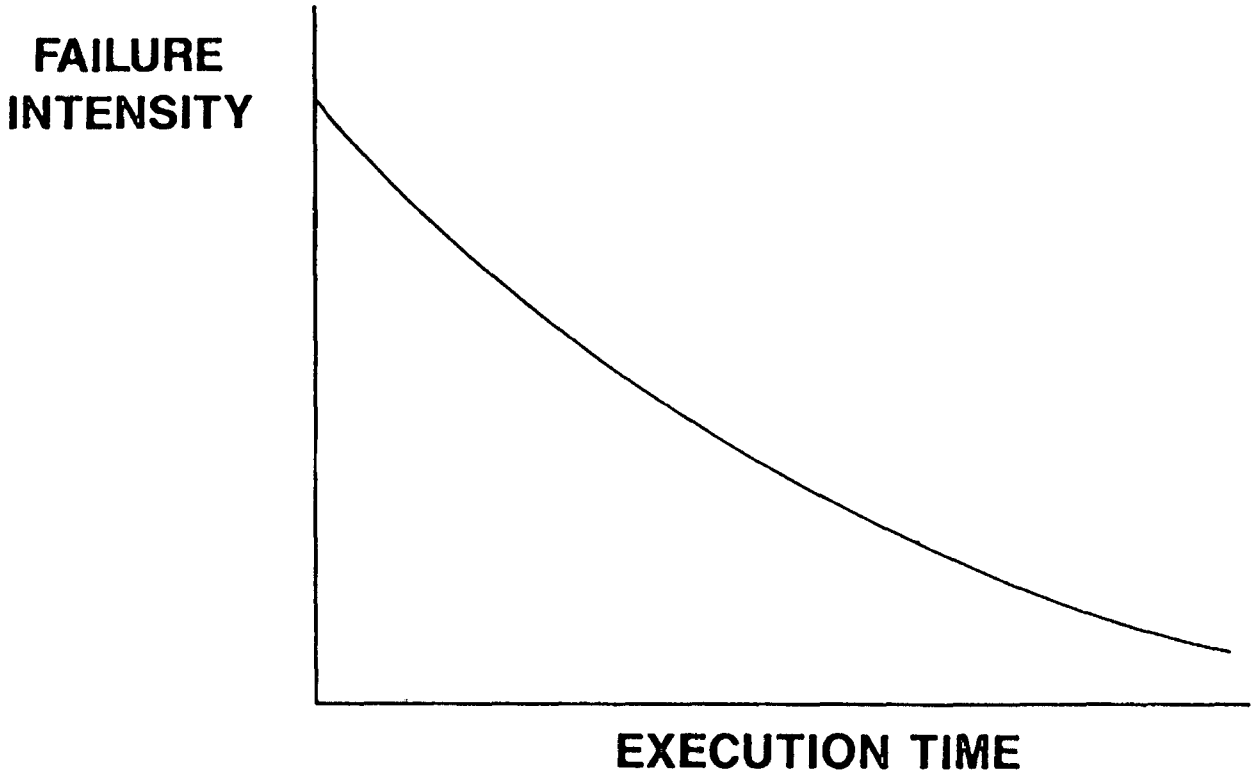


T19

EXECUTION TIME MODELS

1. EXECUTION TIME COMPONENT
   A. EXECUTION TIME
   B. PARAMETERS DEPEND ON PRODUCT, DEVELOPMENT PROCESS, ENVIRONMENT
2. CALENDAR TIME COMPONENT

EXECUTION TIME COMPONENT-APPEARANCE



T23

OUTLINE

1. INCREASING IMPORTANCE
2. BASIC CONCEPTS
3. → SOFTWARE RELIABILITY ENGINEERING AND SOFTWARE LIFE CYCLE
4. ACHIEVING THE VISION

SRE IN DIFFERENT PHASES OF SOFTWARE LIFE CYCLE

1. DEFINITION
2. DESIGN AND IMPLEMENTATION
3. VALIDATION
4. OPERATION AND MAINTENANCE
5. PROCESS IMPROVEMENT

DEFINITION

1. SELECT MIX OF QUALITY FACTORS, INCLUDING RELIABILITY OBJECTIVE (BY FAILURE SEVERITY CLASS IF APPLICABLE) — INVOLVES RELIABILITY PREDICTION

2. DETERMINE OPERATIONAL PROFILE

DESIGN AND IMPLEMENTATION

* 1. SELECT ARCHITECTURE AND ALLOCATE RELIABILITIES

* 2. DESIGN DEVELOPMENT PROCESS

3. CERTIFY REUSED AND ACQUIRED SOFTWARE

4. USE OPERATIONAL PROFILE TO FOCUS EFFORT

* 5. MANAGE DEVELOPMENT PROCESS

   A. COMMIT CONTINGENT RESOURCES
   B. REDESIGN PROCESS
   C. RESPECIFY REQUIREMENTS

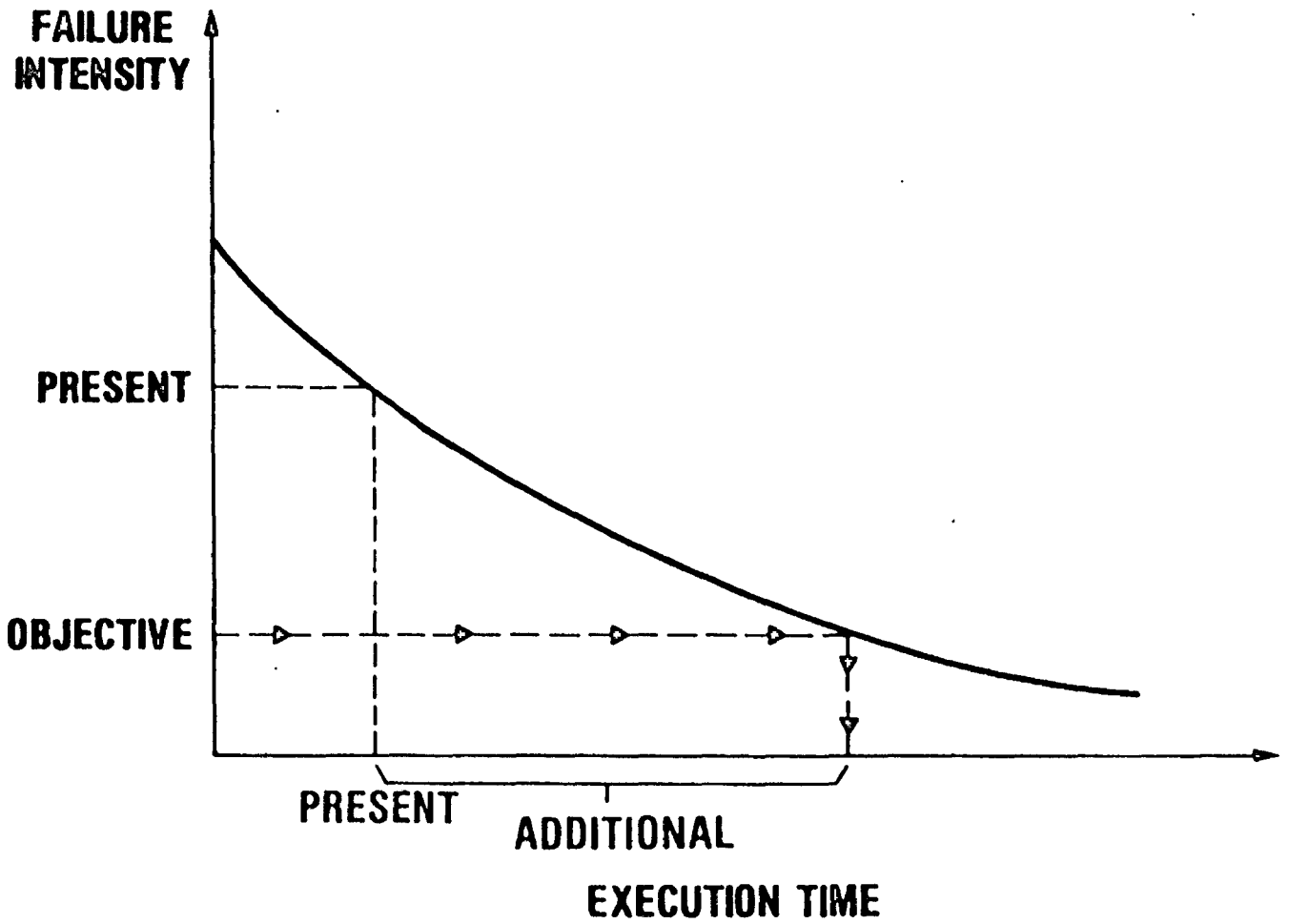
*MAY INVOLVE RELIABILITY PREDICTION

VALIDATION

1. USE OPERATIONAL PROFILE TO DRIVE TESTING

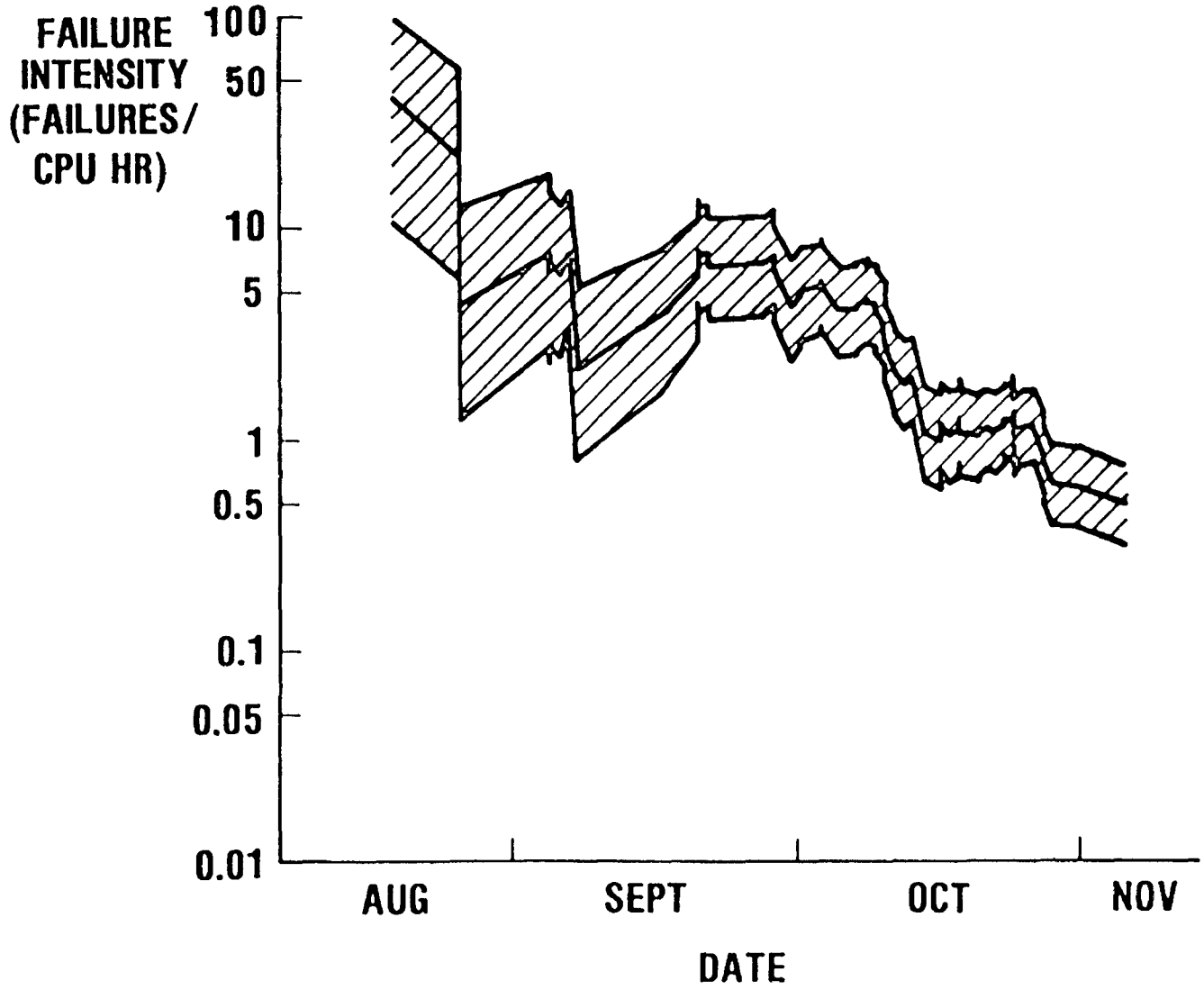
2. MEASURE RELIABILITY TO GUIDE RELEASE

EXECUTION TIME COMPONENT-PREDICTIONS



T30

DETERMINE CURRENT RELIABILITY STATUS

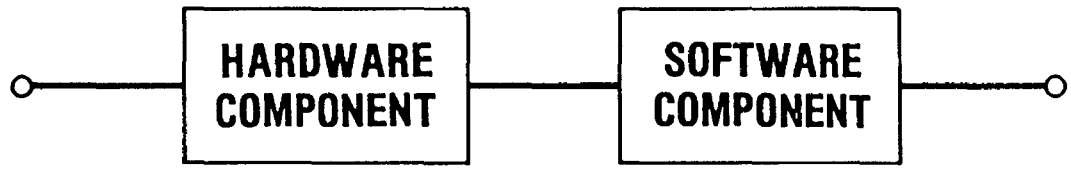


T52

OPERATION AND MAINTENANCE

1. PLAN FIELD SUPPORT
2. MONITOR RELIABILITY
3. MANAGE IMPACT OF SOFTWARE MODIFICATION ON CUSTOMER OPERATION

ALLOCATION OF RELIABILITIES TO COMPONENTS



$R_H = 0.99$

$R_S = ?$

$R_S = 0.96$

$R_{SYS} = 0.95$

PROCESS IMPROVEMENT

1. USE SRE TO EVALUATE CANDIDATES

2. MOTIVATION PROBLEM - ORGANIZATIONAL?

OUTLINE

1. INCREASING IMPORTANCE
2. BASIC CONCEPTS
3. SOFTWARE RELIABILITY ENGINEERING AND SOFTWARE LIFE CYCLE
4. ACHIEVING THE VISION

ACHIEVING THE VISION

1. APPLIED RESEARCH
   - A. PREDICTION
   - B. FAULTS REMAINING AND DESIGN ERRORS, FAULTS DETECTED IN DESK-CHECKS OR CODE WALKTHROUGHS
   - C. UNIT TEST
   - D. TEST STRATEGY COMPENSATION
   - E. ESTIMATION
2. TECHNOLOGY TRANSFER

SOFTWARE RELIABILITY PREDICTION

$$\text{FAILURE INTENSITY} = \text{FAULT EXPOSURE RATIO} \times \frac{\text{PROCESSING SPEED}}{\text{OBJECT PROGRAM SIZE}} \times \text{FAULTS REMAINING}$$

RESOURCES

1. BOOK "SOFTWARE RELIABILITY: MEASUREMENT, PREDICTION, APPLICATION" BY MUSA, IANNINO, OKUMOTO; McGRAW-HILL

2. COURSES

3. VIDEOS: COMPUTER SOCIETY, SOFTWARE ENGINEERING INSTITUTE

4. SOFTWARE TOOLS

5. CONSULTING

WHERE IS SRE NOW?

1. EXPLOSIVE INTEREST
   - A. ADVANCES IN COMPUTERS, SPECTRUM, JOHOSHORI, SOFTWARE
   - B. IEEE SUBCOMMITTEE ON SOFTWARE RELIABILITY ENGINEERING
   - C. U.S. CONGRESS - HOUSE COMMITTEE ON SPACE, SCIENCE, AND TECHNOLOGY
   - D. CONFERENCES
   - E. EEC SCOPE
   - F. JAPANESE, ITALIAN TRANSLATIONS OF BOOK
2. PILOT APPLICATIONS (SAMPLING)
   - A. AT&T: 5ESS®, BILLING SYSTEM, TMAS
   - B. HEWLETT-PACKARD
   - C. JPL
   - D. CRAY RESEARCH

SOFTWARE RELIABILITY ENGINEERING

Resources for Further Exploration

1. *Book*

Musa, J.D., Iannino, A., Okumoto, K. *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill, 1987, 635 pages, ISBN 0-07-044093-X, $52.95

Can be ordered by calling 1-800-338-3987

TABLE OF CONTENTS

2. *Video* (self-study)

SOFTWARE RELIABILITY MEASURES: GUIDING SOFTWARE DEVELOPMENT FOR QUALITY AND COST EFFECTIVENESS
by John Musa, AT&T Bell Laboratories

Price: members $99, nonmembers $119
IEEE Computer Society Press
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
714-821-8380
FAX 714-821-4010

3. *Software Reliability Courses and Consulting*

   1. *Within AT&T (and associates):*

      Bill Everett
      AT&T Bell Laboratories
      200 Laurel Avenue, Room 3K153
      Middletown, NJ 07748-4801

      201-957-6847
      FAX 201-957-7545
      E-MAIL att!whuxr!wwe

   2. *Outside AT&T:*

      Frank Ackerman
      Institute for Zero Defect Software
      85 Poplar Drive
      Stirling, NJ 07980

      201-604-8701
      FAX 201-604-8702
      E-MAIL att!izdsw!afa

4. *Software Reliability Programs*

There are two sets of programs developed at AT&T Bell Laboratories to estimate and predict software reliability.

The first set is a public domain program written in FORTRAN. It implements the basic execution time model essentially as described in the book, "Software Reliability Measurement, Prediction and Application" by J.D. Musa, A. Iannino

and K. Okumoto (McGraw-Hill 1987).

This program is available on 1600bpi magnetic tape (ASCII-coded) and can be obtained from:

Bill Everett
AT&T Bell Laboratories
200 Laurel Ave. - Room 3K153
Middletown, NJ 07748-4801
Tel. 201-957-6847

The second set consists of three UNIX® based programs called RELTOOLS that are available at a nominal ($300 as of 9/16/88, with free distribution and use rights throughout a company or institution) charge through AT&T's UNIX System Toolchest. The programs are written in RATFOR (a front-end processor for FORTRAN) and FORTRAN 77. The reliability programs themselves are:

1. RELTAB - Essentially the same as the public domain software except that it also implements the Logarithmic Poisson Model, can handle evolving programs, estimates confidence intervals more accurately, and uses the more up-to-date failure intensity rather than mean-time-to-failure quantities.
2. RELPLT - Does essentially the same thing as RELTAB but also provides a number of graphical plots of reliability parameters.
3. RELLOG - Converts from a "log format" to a format acceptable for RELTAB and RELPLT. RELLOG can usually be readily interfaced to failure reporting and logging systems.

The RELPLT program also requires you to have the "S"* data analysis package, and 'troff". All these are either standard UNIX offerings or are available through Toolchest.

UNIX is a registered trademark of AT&T.

* "S" itself is a very useful tool for doing statistical data analysis. If you do not want to acquire "S", you may want to stick with tabular output using RELTAB initially. After gaining some familiarity with using the tools, you can run RELPLT without "S", look at intermediate files "S.cmds" and "S.data" and write your own programs to generate plots using information from these files.

To order RELTOOLS through UNIX Toolchest:

1. Contact your AT&T Account Representative or call 1-800-828-UNIX to see if your company/institution already has a licensing agreement with Toolchest and find our who your "buyers" are so orders can be placed through them.

   In the Far East, contact:

   AT&T Unix Pacific Co., Ltd.
   No. 1 Nan-oh Bldg., 5th Floor
   2-21-2, Nishi-Shinbashi
   Minato-ku, Tokyo 105 Japan
   TEL: (03) 431-3305 (Japanese)
   TEL: (03) 431-3670 (English)
   FAX: (03) 431-3680
   TELEX: J34938 ATTUP
   uucp: upshowa!attup
   (TEL: 03-432-3544,
   login: upshowa, modem: 212a)

   In Europe, contact:

   AT&T Unix Europe Ltd.
   International House
   Ealing Broadway
   London W5 5DB, England
   TEL: +44 1 567-7711
   FAX: +44 1 567-2420
   TELEX: 914054 UNIXTM G
   uucp: {...!mcvax!ukc!}uel!uel

   If not, ask the above contact to send a copy of the licensing agreement. Execute and return with $100 registration fee and the names of people authorized to buy software (if you are a source licensee of UNIX System V, registration fee is waived).

2. Call (24 hr/day) Toolchest from US or Canada† at 1-201-522-6900 and logon with "guest." A menu will guide you in looking through a catalog for

† Programs available on tape elsewhere.

RELTOOLS, S, or other available programs. Order the software, it will be sent electronically to you, and you will be billed.

Toolchest software is licensed "as is" without technical support because of the low price. Toolchest software is also available under a lump sum sublicense for those who wish to do secondary distribution.

For more information call:

Software reliability programs: Bill Everett (see above)

UNIX Toolchest: appropriate contact above

Paper 1-3

FOUNDATIONS OF PROGRAM TESTING: CAN TESTED SOFTWARE BE TRUSTED?

Prof. Richard Hamlet
Portland State University

Prof. Richard Hamlet is professor of Computer Science at Portland State University, Portland, OR. He has been actively involved in many phases of program-testing research for almost 20 years. Currently, he is investigating the theoretical foundations of testing under a National Science Foundation research grant. Prof. Hamlet is a frequent speaker at technical conferences and symposia.

FOUNDATIONS OF PROGRAM TESTING

Can Tested Software be Trusted?

Dick Hamlet

Department of Computer Science
Portland State University

If first you don't succeed,
  Just try and try again.
And if you don't succeed again,
  Just try and try and try.
Useless, it's useless,
  Our kind of life's too tough...
Take it from me it's useless:
  Trying ain't enough.

— Bertolt Brecht
Three Penny Opera

SR Quality Week

Summary and Conclusions

- Foundations shaky
- No, it can't be trusted
- But we can improve fault-finding methods
- And we can stop kidding ourselves

Talk Outline

- ☐ I. Practical testing methods
- ☐ II. Critique of method comparisons
- ☐ III. Comparison using a valid standard
- ☐ IV. What should we do now?

Outline of Part I

Practical Testing Methods

- ☐ Structural (program-based)
  - — statement
  - — branch
  - — dataflow
  - — mutation
- ☐ Blackbox (specification-based)
  - — functional
- ☐ Partition testing

Statement Testing

Tests are required to execute each statement.

```
function Sqare(X: real): real;
  { compute the square }
  begin
    Sqare := X + X
  end
```

The call:

```
Sqare(2.0)
```

attains statement coverage, satisfies the specification comment, and does not expose the bug.

Branch Testing

Tests must force both alternatives of each branch.

```
function AbsMid(L,R: real): real;
{ distance from origin to the midpoint
  of L and R on the real line }
  var Pos: real;
  begin
    Pos := R - (R-L)/2;
    AbsMid := Pos;
    if R <= 0
      then
        AbsMid := -Pos
  end
```

The calls:

```
AbsMid(5,3);
AbsMid(-5,-3)
```

attain branch coverage, satisfy the specification, and do not expose the bug.

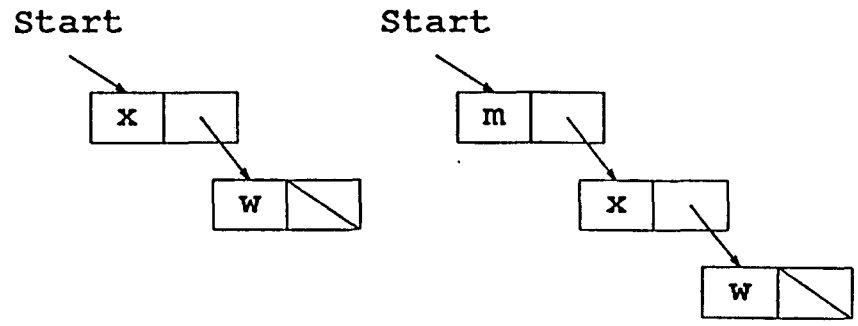
Dataflow Testing (Def-use)

The "def" control point for a variable is one where it is set; a "use" where it is referenced. A "def-use" test must follow one path from each def to each use without an intervening def.

```
type List =
  record Next: ↑List; Val: char end;

function ListMax(Start: ↑List): char;
{ maximum value in a nil-terminated
  list beginning at Start }
var Max: char;
    R: List;
  begin
    Max:=Start↑.Val;
    R:= Start↑.Next;
    while R.Next<>nil
      begin
        if R.Val>Max
          then
            Max:=R.Val;
        R:=R.Next↑
      end;
    ListMax := Max
```

Def-use, cont.

Calls on `ListMax` with arguments corresponding to the following lists:



attain def-use coverage, satisfy the specification, and do not expose the bug.

"Data Coverage" Testing (Mutation)

With a C compiler using Boolean short-circuit evaluation, the code:

```
if (Y<3 || Y>8) then ...
```

is not "data covered" by `Y` = 1, because the second alternative is not used and thus makes no difference to the result. (There is a hidden uncovered path.)

Although there is no hidden path, the value `X` = 2 similarly fails to data-cover `X + X`, because the alternatives like `4, X + 2, X*X, X**2,` etc., cannot be distinguished from the original.

Data coverage is achieved if any small change like this will be exposed by the test. That is, no slightly altered program would give all the same test results.

Data-coverage testing is also called ***mutation testing*** and an altered program is a ***mutant.*** A mutant is ***killed*** by a test when it gives a result different from the original program.

Data Coverage, cont.

Different mutation systems make different alterations, but they have in common that only one part of the program is changed at a time, and the changes are small. For example, the alterations might be legal variable substitutions in each expresssion.

```
type Ar = array [1..5] of char;

function Dups(Vals: Ar; N: integer):
                    boolean;
{TRUE if triplicates in Vals[1..N]}
var i,j,k: integer;
  begin
    Dups:=false;
    for i:=1 to N-2 do
      for j:=i+1 to N do
        if Vals[i]=Vals[j] then
          for k:=1 to i-1 do
            if Vals[k]=Vals[i]
              then Dups:=true
  end
```

One mutant would have for the last loop (substitute `j` for `i`):

```
          for k:=1 to j-1 do
```

Data Coverage, cont.

Test data corresponding to the array:

a
b
b
c
b

will not kill this mutant. However, adding the data:

b
b
x
x
c

does kill all variable-substitution mutants, but fails to expose the bug.

Blackbox Testing

From the specification, isolate natural classes of inputs for coverage.

In a system that should respond to file commands, say `CREATE`, `RENAME`, `COPY`, `DELETE`, each with file-name parameters, the commands define a partition:

> Two inputs are the same if they use the same command. This class is covered by four tests, one each of `CREATE`, etc.

The partition can be refined using parameters:

> For `CREATE`: no parameter (error); existing file (error); nonexisting file.

Further refinement could involve several parameters, a sequence of commands, etc.

Partiton Testing the Triangle Program

INPUT: three integers representing the length of a triangle's sides.

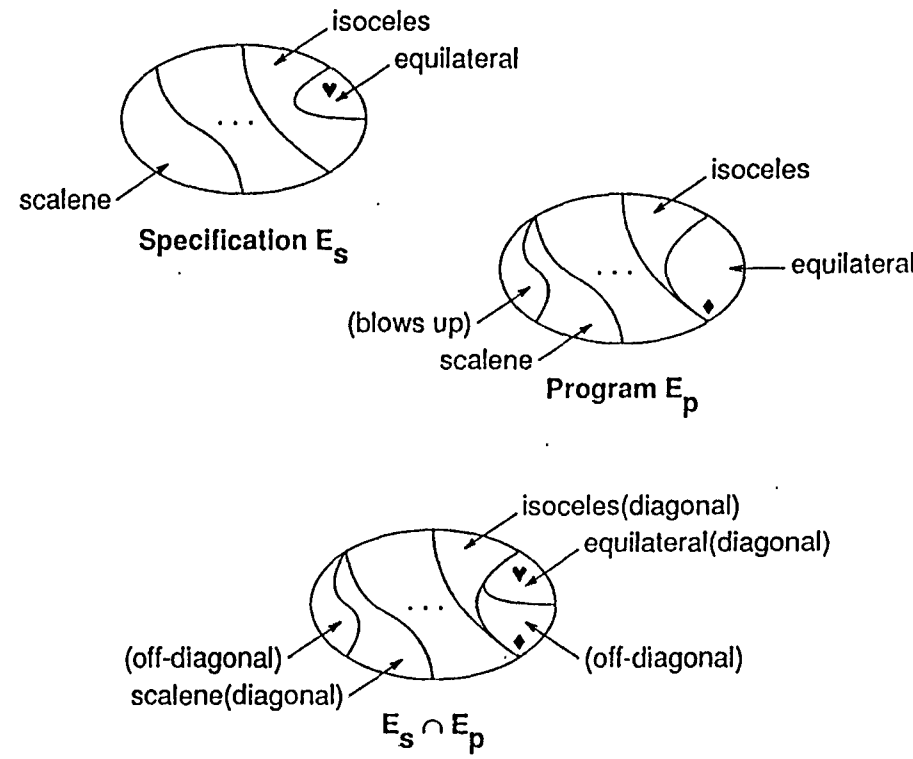
OUTPUT: classify as equilateral, isoceles, scalene, or impossible.

Example: (1,1,5) — impossible

Same-output classes:

- All inputs such that output ***should be*** "equilateral," etc. (specification partition)
- All inputs such that output ***is*** "equilateral," etc., plus class where program blows up (program partition)
- Intersect specification and program partitions to form "diagonal" and "off-diagonal" classes

Partitions of the Triangle Program



No amount of testing in diagonal classes (♥) can determine if an off-diagonal class (♦) is empty.

Outline of Part II

Critique of Method Comparisons

- ☐ Empirical studies
- ☐ Subsumes (inclusion) order
- ☐ Cross-method comparisons

Empirical Studies

Lauterbach/Randall study for RADC:

The only significant factor in testing ***effectiveness*** is the human subject doing the testing.

The testing method is a significant factor in testing ***cost.***

Q: How to control for the "nose-rubbing effect" of people evaluating test methods?

A: Random generation of test data.

Subsumes (Inclusion) Ordering

Definition:

Method *S* ***subsumes*** method *W* iff any test satisfying *S* necessarily satisfies *W*.

But a subsumed method may be intuitively better:

```
function Q(X: real): real;
  {square root of absolute value}
  begin
    if X < 0 then X := -X;
    Q := sqr(X) {oops! sqrt}
  end
```

Haphazard testing with the test set {3} beats statement/branch/dataflow/mutation testing with {-1,0,1}.

Methods may encourage "trivial" satisfaction.

Subsumes is Irrelevant

Definitions:

A test is ***misleading*** iff it succeeds, but some other test would fail.

A test is ***exposing*** iff it fails.

Consider any effective testing method *S* and any monotone testing method *W*. There always exists a program with bugs such that:

(1) *S* has a misleading test.

(2) *W* has an exposing test.

- In a meaningful comparison between two methods, either may appear better, depending on the choice of tests.
- Same for comparing a method with itself!

Cross-method Madness

Use method *T* as an adequacy criterion to judge particular test data obtained from method *M*.
(*T* is the "touchstone".)

(= empirical study of *subsumes*!)

Special cases:

i) The degree of satisfaction of *T* is used to compare *X* and *Y*. Analogy: using an erratic electronic instrument to judge which of two other instruments is better.
(One of *X* or *Y* may excite a deficiency in *T*.)

ii) Heuristic generation of data for a method *M* is compared to *M* itself as touchstone.
(The heuristic may satisfy *M* in a trivial way.)

Outline of Part III

Comparison Using a Valid Standard

- ☐ Random testing
- ☐ Failure-rate standard (reliability)
- ☐ Defect-rate standard (trustworthiness)

Conventional Reliability

A program's behavior is characterized by a ***failure rate*** θ, the long-term ratio of failed to total executions, when independent inputs are drawn from the operational distribution.

The chance that one execution fails is θ, so the chance that none of n executions fails is $(1-\theta)^n$.

Confidence $1-e$ in $\theta \leq p$, requires N tests, where:

$$(1-p)^N = e.$$

For example, 99% confidence in $\theta < 10^{-6}$ (that is, that less than 1 in a million executions will fail) requires about 5 million tests.

Random Testing

The range of each input variable is established, and test data are independently drawn from those ranges. In the absence of information, a uniform distribution may be used, but an "operational distribution" is better.

Operational distribution. If information is available on program use (past or predicted), the input space can be divided into intervals, and a probability of use assigned to each. These probabilities are the chance that any actual execution of the program will have inputs in the given interval. By selecting test data according to the same probability, tests are a valid sample of operation.

Failure-rate Standard

- Partiton testing vs. random testing, compared for ability to expose faults.

  (Full path testing is a partition method, and so are variants of statement testing, etc., that do not define true partitions.)

- Results (Duran & Ntafos, Hamlet & Taylor):

1) ***Partition homogeneity isn't important.*** The intuition that grouped inputs must be "treated the same" is apparently wrong.

2) ***Good classes must have high failure rates.*** Unless you know where the program is likely to fail, random test selection is better than clever schemes for dividing the input space.

3) ***Refining a partition usually isn't useful.*** Unless splitting a partition concentrates failures, it is no more useful than more tests taken at random.

The results suggest better fault-finding methods than those based on flawed intuition.

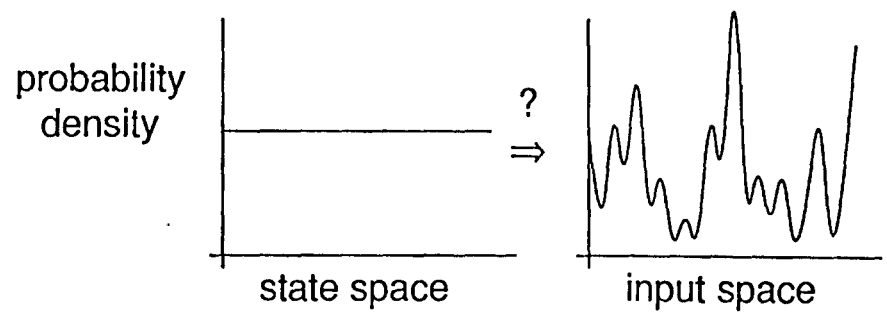
Defect-rate Standard

The failure-rate standard is itself flawed — there may be no sensible operational distribution, and no meaningful failure rate.

It has been suggested that a more plausible theory could be based on *defect rate*, the failures per line of code. Any failure can be traced to a program state-space point:

> Reached *this* statement with *those* variable values...oops!

Tests should be uniformly distributed over the state space (for why should a mistake be more likely at one state than another?). The trouble is that the reflection into the input space is nasty:



Trustworthiness

A defect-rate theory would address what David Parnas calls "trustworthiness" — the property that software can be relied on not to show serious flaws ***under any circumstances.*** This is quite different from the failure-rate theory which deals only with software ***in normal use.***

Unfortunately, rough estimates indicate that it is several orders of magnitude more difficult to test a medium-sized program for trustworthiness than for high reliability; and, the disparity ***increases*** with the size of the program.

Nevertheless, the defect rate theory has some tentative implications for testing practice.

Outline of Part IV

What can we do now?

- ☐ Find more faults (design-based testing)
- ☐ Stop kidding ourselves
- ☐ Work out the theory

"Design-based" Testing

The partitions of specification-based testing are likely to have uniform failure rates.

Here is an alternative:

As design and implementation progresses, use process information to define test classes.

For example, some high-failure-rate classes might be:

- "uses code from novice programmer X"
- "forces collision in hash table"
- "invokes most deeply nested branch of routine with McCabe 25"
- "varies parameter added in design review"
- "executes statements changed after code review"
- "uses failure data reported from beta test"

The failure-rate theory predicts that design-based testing should be superior to random testing, and to other partition methods, in finding faults.

Applications of the Failure-rate Theory

⇒ dataflow ≥ full path

(*Subsumes* goes the other way!)

There are less classes, but failure rates should be higher. Trivial data is less likely.

⇒ Cleanroom development should use design-based testing.

(Of course, it looks good to find no bugs!)

Design-based testing should be an improvement over both random testing and specification-based testing in finding faults, and the cleanroom methodology is ideal for gathering the process information needed to use it.

Applications of Defect-rate Theory

⇒ Unit testing requires units with simple control flow.

For straight-line code uniform state-space sampling is obtained from uniform input-space sampling.

(But effective, "design-level" specifications are lacking for such simple units.)

⇒ Data-coverage testing (mutation) should be superior to path-coverage testing.

It is not easy to find trivial mutation covers. Killing mutants intuitively forces more state coverage than following paths does.

Stop Kidding Ourselves

- Programs are severely undertested. Many orders of magnitude more test points are needed for trustworthiness. ☹
- Testing is a process that parallelizes perfectly. ☺
- Testing "random" large programs isn't practical, and never will be. ☹
- Clever people can take care when they design and implement. ☺
- Tests must monitor development and detect faults that slip in—research is needed. ☹
- Research is needed on fundamental testing theory. ☺

Paper 1-4

VALIDATING PROGRAMS WITHOUT SPECIFICATIONS

Prof. William E. Howden
University of California, Irvine

Prof. Bill Howden received the Bachelor's degree in mathematics and the Master's degrees in both mathematics and computer science. He received the Ph.D. degree in Computer Science from the UC/ Irvine. He is a Professor of Computer Science at UC/ San Diego. His principal area of research is program testing, on which he has written one book and co-edited two others. His current work aims at defect analysis for detecting the occurrence of errors in large, complex systems only informally and incompletely specified. He has written numerous papers in many areas of program testing and analysis, as well as papers on program design, programming environments, and man-machine interaction.

VALIDATING PROGRAMS WITHOUT SPECIFICATIONS

William E. Howden

Department of Computer Science and Engineering
University of California, San Diego
Mail Code C-014
La Jolla, California 92093-0114

This work was supported by the Office of Naval Research and the Naval Weapons Center.

Error-based testing and analysis

In the error-based approach to program testing and analysis, the focus is on errors that a programmer or designer may make during the software development process, and on techniques that can be used to detect their occurrence. This can be seen to differ from other approaches in a variety of ways.

In *fault-based* testing the focus is on flaws in the code or design, or on faults as opposed to errors. Errors by programmers can lead to faults in code, but it is only for certain classes of errors that there are conveniently recognizable fault classes. Typical methods which concentrate on faults in code are algebraic testing, mutation testing, weak mutation testing, perturbation methods and relay testing [1-10]. Error-based testing has a broader focus than fault-based testing and can be thought of as having fault-based testing as a subdiscipline.

In classical *functional testing*, the observable functions implemented by a system or program are tested over typical input-output scenarios. This can be interpreted as an informal kind of statistical testing in which informal measures of reliability, or "levels of confidence", are built up as a system is tested over cases selected according its expected usage profile. Functional testing is often augmented with special case and boundary tests. In this case it becomes an informal, indirect kind of error-based testing since the focus is on the ways that programmers make errors, rather than on the expected functional use of the system.

Coverage measure testing can be interpreted to be an indirect method of fault-based testing. If a program has a fault in it, then we cannot find it unless the code containing it is tested or analyzed. Newer coverage measures contain strategies for requiring the coverage of functionally related pairs or sets of program statements [eg. 11-14]. They can be interpreted in several ways. One is as methods for detecting faults that involve the combined effects of statements. They can also be viewed as techniques for testing embedded subfunctions in programs, functions which are not externally observable but which are invented as part of the program construction process. Statements which have a data flow relationship are probably part of the same function, so that requiring that they be tested together on some test encourages the testing of that function.

Error-based testing methods have been constructed in the past but the error-based focus does not appear to have been as systematically examined as other approaches. As mentioned above, boundary value and special case testing can be thought of as error-based methods. These informal error-based methods were made more systematic in the testing guidelines proposed by Goodenough and Gerhart in [15]. Domain testing [eg. 16,17] may be viewed in a variety of ways. By concentrating on domain shifts, and associated boundary tests, it explores areas where programmers often make errors. Alternatively, domain shifts can be viewed as the effect of faults in conditional branching statements, and domain testing viewed as an indirect fault-based approach in which the faults are indirectly described in terms of data domains rather than executable code.

Our focus on error-based testing is motivated by the following observation. It is often the case that a program is constructed without any formal, detailed specification. In this case the code itself is the only complete specification. This means that the only way to verify such a program is to ensure that no errors were made by the programmer during programming. The term "errors" here means errors that occur due to human fallibility. This requires that we study the ways in which humans make mistakes in the construction of artifacts, and then build methods to detect when they have occurred.

Missing specifications

The lack of specifications for a system may occur in different ways. For many systems, it is simply not possible to construct a compact, concise specification. This is

found in applications in which there are many different kinds of detailed data and data conditions, and a complete specification would have to detail the actions of the system for all of them. In practice such a specification is just a program. The working out of many of the details for such a system often only occurs as the system solution is being solved by being programmed. This has been observed to be a common feature of real-time and data processing systems.

Even when complete specifications exist, if they are not operational then it will be necessary to construct an initial abstract, informal design which points the way towards a programming solution to the problem. This may take the form of a PDL. Filling in the gaps, and adding details to a design, corresponds to programming with informal, incomplete specifications. In order to detect problems that occur in this process it is necessary is to look to see if errors were made in the reasoning that occurs at this detailed level.

Several possible arguments can be made against the above point of view. One is that we should always have detailed specifications, even when they look like a program written in another language. This is unlikely to happen. It is hard enough to construct one large system, let alone two and then prove them equivalent. Another argument is that we can use proofs to connect the details of a program solution with the properties of a higher level abstract, formal specification. In practice, this is impractical and this kind of activity is not even common in mathematics. Proofs are used to establish the validity of a concept and are seldom performed at the level of formality that would be required by detailed program proofs. The problem is that the proofs become overwhelmingly long and bogged down in detail. The programmer is left with the feeling that it would be more likely that an error would be made in such a proof than in the more natural expression of an operational concept in a suitable programming language. Proofs are good for detecting flaws in logical reasoning about complex concepts, but their use appears to be impractical for detecting errors in the translation of concepts into the details of a programming language representation.

The argument which we have presented for error-based testing is the same as that which is often presented for testing in general. There are some differences that can be pointed out. The first is that although the use of error-based, fault-based and coverage testing are all justified by these same arguments, error-based testing focuses directly on the source of problems: errors that humans make when constructing complex objects. Fault-based testing is viewed as suitable for discovering the effects of certain kinds of errors. Coverage testing is considered to be a simple kind of fault-based testing. The purpose of functional testing is viewed as being different from these methods. In the end, the only thing that can really be known about a system is how it has performed in its operational environment. The better it performs, the longer the runs will be between failures, and the higher the resulting possible reliability statistics. Functional testing is used to establish these statistics or, if used informally, to establish informal levels of confidence. Error-based testing is used to detect and get rid of mistakes as quickly as possible, so that long successful runs will occur during functional testing.

Error models

We have used a simple model in which human errors are classified as being either errors of *decomposition* or errors of *abstraction*. It is observed that in order to deal with complexity, humans decompose objects into components and/or delete details by creating abstract descriptions of objects. Abstraction errors occur when a programmer thinks that a piece of code implements some more abstract concept and this is incorrect. Examples of techniques which might be used to detect abstraction errors include symbolic evaluation, which can be used to construct a high level algebraic description of a sequence of code statements. Other examples include those in which the programmer thinks a piece of code will compute sequences of integer indices, correct

samples of which the programmer can easily produce. Testing can be used to confirm that the code, an abstract representation of these sequences, matches the actual, correct sequences.

Abstraction errors can occur during requirements analysis and design, as well as coding. During requirements, an abstraction may not be elaborated in more detail because it is assumed that the details will have no major effect on the system. Later, during design and programming, knowledge of a detail occurs that has a major effect that requires revision at a high level. Requirements and design analysis, in which parts of a system and its interactions with other parts are independently examined in more detail, can be viewed as a way of trying to avoid the occurrence of such errors. Design philosophies such as information hiding, can be viewed as a way of containing the effects of abstraction errors.

Decomposition errors occur when a programmer or designer, working in one part of a system, makes false assumptions about the properties of some other part of the system. The other part may have already been constructed, or is going to be constructed at some point in the future. The original work on static analysis [18] implemented a simple kind of decomposition error analysis method. This tool checked to see if the assumed previous setting of a variable implied by the occurrence of a variable reference was ever actually programmed. This is checked by looking at all paths leading to the reference and examining them for an assignment to the referenced variable. Interface checking between subroutines, in which actual and formal parameters are checked for consistency of type and number, is also a kind of decomposition error analysis.

Type checking can be interpreted as a mechanism for detecting decomposition errors in the following way. When a programmer references a variable in a statement, it is being assumed that that variable contains some kind of data. Errors occur if the wrong variable is being used, or in fact this variable is being used to store some other kind of data. If the property of the variable in question has a lifetime that corresponds to a programming construct that allows type declarations (eg. procedures, compound statements, etc.) then the property can be declared in a type and mis-uses with respect to this property detected automatically.

It has been our observation that there are important classes of decomposition errors that cannot be conveniently checked for using interface checking, type checking and the other conventional mechanisms mentioned above. Decomposition errors often involve object properties whose lifetime does not correspond to a programming language construct that makes type checking an appropriate error checking device. In addition, decomposition errors may involve information whose importance is discovered during programming, and which needs to be documented at that time, and at the point in the program where it occurs and with which it is associated rather than later in a specification or type declaration.

One of the most important properties of decomposition errors in programs is that they are mismatches between abstract properties of different parts of a program or system, and do not involve the details of the code implementing the abstractions. It is often enough to check the consistency of related abstractions, such as those that are documented in comments, without actually examining individual program statements.

The recognition that errors can be separated into abstraction and decomposition has two beneficial effects. The first is that errors involving misunderstandings between different parts of a program can be determined without having to execute or simulate the execution of code, and hence without getting bogged down in irrelevant detail. The second is that abstraction errors, the other half of the picture, are often confined to local areas of the program, and are amenable to techniques like fault-based testing which are guaranteed to be effective on computationally simple structures.

All of our current work has been on techniques for detecting errors which result in inconsistencies between program properties which are established in one part of a program and are assumed to be true in another. In particular we have been concerned

with properties of objects which, although similar to types, differ in that they change dynamically. In addition, the properties are invariably related to the meaning of the object and its stored data as opposed to the kinds of operations that can be carried out on it. The following section describes a technique called *flavor analysis* for detecting decomposition errors of this kind.

Flavor analysis

The basic idea in flavor analysis is that during design and programming, objects are created and manipulated which are expected to have different properties or *flavors* during an execution of the program. When the programmer is working with one or more of these objects in one locale of the program, assumptions will be made about object flavors which are expected to be established at an earlier time during program execution, or assumptions about expected uses of an object at parts of the program that will be executed later in time. In concurrent systems the situation may be more complicated and the assumptions may concern expected orderings of object flavor occurrences in tasks other than the task that is the current focus of attention.

In flavor analysis, objects are abstractions and do not necessarily correspond to program components such as variables or data structures, although this is the usual case. For example, suppose it is necessary to reason about the current and previous value of a variable X. Then there will be two abstract objects involved. One of them will be "previous_value_of_X" and the other "current_value_of_X". Reasoning involving abstract objects and their flavors is done in an object flavor domain of abstraction that is above that of the program. If there is a mismatch between this domain and the program that mismatch is considered to be an abstraction error. Flavor analysis is only used to detect decomposition errors in the abstract domain.

Flavor analysis involves the analysis of a program's *flavor states*. Flavor states form an abstract model of a program at the level of the programmers reasoning about the program in terms of abstract objects which have properties of interest at different times during the execution of the program. The programmer is expected to document a program's flavor characteristics with flavor *assumptions* of the form ?X is alpha? and flavor *assertions* of the form !X is alpha! The flavor analyzer then analyzes the possible flows of control through a program to see if assumptions are justified. An assumption ?X is alpha? is justified by the occurrence of flavor assertions of the form !X is alpha! which must occur along every path of control which arrives at the assumption. Assertions can be thought of as actions which change the current flavor state of a system when control passes along a path through such an assertion. Assumptions can be thought of as statements about the expected flavor state of the program at that point. Even this very simple form of flavor analysis has been found to be extremely useful. The requirement that the programmer construct the appropriate comments has not been found to be an extra burden, but more a set of guidelines for comments that should be included in any case.

Typical examples of the value of flavor analysis include its ability to reveal the occurrence of errors like the following. In a large general ledger system that was analyzed, there is a module that reads in transactions from an accounts file and uses them to produce two reports. There are two kinds of transactions: financial and non-financial. Both kinds contain account numbers and the first kind contains accounting information. The file is sorted on account numbers. The program is supposed to print out one report with account totals, and another with the non-financial records. The program also does many other things, and as a result of this, account breaks (changes from one account to the next) are determined in a part of the program other than that in which subtotals are maintained. The programmer included code to determine if an account break had occurred, and which set a flag to record this, in a section of code that does some preliminary processing of financial records. The programmer apparently did

not remember to also see if an account break had occurred when working on the non-financial record processing code. This resulted in a false assumption in the sub-totals code that the transaction currently under consideration at that point had the flavor "accounts break checked for". The result was that when an account break occurred with a non-financial transaction, and that account also had financial transactions, the account totals for the previous account would not be reported and would be added on to the total for the following account. The use of simple flavor comments to document assumptions that are known during programming would prevent errors like this from going undetected.

Flavor analysis is a kind of dynamic type checking. It allows the programmer to document properties of objects which change during the operation of a program, and to check if assumptions about an object's current set of properties are correct. For example, a simple numeric variable may at different times contain a partial sum, a total sum, or a mean. None of these properties are types in the usual sense of the word, even when named types are available, since they change dynamically. Flavor comments allow checking of these dynamic properties. By including the comments in the code at the points where the flavor assumptions occur, and the points where the flavors are established, this information is captured at the point where it becomes known and is not lost during the programming process. It is argued that flavor information is an important part of the thought processes that occur during programming, and if it is not recorded and made part of the verification process, then those errors which involve it cannot be directly checked.

Two flavor analyzers have been built. The second is the initial analyzer in a planned sequence of increasingly more powerful analyzers. The first was an analyzer for COBOL business data processing programs. This application was chosen because this technology is intended to be a practical solution to real world problems, and COBOL data processing is a rich source of data about the problems that occur in the commercial programming world. This analyzer had very simple flavor comments, and was very effective in detecting errors in a set of production COBOL programs [19]. The construction and use of flavor comments was not found to be a problem in the use of the analyzer. Problems that were observed included that of infeasible paths. Many of the error messages that were produced by initial versions of the tool reported flavor errors that occurred along unexecutable paths and were really not errors. To get rid of these required some form of infeasible path analysis. The approach that was used was to include a facility in the analyzer for tracking the values of selected "control variables" and for determining the values of simple booleans in conditionals occurring along program paths. This worked, but the drawback was the requirement that the programmer supply a list of control variables whose values had to be tracked.

The second analyzer was built for a real-time avionics system written in assembler. This analyzer, called QDA1, used comments of the above type, and avoided the infeasible path problem in the following way. It was observed that in addition to the other abstract information a programmer has in mind during programming, are expectations about possible flows of control through a piece of code. These can be documented using an abstract object called PC (Path Control). At selected places in the program the programmer expects PC to have flavors related to control flow. These are documented in the form of flavor assumptions about PC. If these are not satisfied, analysis does not continue down a program path. This proved to be an effective way of allowing the user to suppress the generation of spurious flavor error messages. It was also natural and easy to use.

Both QDA1 and the earlier COBOL analyzer contain facilities which allow the user to carry out flavor analysis of module boundaries as well as analysis inside modules. This is done using input and output comments. An input comment documents flavor properties that are assumed to hold whenever a module is "called". At each point in a calling module where a module is called the called modules input flavor comments are

treated like assumptions and checked against the current possible flavor states of the calling module. When analyzing a module which has input comments, the comments are treated as assertions, establishing an initial set of flavors for the module. Output comments in a called module are treated as assertions which summarize the effects of the called module. Output comments which are in a module under analysis are treated as assumptions about the final flavor state of the module.

The avionics program for which QDA1 was written is currently being completely re-commented to allow a complete flavor analysis of the code. Our experience so far is that the use of QDA1 and the associated flavor comments is of enormous benefit in reassuring the programmer that assumed flavors of temporary variables and registers are consistent with their past use, as documented by assertion comments at the points of past usage. The technique is particularly useful for maintenance, since when a piece of code is changed, or a temporary used for an alternative purpose, the changes are documented with flavor comments and the code run through the analyzer to ensure that there are no unexpected effects on other parts of the system.

In addition to the two examples of flavor analysis use that involved the construction of tools, flavor analysis was carried out manually on the original design of the QDA1 flavor analyzer. The design was documented in an experimental object oriented language with strong typing facilities. Several important errors were found in this way during design. The comments were also invaluable for documentation purposes. Flavor analysis appeared to be particularly useful for documenting the flavor effects of actions which could be arbitrarily allocated to either one of two or more operators in an object class. Assumptions were used in operators to document the allocation so that when it was assumed in one operator that a data property was expected to be established in another operator, this could be automatically checked for correctness.

Flavor analysis is similar to event sequence analysis [eg. 20-23], and our earliest planned versions of the method were derived from these ideas. In event sequence analysis, the programmer specifies a set of "allowable" or "legal" sequences in which certain events can occur and then the system under analysis is examined to ensure that the specification is not violated. A common form of event sequence specification is some form of finite state diagram or regular expression. In order to analyze a program and compare it with an event specification it is necessary to be able to recognize the occurrence of the events in the program. For this reason the technique appears to be most commonly associated with the analysis of events that correspond to the use of data structure operators or scheduling operations. When it is applied to implicit rather than explicit events, then automated analysis requires the use of comments, like flavor comments, in order to allow the detection of event occurrences. Legal event specifications can arise in different ways. One possibility is as a set of constraints that is created as part of a design. Another possibility is that they document a decomposition of a problem, and record information about the order in which the states associated with different parts of the decomposed system will occur.

The major differences between flavor analysis and event sequence analysis are as follows. One is the emphasis on data flavors and states as opposed to the occurrence of operators. This appears to make flavor analysis more amenable to the analysis of problem dependent properties, as opposed to problem independent events such as the occurrence of data structure operators. Another difference is that flavor analysis uses two kinds of comments, assumptions and assertions, and is concerned with consistency between them rather than between the comments and code. Flavor assertions are not meant to be specifications, and neither are the output flavor statements for a module. If they are being compared to the code with which they are associated, it is to determine if they match the code so that decomposition analysis is not being done with faulty comments. This is the opposite of comparing the code against specifications to see if the code is correct, as is the case for event sequence analysis in which legal event descriptions are considered to be partial program specifications. There is also the

important difference of when flavor assumptions and assertions are constructed. A legal sequences diagram can be thought of as a compound assumption that describes the assumed sequences in which operations (events) will occur in a program. Since it is a compound, complete description, it is likely that it will be constructed either before program construction if it is thought of as part of a design specification, or after program construction if it involves details that become known during programming or design. On the other hand, assumptions and assertions can be created during this process, to capture information as it occurs. It is less likely that information will be lost, and more detailed information is likely to be recorded.

Temporal logic and flavor analysis

A number of desirable features were added in the design of QDA2 which were found to be missing during the use of QDA1. Examples include the use of second order comments. These are comments about other comments and allow the user for example, to describe in one procedure assumptions about the asserted output flavors of another procedure. QDA2 also has facilities for dealing with object flavors that are pointers to other objects, a feature particularly important for documenting and analyzing assembly language programs. In addition, we began to consider other, more advanced features that are planned for QDA3, and which will be designed to allow the application of flavor analysis to concurrent systems and to the analysis of real-time properties.

When programmers reason about a system during its construction, their attention is focussed on some locale in the program. From this locale they make assumptions about what to expect of the data which arrives along paths reaching that locale and make predictions about, or plans for, actions and data properties to be carried out on paths leading out of it. This way of looking at the programming and design process leads naturally to the use of temporal logic. An assumption may always or sometimes true. It is sometimes true if it is only true along some of the paths leading to its location and it is always true if it is true along all such paths. In addition, we may want to make assumptions about the orders in which things became true, or will become true in the future. These, and other considerations are similar to the style of reasoning available in branching temporal logic, such as CTL [24].

Branching temporal logic can be described as starting with a base logic of propositional calculus together with several temporal operators. Two of these concern alternative time paths, and can be denoted as AP() and SP(). The first indicates that the expression in parentheses is true along all time paths originating at the point where an assertion containing such an operator occurs. The second indicates it is true down some path. The remaining operator is the until operator =>. The expression A=>B means that at some point in the future, B is true. Starting now and up to but not necessarily including that point, A is true.

In the application of temporal logic to program analysis, the recommended approach is to try to prove that a program is a model of a temporal logic formula. In the case where the program is a finite state system, model checking has been found to be computationally tractable [24]. The finite state restriction limits the applicability of this result to certain aspects of a program, or aspects of sets of concurrent programs, such as the sequencing of communication states. For example, two concurrent tasks may be in the states "trying to get into a critical region", "in a critical region", or "in a non-critical region". A finite state graph can be used to model the effects of a task scheduling strategy on the combinations of states that a system composed of these tasks can be in, and on the transitions between the states. Model checking can then be used to determine if such a finite state system satisfies a temporal logic formula.

A program together with its flavor comments forms a finite state machine. The flavor assertion comments, inserted at intermediate places in the code, describe changes to the program flavor state as control passes through that part of the program.

Assumptions can be thought of as temporal logic formula which are expected to hold at particular places in a program. This property of flavor commented programs indicates that flavor analysis can be interpreted as model checking against formulae in some kind of temporal logic having properties like branching temporal logic. It is possible to include other aspects of flavor analysis, like input and output flavor statements, within the same approach. Output statements, for example, are either temporal logic formula whose satisfiability is to be proved, or operators which change the state in a finite state machine which is supposed to model some formula. They are theorems about the procedure in which they appear, and state changing operators in the code which calls a procedure containing them.

For the purposes of modelling the reasoning processes that occur during program development, and for providing a formal basis for flavor analysis, we have adopted the use of a logic we have called "network temporal logic" (NTL). The primitives are similar to those in logics such as CTL, but the main difference is that it is possible to focus on things that have occurred along paths in the past leading up to the present, as well as on things that will occur in the future.

As in QDA1 and QDA2, the atomic propositions in NTL are flavor comments. In addition to the usual propositional logic operators, NTL includes the following: AP[f], EP[f], AF[f], EF[f]. Intuitively, the first formula means that along all time paths coming from the past, f is true when they get to this point. EP[f] means f is true at this point along some path from the past. AF[f] means that along all time paths into the future, at some point f is true. EF[f] means that along some path into the future f is true. Also included is the notation f1=>f2 from CTL which means that f2 is true at some point in the future and that f1 is true from now until then. A similar formula fi<=f2 is included which means that f2 was true at some point in the past and that immediately after that point and up till now f1 was true.

We have also included some notational conveniences such as CP[f1,f2,...,fn] which means that for each i, $1 \le i \le n$, there is some path from the past which leads to fi being true. In addition, for each path some fi, $1 \le i \le n$, is true, and for no paths are two fi true. CP stands for "Complete Path" information. Other notational conveniences include f1<=!f2 which means (f2 & f1<=(f2 & not f1)). For example, the formula

CP[(ground-radar is selected)<=!(bit7 is valid),
(forward-radar is selected)<=!bit7 is invalid)]

could be used to characterize the following assumed output flavors for a procedure. At the end of the procedure there are two possible outcomes, depending on the path followed through the procedure. In the first, both "forward-radar is selected" and "bit7 is valid" are true. In addition, "bit 7 is valid" was known to be true before "forward radar is selected", and hence in some informal sense led to this. Similar remarks can be made about the other possible alternative.

The formula AF[f] corresponds to a simple flavor assumption of the kind found in QDA1. The input and output flavor comments from QDA1 are interpreted as place designators in NTL. "output", for example, is used to denote that a formula is associated with a time point corresponding to the place where a program path terminates. NTL is also designed to allow a programmer to make comments about flavors and their occurrence in more than one time domain. A programmer reasoning in the time domain of one task can make comments about object flavors and their occurrences in the time domains of other tasks.

Summary

In the error analysis approach, the verification problem is viewed as the problem of detecting when programmers have made errors. These are errors which are caused by the limitations of human reasoning and the complexity of systems. This is contrasted with approaches in which the goal is to confirm that a program matches its

specifications. The error-based approach allows the possibility of situations in which there is no detailed set of specifications, in which it is determined during code construction exactly what will be done in each of a wide variety of circumstances. In this latter situation the code is the only formal specification since such systems often do not admit the possibility of compact, concise, complete, specifications of the type that are usually present in proof of correctness examples.

Two broad classes of errors: abstraction and decomposition were identified. A technique called *flavor analysis* has been developed and used for preventing and detecting errors of the second kind. In flavor analysis programmers document their assumptions about the current states, or flavors, of objects with assumption comments. They also document with assertion comments the points in code at which flavors or states are established. Objects are abstractions which may or may not correspond to variables and data structures. Analysis of object properties is carried out completely in the domain of the flavor assumptions and assertions. Flavor assumptions and assertions describe the abstractions used by the programmer during reasoning about the program and analysis of them allows the detection of decomposition errors in this domain. Provisions are made for the use of modularized code in the form of input and output comments. Both can be interpreted as either assumptions or assertions. Input comments state assumed flavor states of objects that are expected to exist whenever the procedure containing the input comment is called. Alternatively they state assertions about the initial states of object flavors upon commencement of the code in the procedure body. An output comment is an assumption about the final flavors of data objects upon procedure termination. It can also be viewed as an assertion about its effects on the flavor state of a calling procedure.

The "spatial" aspects of program reasoning which occur during programming can be modelled using temporal logic. "Spatial" refers to reasoning which includes the consideration of properties expected from different parts of a program other than that which is currently being developed. Decomposition errors are thought of as occurring in this domain, and temporal logic is used as the formal basis for the development of techniques for decomposition error detection. The basic idea is that during programming, programmers make assumptions about object properties, or flavors, that will occur earlier in time during the execution of the program, or properties that will occur in the future, later in time. In addition, a programmer may make assumptions about the order in which properties are established in the time domains of other programs. These assumptions correspond to formulae in a suitable temporal logic. The program under construction is documented with both assumption and assertion comments. Assertion comments describe the changes in object flavors which occur in a program as it is executed. This results in a finite state description of how the program operates, in terms of its flavor states. Determining he correctness of a flavor assumption corresponds to determining if the finite state description is a model of the temporal formula corresponding to that assumption.

Current work includes the design of a more advanced non-temporal version of QDA1, a network temporal logic version, and theoretical work on a real-time version of network logic, RTL. The basic idea in RTL is to associate a time range with the NTL operators => and <=.

References

{1} K.A. Foster, Error sensitive test data for logic expressions, ACM Software Engineering Notes, 9,2, 1984.

[2] R.A. DeMillo, D.S. Guindi, K.N. King, W.M. McCracken and A.J. Offutt, An extended overview of the Mothra software testing environment, *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, 1988, IEEE Computer Society.

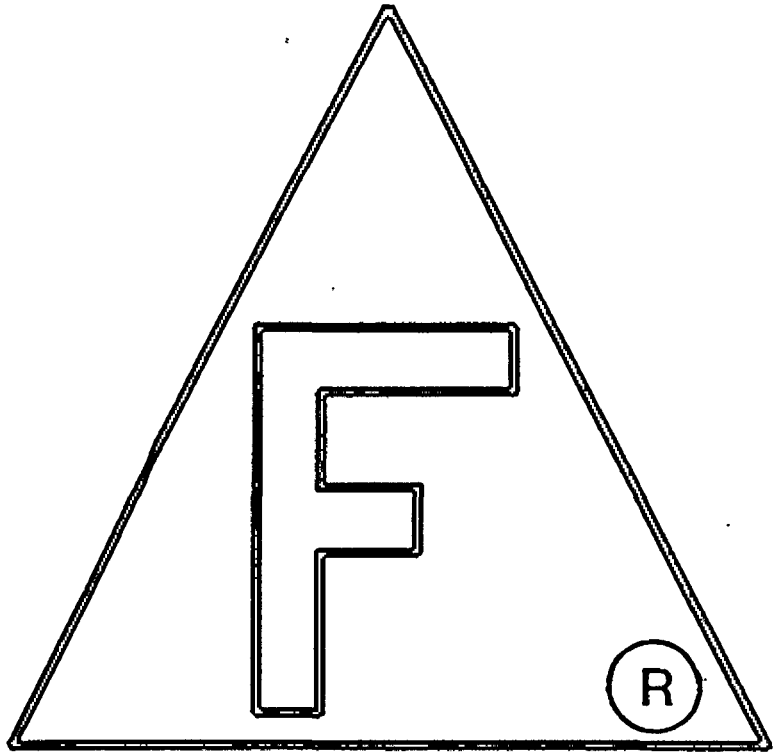
Paper 1-5

INTRODUCING INSPECTIONS IN YOUR ORGANIZATION THE "DEFECT-FREE PROCESS"

Mr. Michael Fagan
Michael Fagan & Associates

Mr. Michael Fagan has had 20 years management experience with IBM. In 1979, IBM awarded him the largest individual Corporate Award for creating the software inspection process. Since leaving IBM, in January, 1989, he has provided education and consulting on improving quality and productivity to several organizations, both large and small. From 1983 to 1985, he was a Visiting Professor in the Department of Computer Science and was a member of the Graduate Council of the University of Maryland.

INTRODUCING INSPECTIONS IN YOUR ORGANIZATION - THE "DEFECT-FREE PROCESS".



Michael Fagan (619) 456 - 9090

AGENDA.

QUALITY INSPECTIONS & THE "DEFECT-FREE PROCESS".

- PROBLEM.
- DEFECT-FREE PROCESS.
- GOAL LEVELS OF QUALITY AND PRODUCTIVITY PERFORMANCE - WITH EXAMPLES OF ATTAINMENT.
- FORMAL PROCESS DEFINITION.
- FAGAN INSPECTION PROCESS.
- WALK-THROUGHS & INSPECTIONS.
- INTRODUCING THE "DEFECT-FREE PROCESS" INTO YOUR ORGANIZATION.

Michael Fagan (619) 456 - 9090

PROBLEM:

- Design/Coding from incomplete Requirements can halve productivity by causing "multiple trip shopping" to get sufficient Requirements - in order to rework the design.

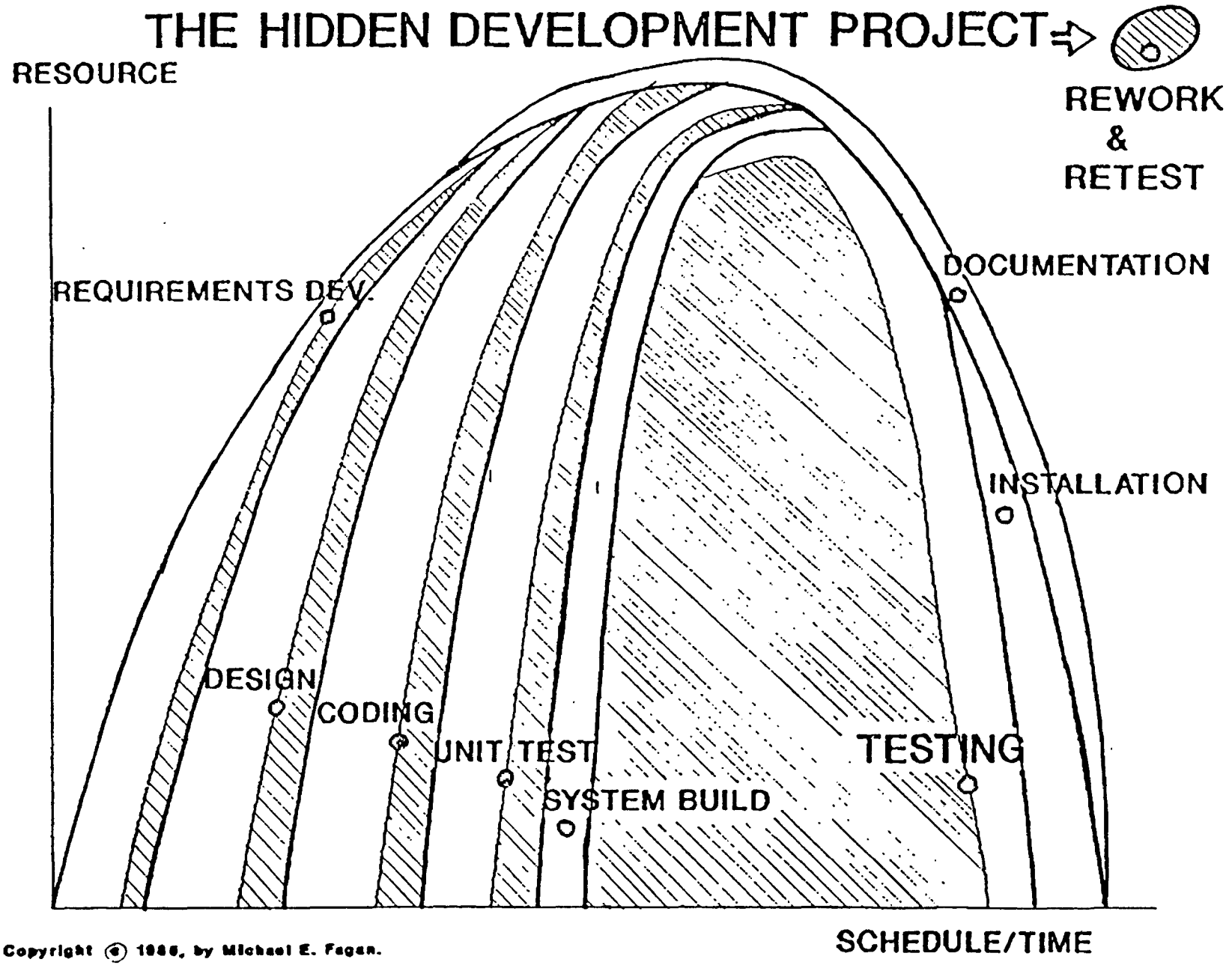
- Changes in design - due to additions/changes in Requirements *and* Defect Rework - contain 3 times the defect rate of the base design. This generates excessive defects to be handled by development and by the users.

- Defect rework reduces development productivity.
  - Some estimates cite 30 - 60% reductions.

LIFE-CYCLE DEFECT REWORK COSTS
- TYPICAL COST PER DEFECT (Person.Hours.)

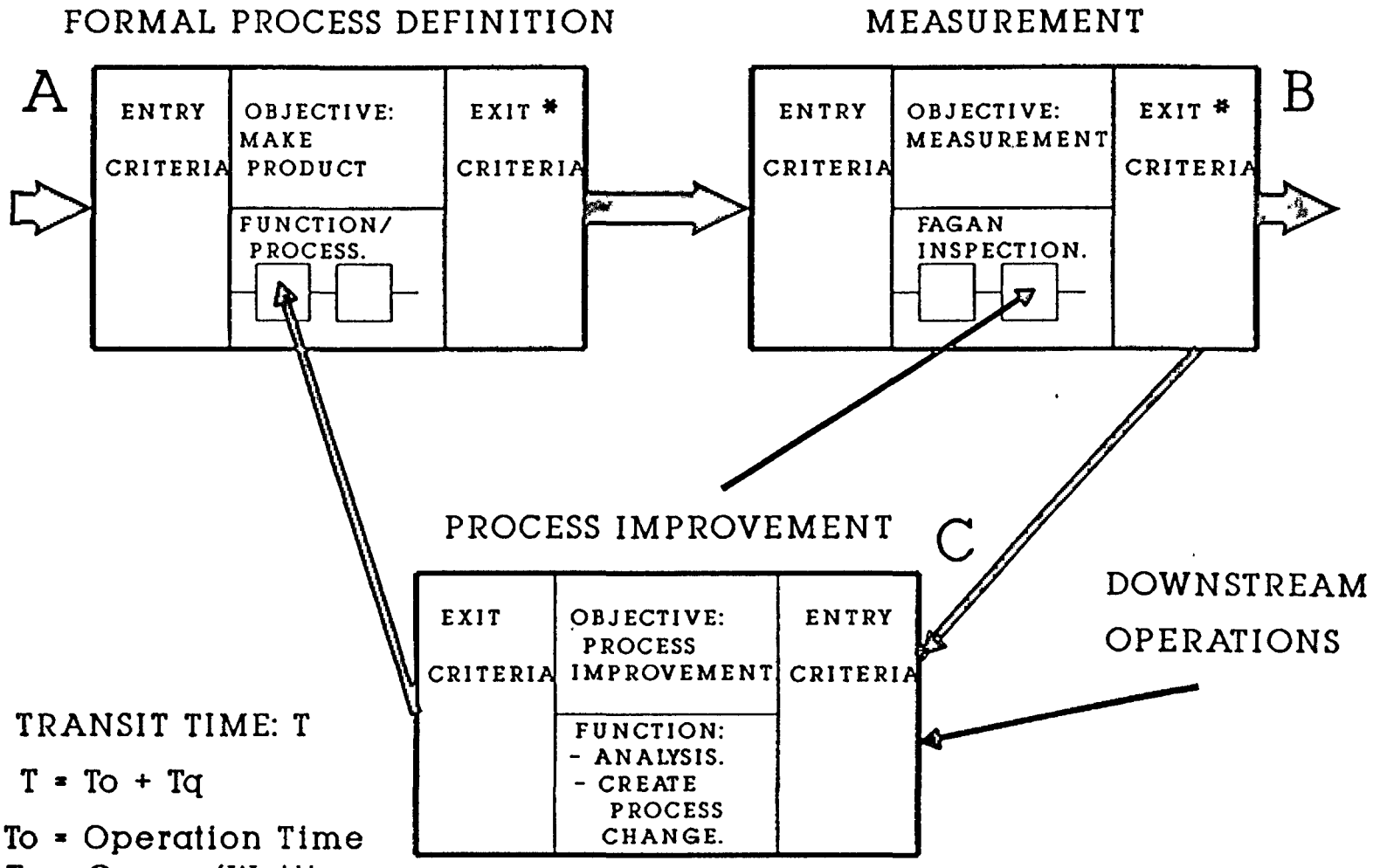
		SMALL PROGRAM.	SYSTEM PROGRAM.
REQUIREMENTS	- IR	1	1 (P.Hr.)
HIGH LEVEL DESIGN	- IO	1	1
LOW LEVEL DESIGN	- I1	1	1
CODE	- I2	1	1
UNIT TEST		3	4
FUNCTION TEST		5	7
SYSTEM/ACCEPTANCE TEST		12	20
FIELD/IN USE		20+	30+

THE HIDDEN DEVELOPMENT PROJECT



DEFECT-FREE PROCESS

- Through continuous incremental process improvement.



TRANSIT TIME: T

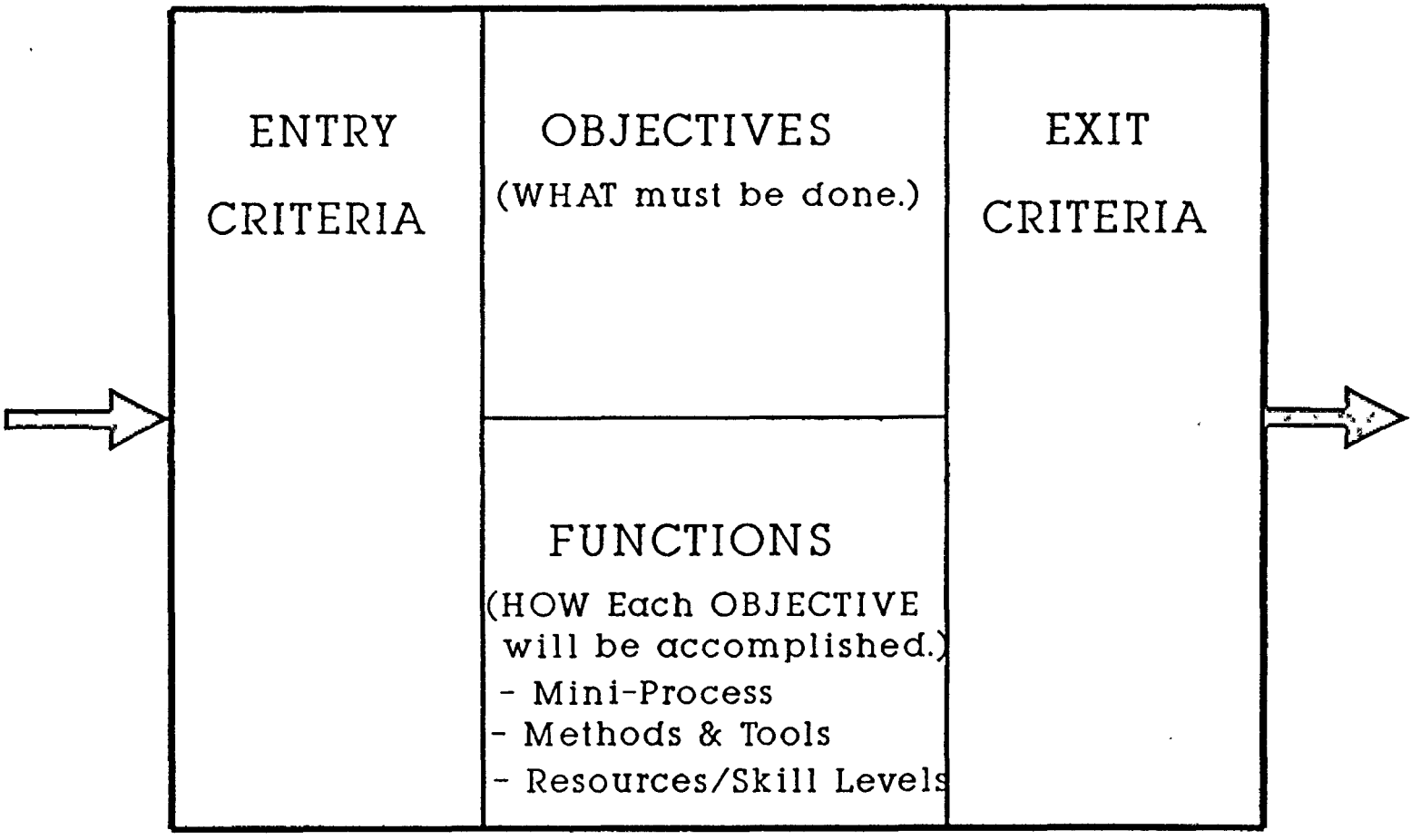
$T = To + Tq$

To = Operation Time

Tq = Queue/Waiting

FORMAL PROCESS DEFINITION.

FORMAL PROCESS DEFINITION.



QUESTION: CONSIDERING YOUR DEPT. AS A PROCESS OPERATION OF WHICH YOU HAVE A CLEAR UNDERSTANDING OF THE OBJECTIVES/FUNCTIONS\ENTRY CRITERIA/EXIT CRITERIA, HOW WOULD YOU CLASSIFY IT IN TERMS OF?:

	CLEAR	FUZZY	UNCLEAR	AGREED WITH SUPPLIERS	AGREED WITH CUSTOMERS	ROUTINELY MEAS	ROUTINELY MET
OBJECTIVES	___	___	___				
ENTRY CRITERIA	___	___	___	___		___	___
EXIT CRITERIA	___	___	___		___	___	___
FUNCTIONS	___	___	___				

GOAL LEVELS
- OF QUALITY AND PRODUCTIVITY PERFORMANCE.

FRAMEWORK FOR COMPARING LEVELS OF:

- PROCESS DESIGN and PROCESS EXECUTION,

  and resulting

- QUALITY and PRODUCTIVITY improvements.

(Goal Level case study data from Formal Process Definition and Fagan Inspections appear in their sections.)

GOAL LEVELS OF PRODUCTIVITY AND QUALITY - due to FORMAL PROCESS DEFINITION.

GOAL LEVEL	DEVELOPMENT PROCESS	IMPROVEMENT RANGE
A	Existing Process.	Baseline.
B	Requirements EXIT CRITERIA (Design ENTRY CRITERIA) defined & satisfied.	50 - 200% of total development.
C	Test ENTRY CRITERIA defined & satisfied.	50 - 300% of Test.

PRODUCTIVITY GAIN due to "ONE-STOP SHOPPING" - WITH ENTRY & EXIT CRITERIA SATISFIED.

DEPARTMENT:	ESTIMATED PRODUCTIVITY GAIN:
Wiring Design Engineering	49%
Avionics Design Engineering	61%
Software Applications Program'g	67% (*)
Software Complex Applications	98% (*)
Software System Product Develop.	400% (*)

* - Improvement in overall development productivity when Requirements EXIT CRITERIA (= Design ENTRY CRITERIA) are satisfied.

INDICATORS OF POSSIBLE PROCESS INADEQUACY:

- No Process Description Or Standards.
- Project Personnel Not Familiar With Process Description Or Standards.
- Project Personnel Not Involved In Developing Process Description And Standards.
- ENTRY CRITERIA Does Not Match EXIT CRITERIA Of Supplier.
  - No Documented Agreement With Supplier.
  - No Measurements.
- EXIT CRITERIA Does Not Match ENTRY CRITERIA Of Customer.
  - No Documented Agreement With Customer.
  - No Measurements.
- No Method Of Process Change Control That Involves The Users Of The Process. - Is There Anyone On The Project Who Does Not Know The Latest Change In The Process?

MEASUREMENT:

- SOFTWARE INSPECTION

(FAGAN INSPECTION)

GOAL LEVELS with FAGAN INSPECTION PROCESS

	GOAL LEVEL	PROCESS	PROCESS EXECUTION
1	BASELINE.	UNDEFINED, INTUITIVE PROCESS WITH REVIEWS/WALKTHROUGHS	DISCIPLINE LEVEL: MANY JUDGEMENT CALLS.
2	60% DEFECTS FOUND BEFORE TESTING. + 10% PRODUCTIVITY.	Io, I1, I2 ... INSPECTIONS.	INFORMAL MODERATOR TRAINING (0 - 1 DAY)
3	90% DEFECTS FOUND BEFORE TESTING. + 25% PRODUCT'Y.	FORMAL EXIT CRITERIA. Io, I1, I2 FAGAN INSPECTIONS. DESIGN CHANGE CONTROL.	FORMAL MODERATOR TRAINING (3 + DAYS) - ALL CHANGES.
4	DEFECT-FREE IN CUSTOMER USE. + 40% PRODUCT'Y.	FORMAL PROCESS DEFINITION. IR,I0,I1,I2,IT1,IT2,PI0,PI1 & PI2 - FAGAN INSPECTIONS. DESIGN CHANGE CONTROL. CODE CONTROL. TRACK ALL TEST DEFECTS.	ALL REQUIREMENTS, DESIGN, CODE, TEST PLANS/CASES, AND USER DOCUMENTATION. - 100% CHANGES. - 100% & ALL CHANGES. - 100%
5	DEFECT-FREE IN CUSTOMER USE. EXIT DEFECT-FREE EVERY IN-PROCESS OPERATION. + 50% PRODUCT'Y.	(SAME AS 4, ABOVE.) IDENTIFY & FIX DEFECT-PRONE CODE PRE-TEST & PRE-SHIP. DEFECT CAUSE REMOVAL IN EVERY OPERATION.	(SAME DISCIPLINE AS 4, ABOVE.) ROUTINE PRACTICE. "TWO FIXES FOR EVERY DEFECT" - A ROUTINE PRACTICE.

IS ERROR-FREE SOFTWARE ACHIEVABLE?

- IBM Houston - Won one of two NASA Quality Awards for its On-Board Space Shuttle Software.
- Over 2 million lines of code developed over several years.

QUALITY LEVEL:

- 0.11 Defects per 1000 lines of code in use over 2 years.
- 6 space missions with zero defects (IEEE report).

PROCESS:

- Formal design and code inspections.
- 85% of life-cycle defects found by formal inspection.
- 8 levels of testing following inspection (adding greatly to cost) found the remaining 15% of life-cycle defects.

(Edward Joyce, Datamation, 15 Feb. 1989.)

INSPECTION BENEFITS - ZERO DEFECTS.
- AETNA LIFE & CASUALTY, HARTFORD, CT.

- 4439 NCSS COBOL, 8 MODULES - 2 PROGRAMMERS.

- PROJECTED: 62 P.DAYS.
  ACTUAL: 46.5 P.DAYS (Incl. 6.8 P.Days for inspection)
  PRODUCTIVITY INCREASE: 25%

- DEFECTS DETECTED: I1 - 10 DEFECTS/KNCSS.
  I2 - 28
  UNIT TEST - 8
  ACCEPTANCE TEST - 0
  AFTER 2 YEARS PRODUCTION: ZERO DEFECTS.

- 82% OF DEFECTS FOUND BY INSPECTION.(Of 46 Def/K total

INSPECTION BENEFITS - ZERO DEFECTS.
- IBM RESPOND, UNITED KINGDOM.

- 6250 NCSS PL/1, CE-PRS - 7 PROGRAMMERS.

- 9% PRODUCTIVITY IMPROVEMENT OVER PREVIOUS BEST. (Included writing the inspection manual.)

- DEFECTS DETECTED:

I0	-	0.8 DEFECTS/KNCSS.
I1	-	7.1
I2	-	16.4
ALL TESTS	-	1.8 (Cf. 8.7 w/W-Thrus.)
ACCEPTANCE TEST	-	ZERO DEFECTS. (Cf. 2.6 w/W-Thrus.)

- 93% OF DEFECTS FOUND BY INSPECTION.(Of 26.1 Def/K tot

INSPECTION BENEFITS - COST SAVINGS.

UNISYS/BURROUGHS INITIAL INSPECTION EXPERIENCE.
(L.TRACEY, 1986)

- CUMBERNAULD, SCOTLAND.
  110 INSPECTIONS - 571 MAJ & 4357 MIN DEFECTS.
  68% OF ALL DEFECTS FOUND BY INSPECTION METHOD
  NET SAVINGS: $483,300

- FELTHAM, UNITED KINGDOM.
  157 INSPECTIONS - 463 MAJ & 4180 MIN DEFECTS.
  70% OF ALL DEFECTS FOUND BY INSPECTION METHOD
  NET SAVINGS: $397,945

- ATLANTA, GA.
  57 INSPECTIONS - 281 MAJ & 562 MIN DEFECTS.
  NET SAVINGS: $77,145.

(Total NET Savings: $958,390)

STERLING SOFTWARE - INSPECTION RESULTS
NASA/AMES STANDARDIZED WIND TUNNEL SYSTEM.

PILOT STUDY - 1979:

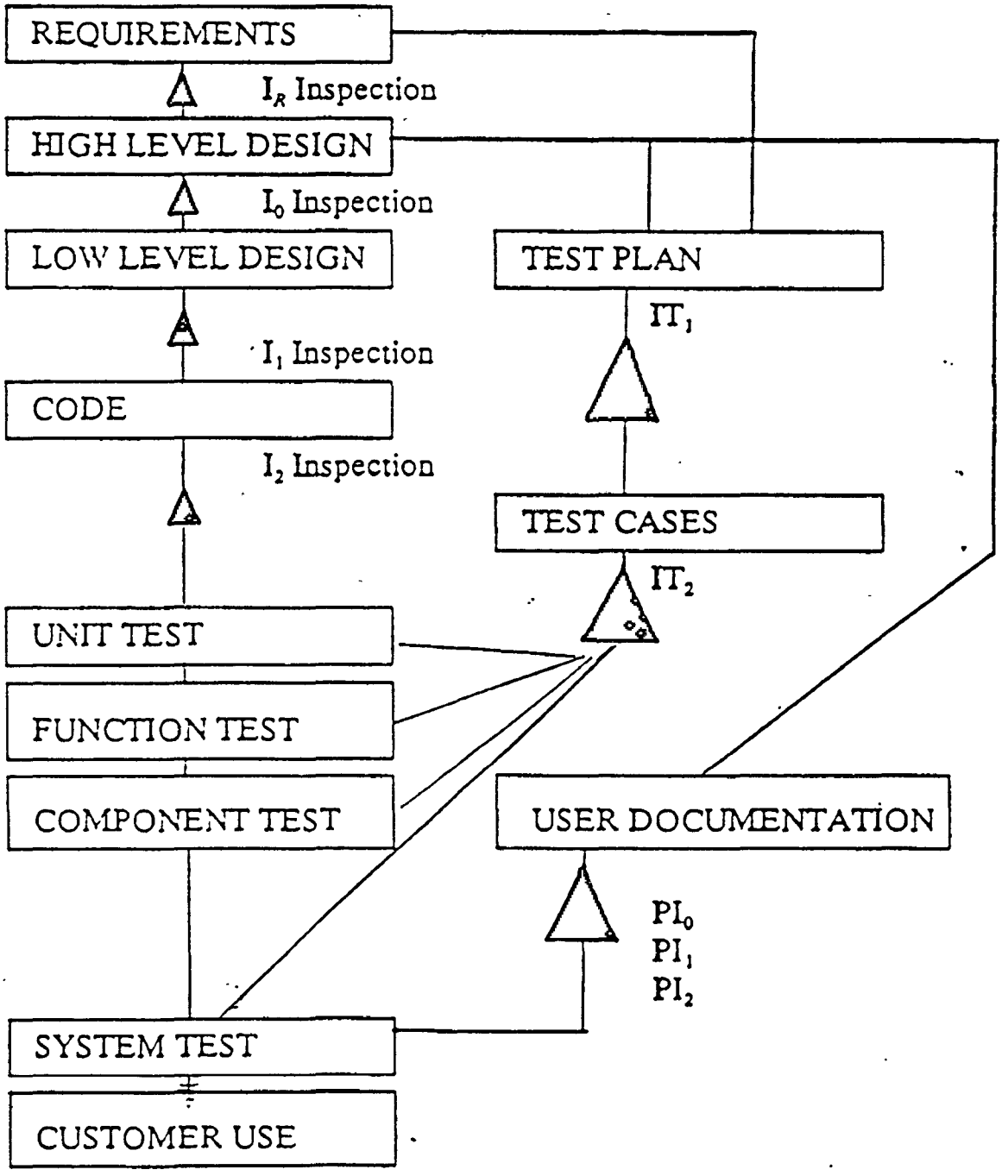
- PRODUCTIVITY GAIN: 40%
- DEFECT DETECTION OF LIFETIME DEFECTS: 90%
- TESTING TIME REDUCTION: 35 - 65%
- POST-RELEASE REDUCTION IN DEFECTS: 40%

FOLLOW-ON INSPECTION ACTIVITY:

- COMMERCIAL PILOT PROJECTS - 1980.
- DoD PILOT PROJECTS - 1985.
- NOW USED ON MOST STERLING PROJECTS.

R.MOLARI, STERLING SOFTWARE, JULY 1988.

DEVELOPMENT PROCESS AND INSPECTIONS



(Michael E. Fagan)

EXIT CRITERIA FOR HIGH LEVEL DESIGN. - Io INSPECTION.

THERE MAY BE 20 ITEMS IN THESE EXIT CRITERIA, INCLUDING (Note qualification/quantification of attributes, wherever possible.):

- DESIGN SATISFIES PRODUCT REQUIREMENTS.
- DESIGN SATISFIES THE OBJECTIVES OF THE HIGH LEVEL DESIGN OPERATION.
- LEVEL OF DETAIL (OR LEVEL OF ABSTRACTION). ONE DESIGN STATEMENT WILL GENERATE 15-25 NCSS.
- CONTROL FLOW DOWN TO THE MODULE NAME LEVEL
- CONTROL BLOCK STRUCTURE DEFINED DOWN TO THE FIELD NAME LEVEL ONLY.
- ALL REWORK FROM Io COMPLETED AND VERIFIED.

EXIT CRITERIA FOR CODING OPERATION. - I2 INSPECTION.

ABOUT 15 EXIT CRITERIA ITEMS MUST BE SATISFIED, INCLUDING:

- CODE ACCURATELY IMPLEMENTS LOW LEVEL DESIGN.
- CODE MUST BE AT "CLEAN COMPILE" LEVEL (NO WARNING MESSAGES).
- ALL PROJECT STANDARDS COMPLIED WITH.
- ALL DESIGN CHANGES TO DATE ARE INCLUDED.
- ALL REWORK FROM I2 COMPLETE AND VERIFIED.

6-STEP INSPECTION PROCESS

OPERATION	OBJECTIVE
1. PLANNING	∘ MATERIALS MEET ENTRY CRITERIA ∘ PARTICIPANTS, ROLES ∘ SCHEDULE ∘ MEETING PLACE
2. OVERVIEW	∘ GROUP EDUCATION
3. PREPARATION	∘ INDIVIDUAL PREPARATION TO TO FULFILL ASSIGNED ROLES
4. INSPECTION	∘ FIND DEFECTS!
5. REWORK	∘ REWORK ALL DEFECTS
6. FOLLOW-UP	∘ VERIFY ALL DEFECTS RESOLVED

INSPECTORS AND THEIR ROLES.

- Each role has a unique function to perform.

- MODERATOR - Manages the team through the 6-Step inspection Process, while playing an active role as an inspector. (Pre-requisite is competence in programming, ability to be tactful, diplomatic, forceful, and work well with others.)

- AUTHOR - Creator of the product or specification that is being inspected. Vested interest is to ensure that the product completely meets exit criteria without ambiguity.

- READER(S) - Usually a developer. Reads the code or spec.- expresses in his/her own words what each statement means - with a level of understanding that they would need to implement it. (This is STEP-4 of the Inspection Process.)

  If the Req. Spec. includes the product externals, an intended end-user should also read through typical work scenarios during Step-4 while utilizing the externals. (Externals and work scenarios must make sense together.)

- TESTER - Will consider testability, interactions with other products/systems, and performance implications.

DESIGN INSPECTION DEFECT FILE

VP	Individual Name	Missing	Wrong	Extra	Errors	Error %
CD	CB Definition	16	2		18	3.5 } 10.4
CU	CB Usage	18	17	1	36	6.9 }
FS	FPFS	1			1	.2
IC	Interconnect Calls	18	9		27	5.2
IR	Interconnect Reqts	4	5	2	11	2.1
LO	Logic	126	57	24	207	39.8 ⇦
L3	Higher Lvl Docu	1		1	2	.4
MA	Mod Attributes	1			1	.2
MD	More Detail	24	6	2	32	6.2
MN	Maintainability	8	5	3	16	3.1
OT	Other	15	10	10	35	6.7
PD	Pass Data Areas		1		1	.2
PE	Performance	1	2	3	6	1.2
PR	Prologue/Prose	44	38	7	89	17.1 ⇦
RM	Return Code/Msg	5	7	2	14	2.7
RU	Register Usage	1	2		3	.6
ST	Standards					
TB	Test & Branch	12	7	2	21	4.0
		295	168	57	520	100.0
		57%	32%	11%		

WALKTHROUGHS & INSPECTIONS
- Two complementary processes.

WALKTHROUGHS.

OBJECTIVE:
Communicate Design/Code and rationale to others.

PROPERTIES:
- Process adjusted to needs.
- Driven by the author.
- Material can be at any level of completion.
- (Usually no exit criteria.)
- (Usually no defect checklist)
- (Usually no defect distribution.)

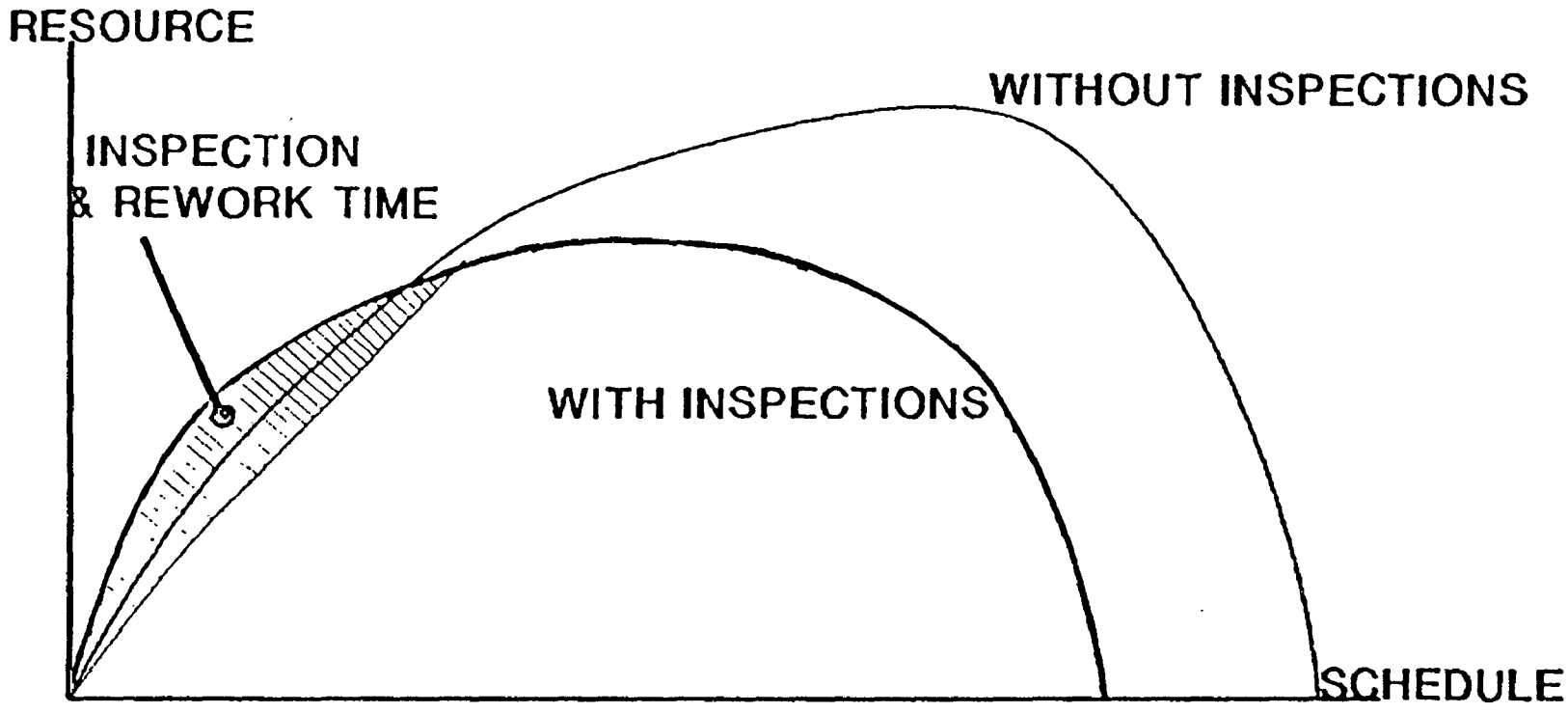
INSPECTIONS.

OBJECTIVE:
Verify that the product of an operation (requirements, design or code) meets exit criteria/ requirements of the operation.

PROPERTIES:
- Specific 6-Step Process.
- Managed by trained moderator.
- Material must meet detailed entry & exit criteria.
- Defect checklist.
- Defect distribution to improve defect detection.
- Defect report for follow-up, project management tracking, & pre-test defect management.

SCHEDULE/RESOURCE CHANGES DUE TO INSPECTIONS.

- INSPECTIONS USE 15% OF NET RESOURCE, AND ARE FRONT-END LOADED.
- INSPECTIONS REDUCE NET RESOURCE 10 - 40%, AND USUALLY SHORTEN (NEVER LENGTHEN) SCHEDULE.



INTRODUCING THE "DEFECT-FREE PROCESS" INTO YOUR ORGANIZATION.

REQUIRES:

- FORMAL PROCESS DEFINITION OF KEY OPERATIONS.
- RIGOROUS EXECUTION OF THE FORMALLY DEFINED PROCESS.
- FAGAN INSPECTIONS OF REQUIREMENTS, DESIGN AND CODE.
- PARTICIPATION IN *CONTINUOUS PROCESS IMPROVEMENT* BY MANAGERS AND DEVELOPERS - *BECAUSE THEY WANT TO.*

Michael Fagan (619) 456 - 9090

Paper 2-T-1

MOVING TOWARD DATA USE TESTING

Mr. William J. Bently
Miles Inc.

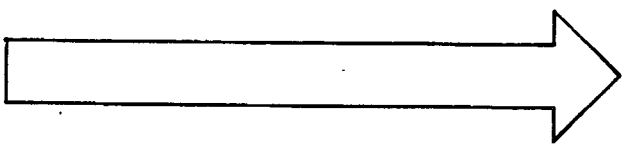
Mr. Bill Bently is in charge of the testing of the data management software products at Miles Inc. These innovative software products are used by physicians and medical laboratories for diabetes care and urinalysis. Previous to entering the biomedical computing field eight years ago, Mr. Bently headed a small R&D group which developed real-time 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.

MOVING TOWARD DATA USE TESTING

W. G. Bently
Miles Inc.

SOFTWARE QUALITY WEEK 1990
San Francisco

- Develop a series of coverage measures of increasing reliability
- Present BASIC CONCEPTS of data flow testing
- Introduce Cd graph and Cd testing
- MOVING TOWARD DATA USE TESTING



RELIABILITY

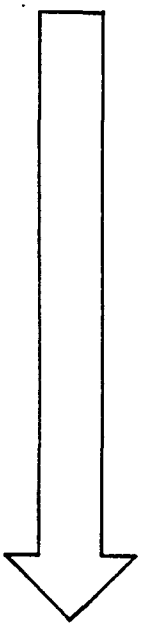
C1 Ct k=1 Cd Ct k=∞

OVERVIEW

WHAT COMBINATIONS OF PROGRAM SEGMENTS SHOULD BE "COVERED" IN ORDER TO ACHIEVE HIGH RELIABILITY ?

MOVING TOWARD PATH TESTING

STATIC ANALYSIS



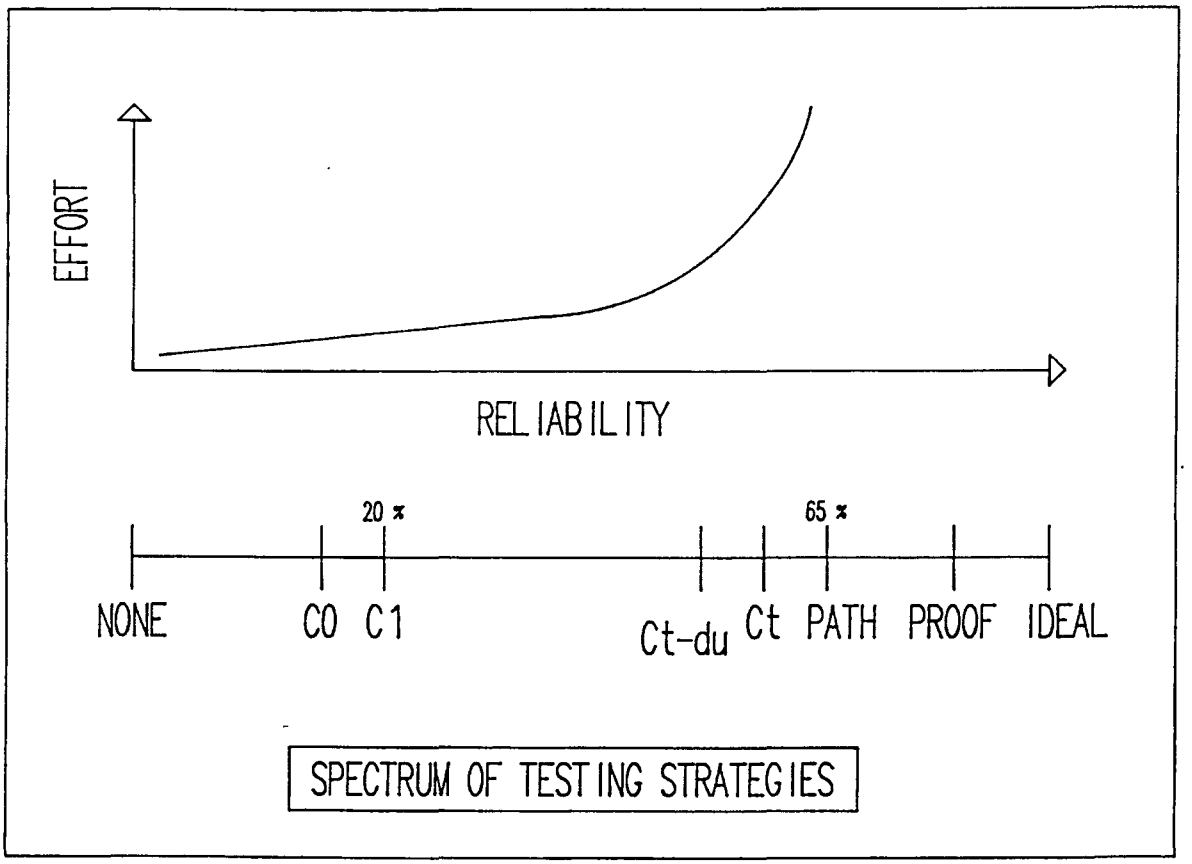
DYNAMIC ANALYSIS

PROGRAM VERIFICATION	discover all properties of all possible executions
DATA FLOW ANALYSIS	discover some specific properties of program
PROGRAM TESTING	execution of program over sample test data

TECHNIQUES OF SOFTWARE QUALITY ASSURANCE

PROGRAM TESTING STRATEGY	method for selecting input test data; partition of the input domain
PROGRAM-BASED TESTING	program testing strategy based on the analysis of source code
PROGRAM STRUCTURE	control flow data flow
COVERAGE ANALYSIS	measure the degree to which the input test data exercises some aspect of program structure

THE STURCTURAL APPROACH TO SOFTWARE TESTING



RELATIVE EFFECTIVENESS OF STRUCTURAL TEST METHODS

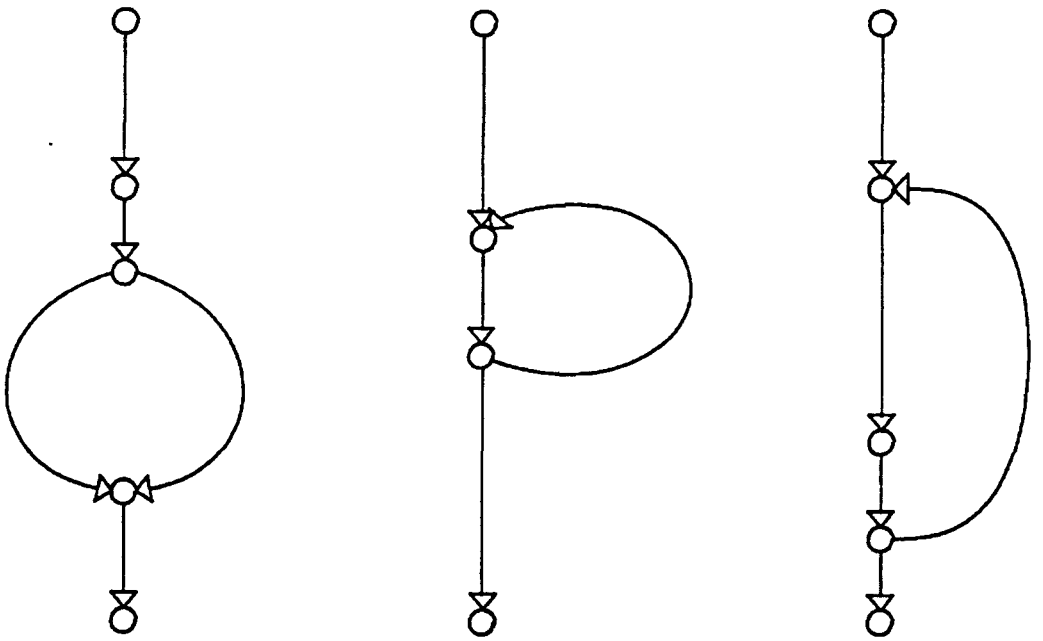
LIMITATIONS OF PATH TESTING

• *THEORETICAL LIMIT*	there is no general purpose procedure of testing and analysis for proving correctness
• *LARGE SIZE OF INPUT DOMAIN*	oracle
• *LARGE NUMBER OF PATHS*	loops
• *INFEASIBLE PATHS*	no algorithm

GOALS FOR PRACTICAL PATH TESTING STRATEGY

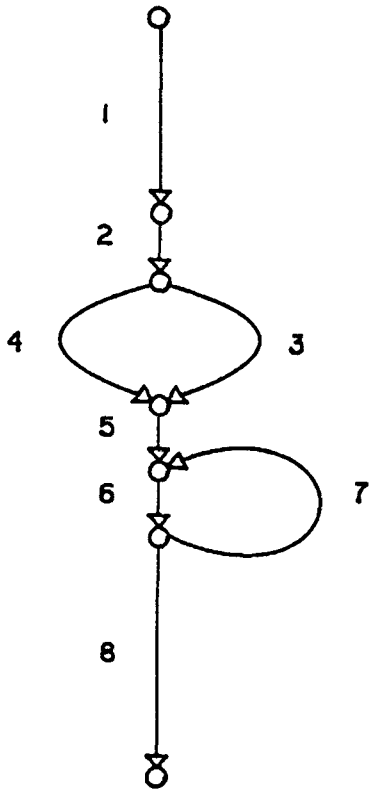
- *MINIMIZE SIZE OF TEST SET*
- *MAXIMIZE RELIABILITY*

PATH TESTING



conditional transfer statement is treated as separate segment

SEGMENT DEFINITION AUGMENTED
FOR DATA FLOW ANALYSIS



S1
IF (x)
    S3
ELSE
    S4
END
S5
WHILE (y)
    S7
END
S8

<u>Ct k=1 PATHS</u>

1 2 3 5 6 8

1 2 4 5 6 8

1 2 3 5 6 7 {< 7 >} 8

1 2 4 5 6 7 {< 7 >} 8

Ct PATH TESTING EXAMPLE

- Evolved from Ed Miller's level-i path testing strategy
- Enumerates all noniterative paths and maps iterative paths into finite set of path classes by specifying a minimum iteration count k
- Ct test coverage metric measures the proportion of Ct path classes exercised by a set of inputs
- An initial evaluation indicated that Ct k = 1 testing was feasible for testing the majority of C functions (73%) in a production program
- Provided clues to the development of a potentially more practical strategy which would restrict coverage to dependent segments. In one example, the number of Ct k = 1 paths was 20, but only 3 were necessary to cover all k = 1 du-pairs.
- Led to reexamination of Ed Miller's Cd testing strategy

Ct TESTING STRATEGY

MODELS PROGRAM AS A SERIES OF COMPUTATIONS

inputs ⇨ *intermediary values* ⇨ *outputs*

EARLY USE IN CODE OPTIMIZATION

FIRST USE IN TESTING WAS STATIC ANALYSIS FOR THE DETECTION OF DATA FLOW ANOMALIES (Fosdick and Osterweil 1976)

ANOMALY suspicious pattern of usage of a variable
xu dx dd

DATA FLOW ANALYSIS AS TESTING STRATEGY (Herman 1976, Ntafos 1984, Rapps and Weyuker 1985)

DATA FLOW TESTING

- ARRAYS
- POINTERS
- RECORDS
- ALIASING
- SIDE EFFECTS
- CALL BY REFERENCE
- RECURSION
- PATH INFEASIBILITY
- UNSTRUCTURED PROGRAMS
- INTERPROCEDURAL DATA FLOW

DIFFICULTIES IN STATIC DATA FLOW ANALYSIS

DEFINITION	assignment of value to a variable; memory or i/o "write"
UNDEFINITION	assignment of indeterminate value to a variable
USE	reference to the value of a variable; memory or i/o "read" (non-destructive)
P-USE	predicate-use; reference that appears in the predicate of a conditional expression
C-USE	computation-use

CLASSIFICATION OF EACH OCCURRENCE OF A VARIABLE

DEF-CLEAR PATH WITH RESPECT TO X	a path containing no definitions or undefinitions of X
REACHING DEFINITION	a definition of the variable X "reaches" a specific use of X if there is a def-clear path with respect to X from the definition to the use
LIVE VARIABLE	a definition of the variable X is <u>live</u> at specific point in the program if the definition of X reaches that point

SOME DATA FLOW TERMINOLOGY

- Information flow includes implicit flows in conditional statements (Cheheyl 1981)
- Static Information flow analysis (Bergeretti and Carre' 1985)
- dr-pair in which variable is referenced in branch condition (definition) and is referenced in one of the branches (use)
- Program slicing (Weiser 1984) *debugging*
- Program Dependence Graph (Ottenstein and Ottenstein 1984) *optimization*
- Program Dependence Graph (Korel 1987) *static testing*
- Program Dependence Network (Korel 1988) *error location via execution trace; node in graph represents instruction*

INFORMATION FLOW AND DEPENDENCE ANALYSIS

Cd CONCEPT

based on the notion that in chosing combinations of segments, it is necessary to test only those which consist of dependent segments

ASSUMPTIONS

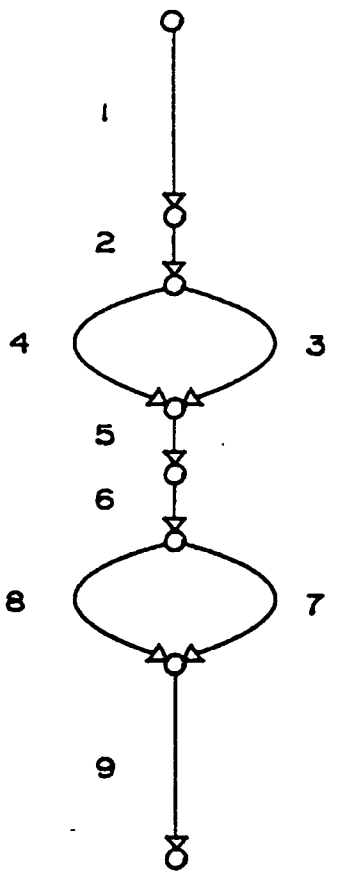
- An error is associated with a particular pattern of data flow
- Intraprocedural level
- Structured program
- All paths are feasible
- Error will be propagated and revealed in output(s)

Cd TESTING

NODE	represents the definition or use of a single variable
d$i(X)$	definition of variable X in segment i
u$i(X)$	use of variable X in segment i
pu$i(X)$	p-use of variable X in segment i

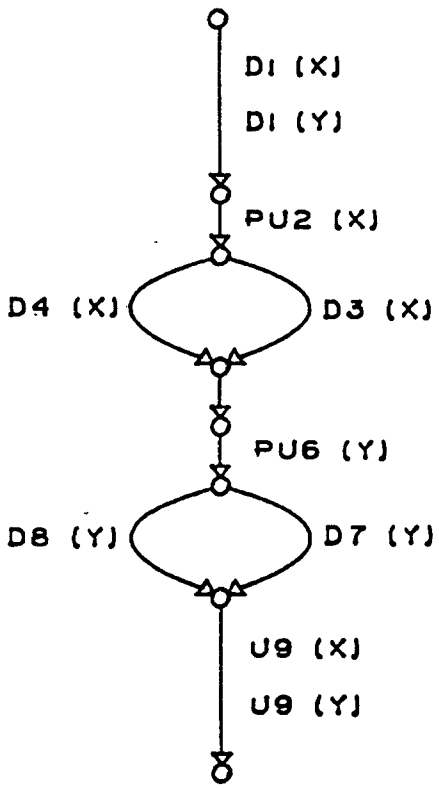
EDGE	ordered pair of nodes which represents flow relationship between two nodes
du-PAIR	a definition and use which appear in distinct segments. The first element is the definition of a variable and the second element is a use of the (same) variable.
ud-PAIR	a use and definition which appear in the same segment. The value of the variable in the first element (use) is referenced in the assignment of a value to the variable in the second element (definition)
uu-PAIR	

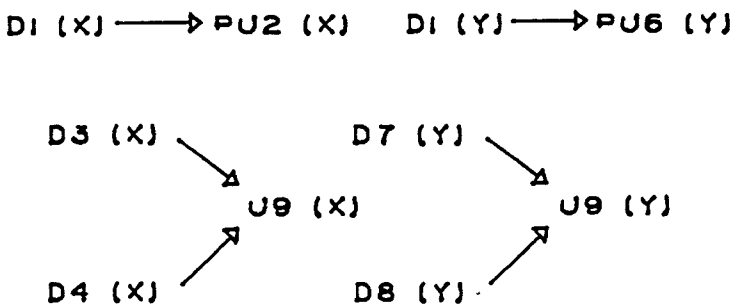
Cd FLOW GRAPH




```
1    INPUT X
     INPUT Y
2    IF (X > 0)
3          X = 1
     ELSE
4          X = 0
5    END
6    IF (Y > 0)
7          Y = 1
     ELSE
8          Y = 0
9    END
     PRINT X
     PRINT Y
```

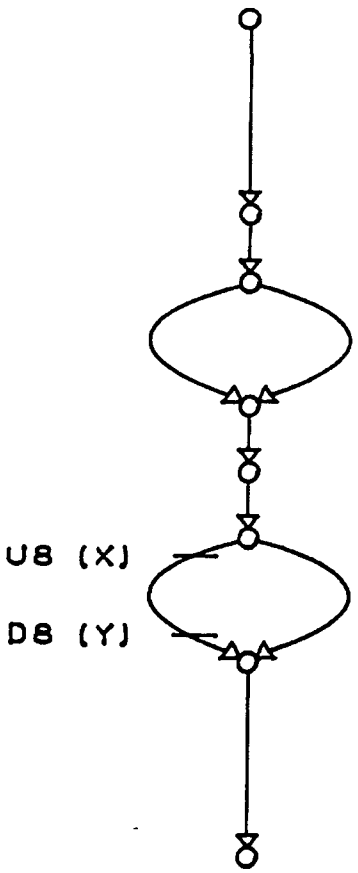
DIGRAPH OF EXAMPLE #1





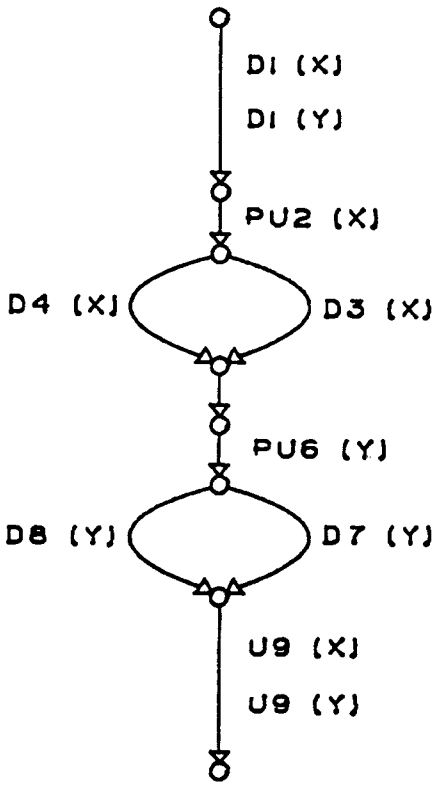
- Flow is from left to right (in each subgraph)
- Coverage of all edges in Cd flow graph *all-uses*

Cd FLOW GRAPH OF EXAMPLE #1




```
INPUT X
INPUT Y
IF (X > 0)
      X = 1
ELSE
      X = 0
END
IF (Y > 0)
      Y = 1
ELSE
      Y = -X
END
PRINT X
PRINT Y
```

DIGRAPH OF EXAMPLE #2



$d_1(X) - pu_2(X)$

$d_3(X) - u_9(X)$

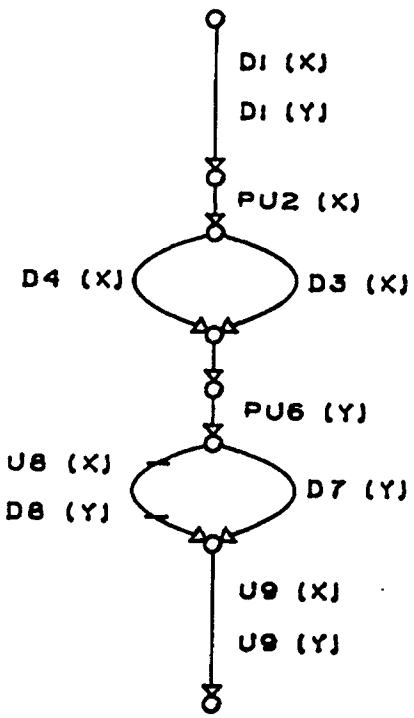
$d_4(X) - u_9(X)$

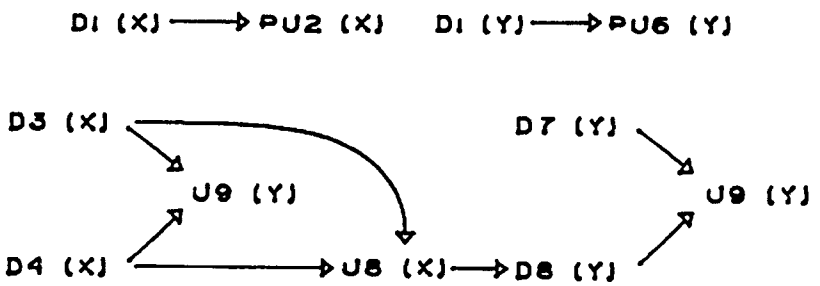
$d_1(Y) - pu_6(Y)$

$d_7(Y) - u_9(Y)$

$d_8(Y) - u_9(Y)$

du-PAIRS IN EXAMPLE #2



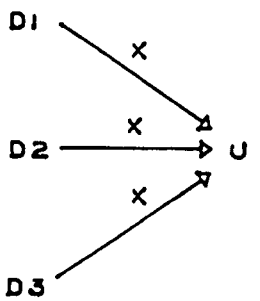


- du-chain → *Cd path*

Cd FLOW GRAPH OF EXAMPLE #2

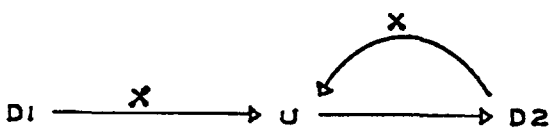
DECISION

The control structure determines which definition of a variable reaches a specific use of that variable.

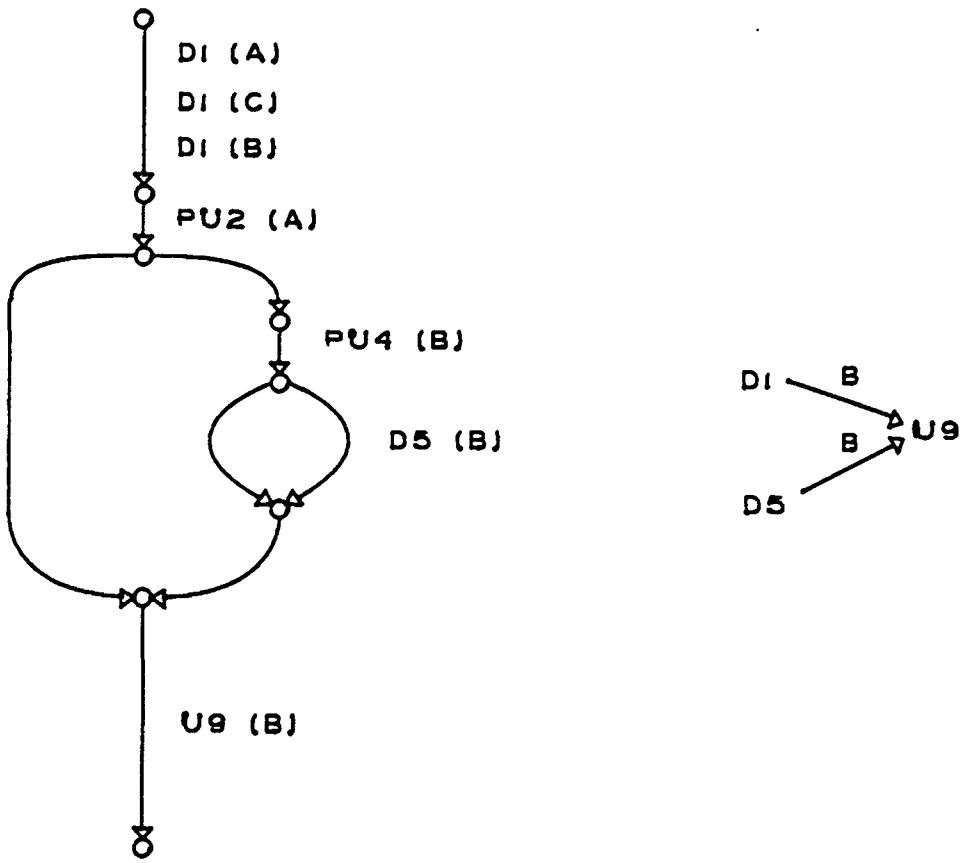


LOOP

The control structure determines which Cd paths are traversed in reaching a specific use.

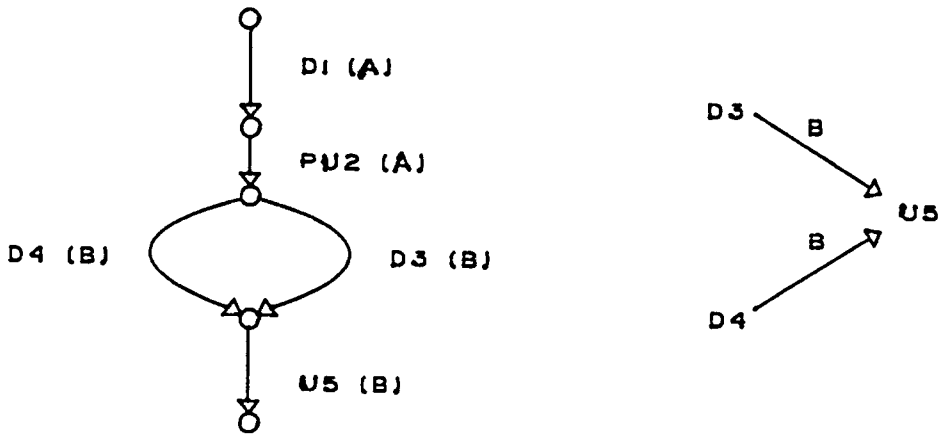


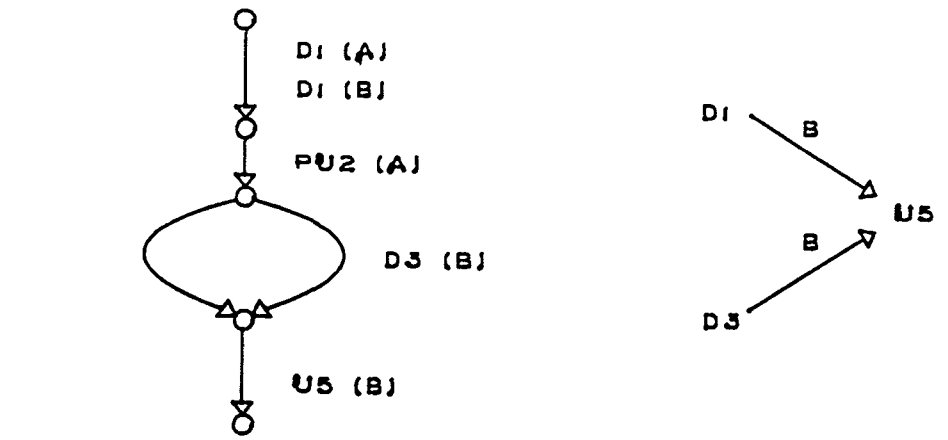
Cd VIEW OF INTERFACE BETWEEN DATA FLOW STRUCTURE AND CONTROL STRUCTURE OF PROGRAM

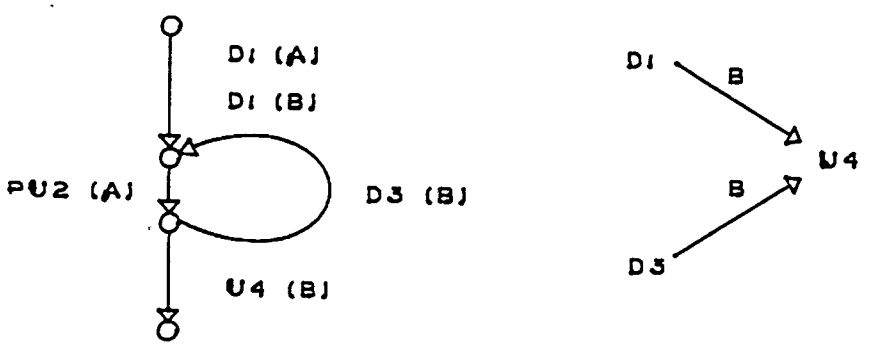


The predicate-use pu(a) can influence the use u(b) if it can prevent at least one definition of b from reaching u(b).

CONCEPT OF uu-PAIR







uu-PAIR AS INTERFACE BETWEEN THE DATA FLOW STRUCTURE AND THE CONTROL STRUCTURE OF A PROGRAM

CONTROL SCOPE

A definition or use is in the control scope of a predicate use pu(a) if pu(a) is in the conditional expression E that controls whether the segment containing the definition or use is executed or not executed.

EXAMPLES

IF E THEN $S1$ ELSE $S2$

WHILE E DO $S1$

DO $S1$ WHILE E

DEFINITION OF CONTROL SCOPE

The pair of uses pu(a) - u(b) is a decision uu-pair if:

- u(b) is a successor of pu(a) along some path

  AND

- u(b) is <u>outside</u> the control scope of pu(a)

  AND

- there is at least one definition of b within the control scope of pu(a) which reaches u(b)

  AND

- pu(a) is not the predicate of a DO_WHILE loop

  AND

- there is no ud-anomaly: i.e., there exist du-pairs d(a) - pu(a) and d(b) - u(b) for some d(a) and d(b)

DEFINITION OF DECISION uu-PAIR

The pair of uses pu(a) - u(b) is a loop uu-pair if:

- u(b) is a successor of pu(a) along some path

  AND

- u(b) is <u>inside</u> the control scope of pu(a)

  AND

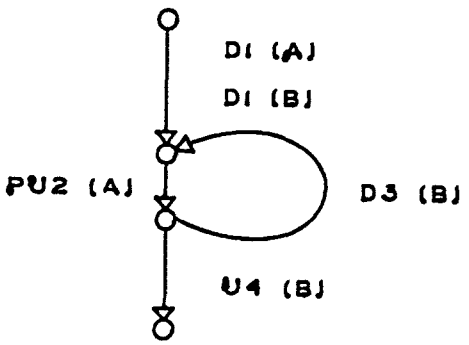
- there is at least one definition of b within the control scope of pu(a) which reaches u(b) only through pu(a)

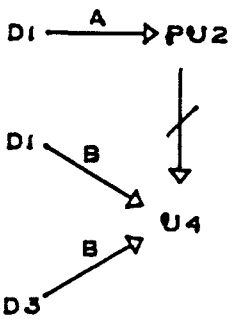
  AND

- there is no ud-anomaly

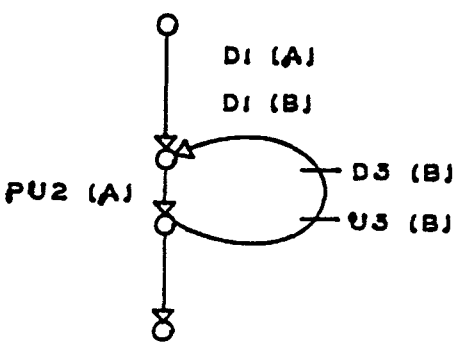
DEFINITION OF LOOP uu-PAIR

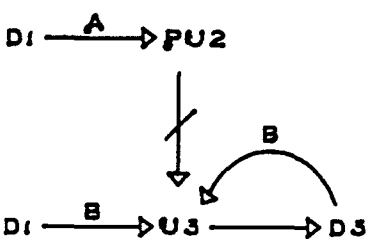
DECISION uu-PAIR



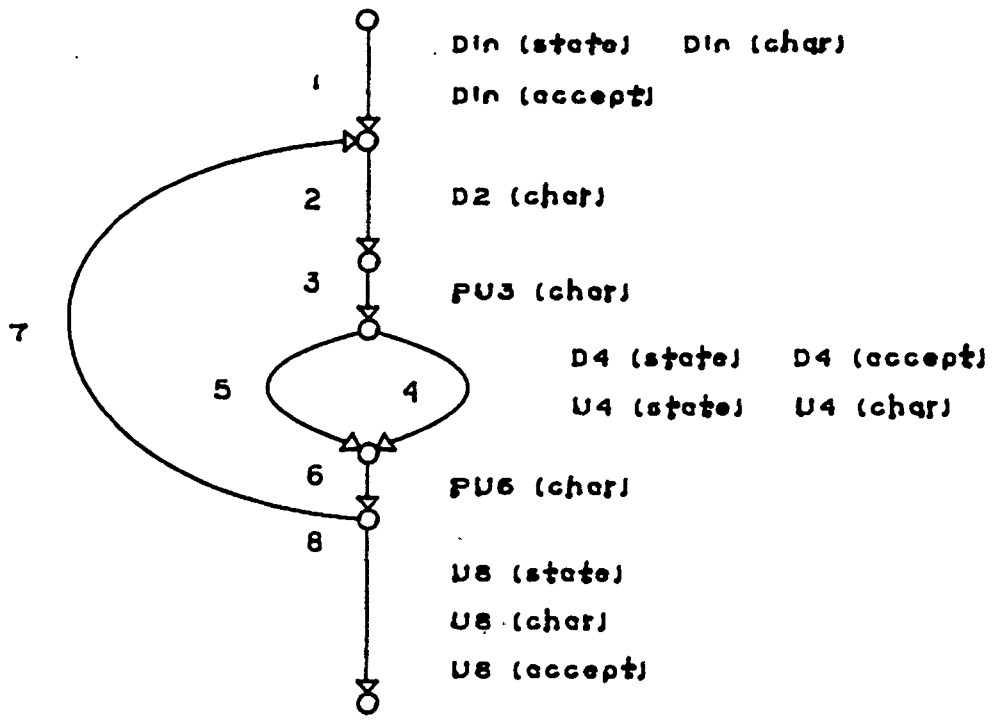


LOOP uu-PAIR





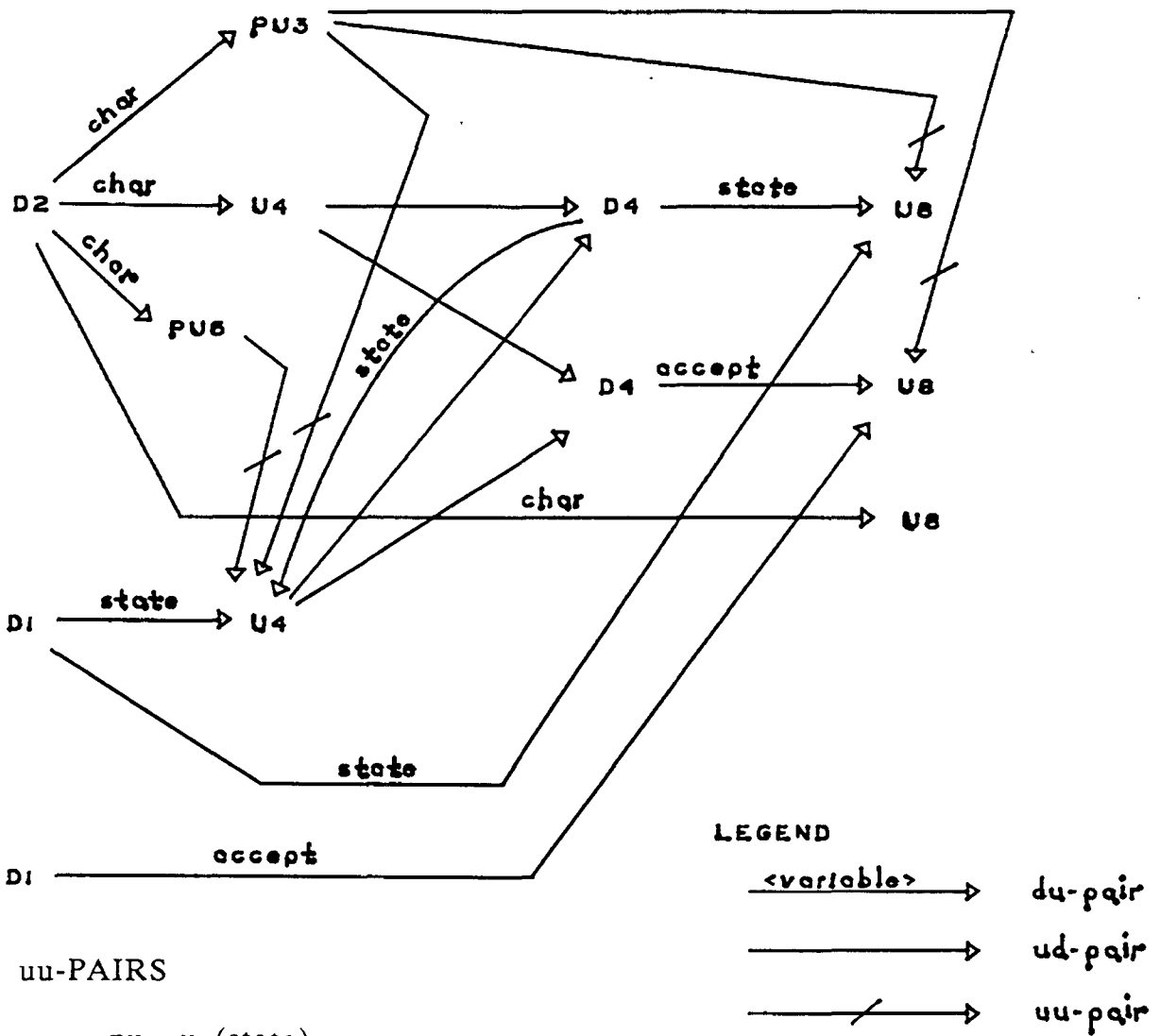
EXAMPLES OF uu-PAIRS



/ * adapted from Module M1 example in Clarke et al. (1986)
INITIAL, *BLANK* and *CR* are constants */

```
state = INITIAL
DO
        INPUT char
        IF (char IS NOT EQUAL TO BLANK OR CR)
                CALL fsa (char, state)
        / * fsa () RETURNS state AND accept AS OUTPUTS */
        END
WHILE (char DOES NOT EQUAL CR)
END
OUTPUT accept
```

DIGRAPH OF EXAMPLE #3



uu-PAIRS

pu_3 - u_8 (state)
pu_3 - u_8 (accept)
pu_3 - u_4 (state)
pu_6 - u_4 (state)

Cd FLOW GRAPH OF EXAMPLE #3

ONE	C1, edge of control flow graph
TWO	du-pair, edge of Cd data flow graph
FOUR	uu-pair (pair of du-pairs), interface between data flow and control flow in Cd flow graph
DATA FLOW	du-chain, path in Cd data flow graph
INFORMATION FLOW	Cd path

COMBINATIONS OF SEGMENTS TO TEST

1. Build automated testing tool that measures Cd coverage for C language programs

2. Evaluate the feasibility of Cd testing by performing a comparison of the number of test cases required in the testing of a production program for:

    - High C1 coverage
    - High Ct k = 1 coverage
    - High Cd coverage

3. Evaluate the reliability of the Cd testing strategy.

Cd RESEARCH STUDIES

Bently, W.G., and Miller, E.F., "Ct coverage - An initial evaluation," Proceedings of Quality Week Conference, San Francisco, CA, May 1989.

Bergeretti, J-F., and Carre', B.A., "Information-flow and data-flow analysis of while-programs," ACM Trans. Prog. Lang. and Systems, Vol. 7, No. 1, January 1985, pp. 37-61.

Cheheyl, M.H., Gasser, M., Huff, G.A., and Millen, J.K., "Verifying security," ACM Computing Surveys, Vol. 13, No. 3, September 1981, pp. 279-339.

Clarke, L.A., Podgurski, A., Richardson, D.J., and Zeil, S.J., "An investigation of data flow path selection criteria," in Proc. Workshop Software Testing, Banff, Canada, July 1986, pp. 23-31.

Fosdick, L.D., and Osterweil, L.J., "Data flow analysis in software reliability," Computing Surveys, Vol. 8, no. 3, Sept. 1976, pp. 305-330.

Frankl, P.G., "The use of data flow information for the selection and evaluation of software test data," Ph. D. Thesis, New York University, October, 1987.

Herman, P.M., "A data flow analysis approach to program testing," The Australian Computer Journal, Vol. 8, no. 3, November 1976.

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

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

Howden, W.E., "Symbolic testing - Design techniques, costs and effectiveness," U.S. National Bureau of Standards GCR77-89, National Technical Information Service PB-268517, Springfield, VA, 1977.

Korel, B., "PELAS - Program error-locating assistant system," IEEE Trans. Software Eng., Vol. 14, September 1988, pp. 1253-1260.

Korel, B., "The program dependence graph in static program testing," Information Processing Letters, Vol. 24, 1987, pp. 103-108.

Laski, J.W., and Korel, B., "A data flow oriented program testing strategy," IEEE Trans. Software Eng., Vol. SE-9, no. 3, May 1983, pp. 347-354.

Miller, E.F., and Howden, W.E., eds. "Tutorial: Software Testing and Validation Techniques," IEEE Computer Society, 1981.

Ntafos, S.C., "On required element testing," IEEE Trans. Software Eng., Vol. SE-10, no. 6, November 1984, pp. 795-803.

Ottenstein, K.J., and Ottenstein, L.M., "The program dependence graph in a software development environment," ACM SIGPLAN Notices Vol. 19, no. 5, May 1984, pp. 177-184.

Paige, M.R., "On partitioning program graphs," IEEE Trans. Software Eng., Vol. SE-3, November 1977, pp. 386-393.

Rapps, S., and Weyuker, E.J., "Selecting software test data using data flow information," IEEE Trans. Software Eng., Vol. SE-11, no. 4, April 1985, pp. 367-375.

Weiser, M., "Program slicing," IEEE Trans. Software Eng., Vol. SE-10, no. 4, 1984, pp. 352-357.

BIBLIOGRAPHY

Paper 2-T-2

NUMBER OF TESTS FOR THE "ALL DU PATHS" CRITERION

Prof. J. Bieman
University of Wyoming

Dr. James M. Bieman is Associate Professor of Computer Science at Colorado State University, Fort Collins, Colorado. He was previously an Assistant Professor at Iowa State University. Dr. Bieman earned the Ph.D. and M.S. in Computer Science at the Center for Advanced Computer Studies (USL) in Lafayette, LA, an M.P.P. (Public Policy) from the University of Michigan, and a B.S.Ch.E.(Chemical Engineering) from Wayne State University in Detroit. Dr. Bieman's research is focused on the structural analysis of software,software measures, software testing, and executable specification languages. He has developed software measurement tools to estimate the required number of test cases necessary to meet data flow testing criteria. Results of a case study of a commercial software system suggest that the strongest of the data flow criteria the all-du-paths criterion, is much more practical than the theoretical results indicate. His research on executable software specifications demonstrate that specification assertions can be effectively expressed within executable type expressions. He is a principal investigator on a NATO funded international research team investigating "Formal Foundations of Software and Systems Measurement." Dr. Bieman (with J. Leszczylowski) designed the PROSPER executable specification language.

Estimating the Number of Test Cases Required to Satisfy Data Flow Testing Criteria

James M. Bieman
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA

References:

- J. Bieman & J. Schultz, "Estimating the Number of Test Cases Required to Satisfy the All-du-paths Testing Criterion," *Proc. ACM TAV3-SIGSOFT89.*
- J. Bieman & J. Schultz, "An Empirical Evaluation of the All-du-paths Testing Criterion," TR #CS-89-118, Dept. of Computer Science, Colorado State Univ.

Outline

- Motivation
- Case Study
- Analysis tools
- Case Study Data
- Results

Testing Goal

General Goal: Find the smallest number of test cases that uncover as many errors as possible.

Our Goal: Determine the practicability of structural testing criteria.

Given a testing criterion — how many test cases are required?

We focus on complexity of criteria.

Structural Testing

Common criteria:

- all nodes, all edges, all acyclic paths.
- all acyclic paths + once thru each loop,
- test each "linear independent" path,
- all paths

Data Flow Criteria:

du-path: a definition-free path from where a variable is set to where it is used

all-uses criteria: test **at least one** du-path for all uses of each variable definition.

all-du-paths criteria: test **all** du-paths in a program, for all variables.

- Number of du-paths in a program is bounded by 2^t. [Weyuker84]

Case Study

How many du-paths in real programs?

How many test cases needed for du criteria?

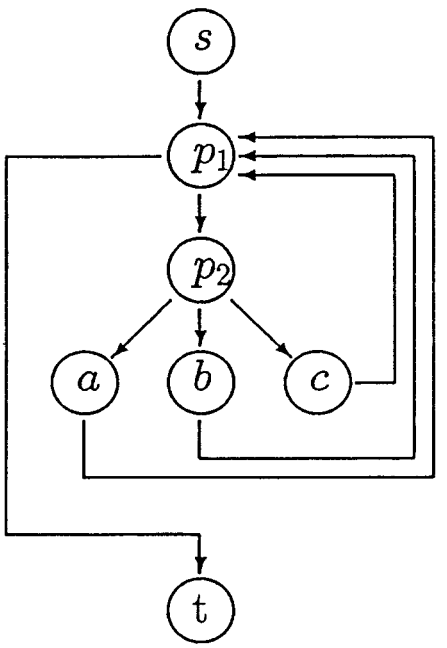
The experiment:

1. Formally specify the all-du criterion.
2. Specify method to estimate number of required complete paths to meet criterion.
3. Translate formal specifications into executable specifications.
   – Prolog rules
4. Apply to industrial software.
   (a system implemented in Pascal)

Domain of Criteria Specification

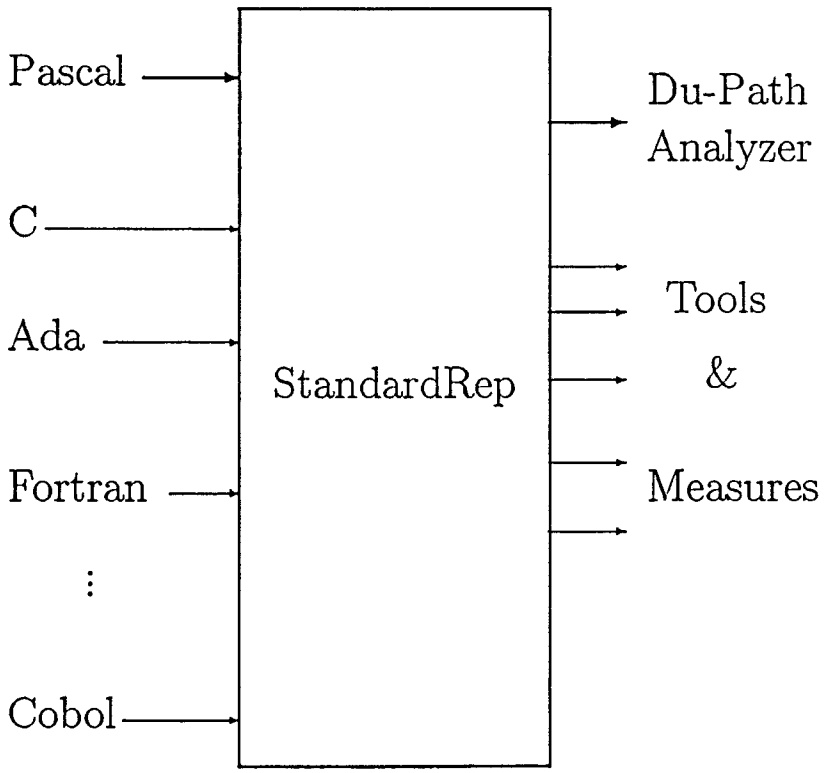
StandardRep of imperative programs

- Formally defined as a control flowgraph (using SPECS)



- Nodes contain additional info:
  variables defined/used,
  functions/procs invoked, etc.

Research Environment



Pascal $\longrightarrow$ StandardRep

- Full ISO Pascal.
- Uses YACC and LEX.
- Credit Kyung-Goo Doh
  M.S. Project.
- For du-path analysis
  - parameters set at proc/function entry (in start node).
  - arguments in proc/function calls
    - * assume they are referenced
    - * if they bind to var parameters assume they are set.

Prolog Data Base (PDB)
Abstract Specification

For each flowgraph node:

- o – variables set
- o – variables with a global c-use
- o – variables with a global p-use

$$
\begin{aligned}
PDB = \; & tuple(\\
& \quad Nodes : set\ of\ NodeID, \\
& \quad DCP : set\ of\ DCPType, \\
& \quad Edges : set\ of\ EdgeType)
\end{aligned}
$$

$$
\begin{aligned}
DCPType = \; & tuple(\\
& \quad NID : NodeID, \\
& \quad D : set\ of\ VarID, \\
& \quad C : set\ of\ VarID, \\
& \quad P : set\ of\ VarID)
\end{aligned}
$$

Example PDB

```
nodes([s,1,2]).

global_defs(s,[input]).
global_c_uses(s,[]).
p_uses(s,[]).

global_defs(1,[x,y]).
global_c_uses(1,[y]).
p_uses(1,[x,y,2,3]).

global_defs(2,[stop,10,x,z]).
global_c_uses(2,[]).
p_uses(2,[stop,100]).

edge(s,1).
edge(s,2).
edge(1,2).
edge(2,t).
```

PDB Generator Spec

function ProducePDB (UR : UnitRepType,
SR : StandardRep) : PDB

pre: UR $\in$ SR

post:

$$
\begin{aligned}
& Nodes(ProducePDB(UR, SR)) = \\
& \{x : NodeID \mid \exists n[n \in Nodes(UFS(UR)) \wedge NID(n) = x]\} \\
& \wedge DCP(ProducePDB(UR, SR)) \\
& = \{x : DCPType \mid \exists n[n \in Nodes(UFS(UR)) \wedge \\
& \quad NID(n) = NID(x) \wedge \\
& \quad D(x) = \{v \mid GlobalDefs(n, v, SR)\} \wedge \\
& \quad C(x) = \{v \mid GlobalCUses(n, v, SR)\} \wedge \\
& \quad P(x) = \{v \mid PUses(n, v, SR)\}]\} \\
& \wedge Edges(ProducePDB(UR,SR)) = Edges(UFS(UR))
\end{aligned}
$$

Prolog Program Count Algorithm

1. Find all du-paths in PDB. Output number of du-paths.
2. Find successor nodes for each node.
3. Remove redundant du-paths found in Step 1. Output number of remaining du-paths.
4. Determine cardinality of "small" set of complete paths that cover all du-paths from Step 3. Output this number.

Finding du-paths

For **each** ordered pair of nodes (x, y)

1. Find the set of common variables that are defined in x and referenced in y.
2. If this set is non-empty, conduct a depth first search for du-paths from x to y.
   - At least one common variable from Step 1 must not be redefined on a du-path.
   - Each path found is output.

p-uses are associated with the successor nodes of x – such du-paths include successors.

There are at most $n(n-1)$ distinct node pairs.

du-path representation
Prolog list of lists

```
du_paths([[1,2,3],
          [1,2,4],
          [1,3,4],
          [1,2,4,5],
          [1,3,4,5],
          [6,5],
          [6,5,6],
          [6,5,7]]).
```

Counting Complete Paths

```
DUP := set of condensed du-paths;
SP := [s]; {search path}
Count := 1;
While DUP is non-empty do
 if SP = [N] {contains a single node}
 then if there is a du-path starting
          with a successor to N
      then 1. choose a path P which
              starts with the closest
              successor;
           2. DUP := DUP - {P};
           3. SP :=  tail(P);
      else 1. increment Count;
           2. SP := [s];
 else if there is a du-path P
           such that SP is a prefix
      then 1. D := D - {P};
           2. SP := tail(P);
      else SP := tail(SP);
end While.
```

Case Study Data
NLTAS: Natural Language Text Analysis System

- analyzes verbatim responses to open ended surveys
- identifies words and phrases that belong to specified "meaning units"
- used in marketing research since 1985
- product of IRIS Systems Inc.
- statistics
  - 7,413 lines of Pascal
  - 143 subroutines
  - ave subroutine is 52 lines
  - longest subroutine is 367 lines
  - all but 10 subroutines are less than 100 lines

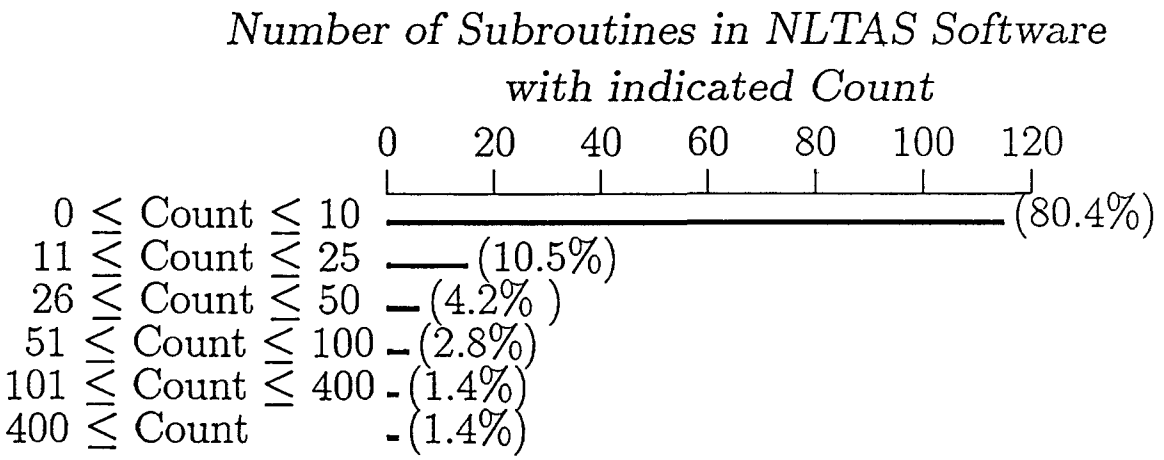
Recorded Data
for each NLTAS suboutine

1. number of lines of code
2. number of flowgraph nodes
3. number of flowgraph edges
4. number of du-paths
5. number of non-redundant du-paths (Condensed)
6. estimated number complete paths required to meet the all-du-paths criterion (Count)

Results: to satisfy the all-du-paths criterion

- 115 of 143 subroutines (80%) require 10 or fewer tests!
- 91% require 25 or fewer tests.
- One subroutine requires on the order of 10,000 tests.
- One subroutine requires on the order of 2^{32} tests.
- Using the weaker all-uses criterion, all subroutines can be tested with a tractible number of tests.

Required Number of Complete Paths



Code Structure of A18
A Procedure with Exponential Count

Define X: X := Y;
if P_1 then S_1;
if P_2 then S_2;
$\vdots$
if P_{32} then S_{32};
Use X: Y := F(X);

All-uses Criterion
Applied to A18 and A59

Program Unit	Du-Paths	Condensed	Count
A59	346	139	28
A18	672	568	463

Contributions

- Estimator of required number of test cases for a criteria.
  - all-du-paths
  - adaptable to other criteria (all-uses)
- Case Study indicates
  - all-du-paths criterion more realistic than expected
  - few routines require too many test cases.
- Formal specification of criteria and tools allows wide use of tools.
  - Our prolog implementation is really an executable spec.

Paper 2-T-3

THE EFFECTS OF HARDWARE ADVANCES ON SOFTWARE QUALITY ASSURANCE

Mr. Doug Hoffman
MasPar Computer Corporation

Mr. Doug Hoffman is the Manager of Software Quality Assurance at MasPar Computer Corporation in Santa Clara. He has been in the QA field for over 16 years, working for such companies as Informix Software, Pyramid Technology and Hewlett Packard. He received his MBA from Santa Clara University and his MSEE from UC Santa Barbara and Santa Cruz.

The Effects of Hardware Advances on Software Quality Assurance

Some Impacts on Software Testing

Douglas Hoffman

MasPar Computer Corporation

MasPar Computer Corporation

Family of Massively Parallel Computer Systems

Single Instruction, Multiple Data (SIMD)

1,600 to 26,000 MIPS

1K to 16K Processors

$170,000 to $810,000

UNIX Operating System

Trends in Software Programs

Programs are getting larger

Programs are getting more complex

Customers are requiring more reliability

===> SQA is getting more difficult and more important

Trends in Software Test Methods

More Formal Testing

More Tools for Analysis

More Tools for Testing

Today's Execution Speeds

Hermann Kopetz example in 1976:

"3 inputs of 16 bits" ...
"would require aproximately 900 years of CPU time"

Assuming he was using about a 1 MIP system then:

900/3200 = 100 days on same price system today

900/26,000 = 12 days on large system today

Hermann Kopetz, "Software Reliability" (Macmillan Press, Springer-Verlag New York, 1979) p. 55.

Today's Execution Speeds

Doug Hoffman example in 1990:

"1 input of 32 bits for floating square root requires 6 seconds of CPU time"

Assuming a 2K PE system:

50ms * 2**31 cases / 2**11 processors

= 50 seconds on same price system

= 6 seconds on large system

Effect on SQA Testing

More Complex Systems Under Test

How We Design and Develop Tests

How Much We Can Test

Input/Output Mapping

```
                  ------------------
                  |                |
INPUT ---->|  FUNCTION  |----> OUTPUT
                  |                |
                  ------------------
```

Equivalence Classing of Inputs

Selecting Subgroup Classes

Mapping all Inputs Into Subgroups

Sele cting Representative Values from each Subgroup

The Set of Representative Values is used as Inputs

MasPar Experience

Exhaustive Testing of Modules

32 bit sqrt all cases

64 bit sqrt all cases in classes

other math functions

Side Effects of More Exhaustive Testing

Test Effectiveness Increases

Test Design Time Decreases

Test Development Time Decreases

Automatic Test Generation Easier

Test Reliability May Increase

Development Debug Time Decreases

QA Areas Which Are Impacted

Black Box / White Box Testing

Basic Test Design

Statistical Sampling

Alternatives to Testing

Areas Which Fit Exhaustive Testing

Basic Math Functions

Machine Instruction Sets

Simple Library Functions

Complex functions with limited Inputs/Outputs

Modules of Large Programs

Considerations

Not Everyone Has Access To Massively Parallel Systems

Not All Problems Fit The Model

Conversion Is Likely To Be Slow

Machines Are All Getting Much Faster

New Technology Is Growing Fast

Paper 2-T-4

EMBEDDED SOFTWARE TESTING

Mr. Bruce Ableidinger
Cadre Technologies, Inc.

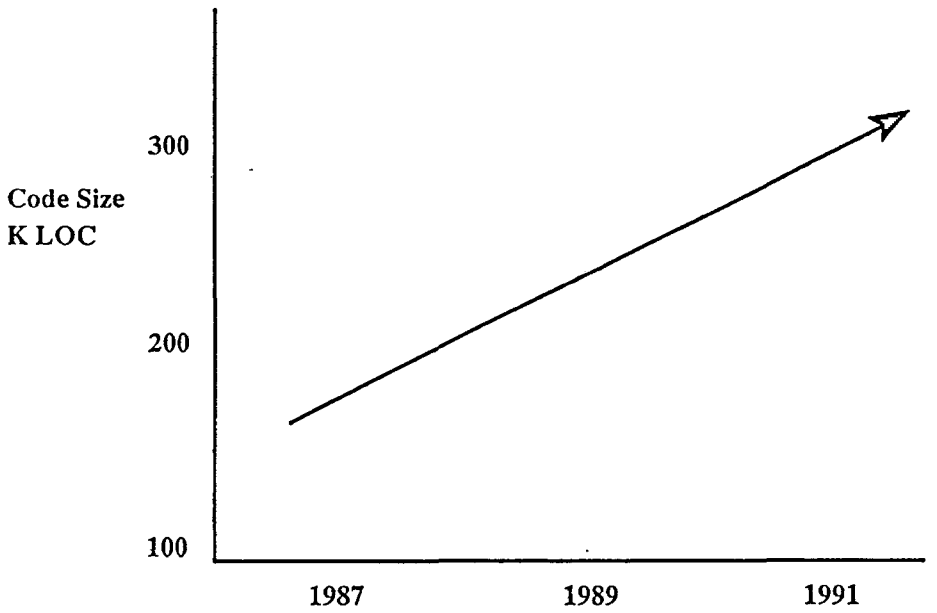
Mr. Bruce Ableidinger is Cadre Technologies' Chief Systems Architect. Bruce joined Northwest Instrument Systems, which has since merged with Cadre, in 1982. He was the lead design engineer for the development of their Logic Analysis Workstation and Software Analysis Workstation products. He is presently working on "TaskView" - a hardware/software package used to view the execution of an Ada multi-tasking run-time system. Mr. Ableidinger previously worked for Intel and Tektronix on logic analyzer, emulation, and microprocessor development system products. He received a BSEE from Washington State University in 1975, and a MSCS from Oregon State University in 1982.

Embedded Software Testing

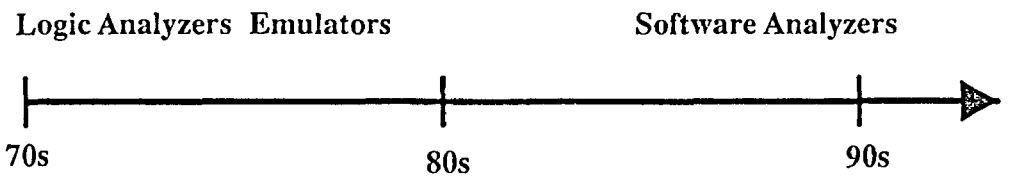
Bruce Ableidinger
Chief Systems Architect

Cadre Technologies Inc.

Applications Are Increasingly More Software Complex



Tool Evolution



Changing Tool Views 70's

Binary	Disassembly
10101	Move.B D0, 1000
00011	DEC.B D0
11010	BEQ.L 4000
10111	Move.L A0, 6000

Changing Tool Views 80's

Annotated Disassembly		Source Code Trace
Move.B	D0, foo	foo—;
DEC.B	D0	Move.B D0, foo
BEQ.L	PROC_B	DEC.B D0
Move.L	A0, POINTER 1	if (foo == 0) PROC_B
		BEQ.L PROC_B

Changing Tool Views 80's (continued)

Code Coverage

PROC_A	90%
PROC_B	80%
PROC_C	95%

Procedural Trace

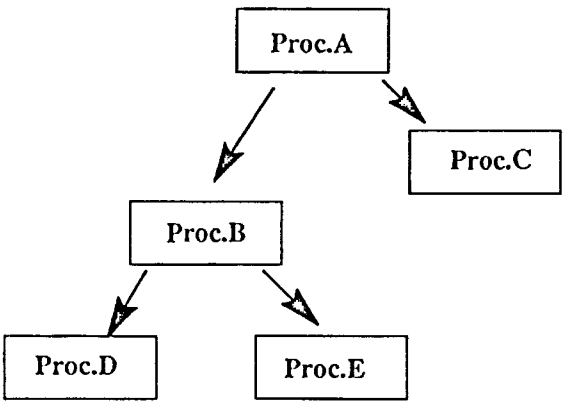
PROC_A	Entry
PROC_B	Entry
PROC_B	Exit
PROC_C	Entry
PROC_C	Exit
PROC_A	Exit

Performance Analysis

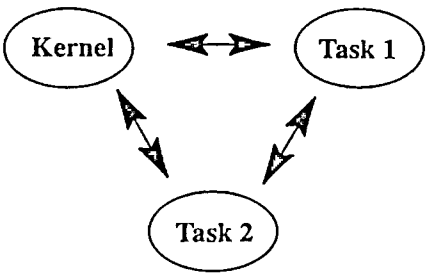
	min.	max.
PROC_A	10ms	25.3ms
PROC_B	125ms	226ms
PROC_C	700ms	953ms

Changing Tool Views 90's

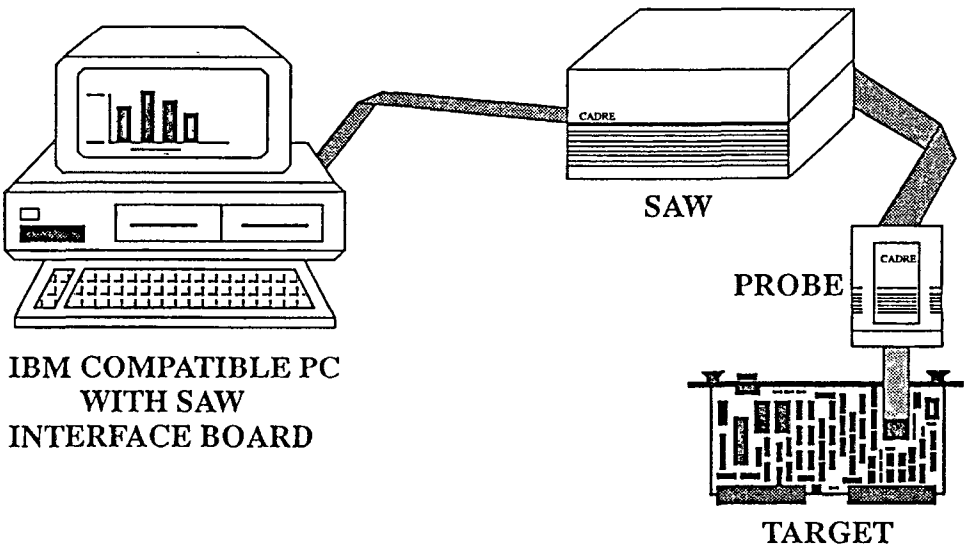
Structural Representation



Operating System Interactions



Software Analysis Workstation



Annotated Disassembly Trace

```
File  Edit  Display  Filter  Setup  Events  Control          State Analysis
ISA1 - State Analysis MicroTrace - ACQ Memory
Status:  STOPPED at state <1> on measurement complete
t to c:  10.00 uS                                        r to c:  10.00 uS
LOC        Sym/Events                                   Mnemonic
           Hex
0040 c     ar_debugger_support+65C0 UP: BSR calendar
0041       ar_debugger_support+65C4 UP:?  <word> read & MOVE.L (****,A7),D7
0042       calendar                 UP: MOVEM.L A3-A4,-(A7)
0043       ctr_pkg.Line_19+0008E090 UD:   <long> write
0044       calendar            +0004 UP: LEA ctr_pkg.Line_19+FFF87558,A4
0045       ctr_pkg.Line_19+0008E08C UD:   <long> write
0046       ctr_pkg.Line_19+0008E088 UD:   <long> write
0047       calendar            +0008 UP:   <word> read & MOVE.B (0000,A4),D0
0048       calendar            +000C UP:   <word> read & LEA calendar+0421,A3
0049       calendar            +0010 UP:   <long> read
0050       ctr_pkg.Line_19+FFF87558 UD:   <hi byte> read
0051       calendar            +0014 UP: MOVE.B (0001,A4),-(A3)
0052       calendar            +0018 UP: MOVE.B (0002,A4),-(A3)
0053       ctr_pkg.Line_19+FFF87559 UD:   <hi byte> read
0054       calendar            +001C UP: MOVE.B (0003,A4),-(A3)
0055       calendar            +0420 UD:   <hi byte> write
0056       ctr_pkg.Line_19+FFF8755A UD:   <hi byte> read
```

Procedure Level Trace

File Edit Display Setup Events Control SymTrace

SFA1 - SymTrace List - DBG3_2.STA

Status: STOPPED after trigger Run Time: 00:19:23

t to c: 367. uS r to c: 356. uS

LOC	Event	Type	rel Time	Data
Trig	array_index	dwrdb	7.00 uS	0000
0001	set_light_state	entry	2.80 uS	
0002	set_light_state	exit	1.60 uS	
0003 r	put_light_state	entry	3.00 uS	
0004	put_light_state	exit	353. uS	
0005 c	array_index	dwrdb	1.09 S	0001
0006	set_light_state	entry	2.80 uS	
0007	set_light_state	exit	1.60 uS	
0008	put_light_state	entry	3.00 uS	
0009	put_light_state	exit	353. uS	
0010	array_index	dwrdb	6.21 S	0002
0011	get_event	exit	2.60 uS	
0012	io_event	dbytb	197. uS	x
0013	array_index	dwrdb	4.60 uS	0002
0014	valid_car_event	entry	4.40 uS	
0015	valid_car_event	exit	565. uS	
0016	get_event	entry	494. mS	
0017	get_event	exit	2.40 uS	

Synchronous Traces

File Edit Display Setup Events Control SymTrace

SFA1 - SymTrace List

Status: STOPPED on measurement c

t to c: 201. uS r to c: 201.

LOC	Event
-0008	intr_ctr.Line_193
-0007	intr_ctr.Line_199
-0006	intr_ctr.Line_220
-0005	intr_ctr.Line_191
-0004	intr_ctr.Line_193
-0003	intr_ctr.Line_199
-0002	intr_ctr.Line_220
-0001	put_light_state
Trig r	io_event
0001	intr_ctr.Line_241
0002	intr_ctr.Line_243
0003 c	valid_xing_req
0004	valid_xing_req
0005	intr_ctr.Line_191
0006	intr_ctr.Line_193
0007	intr_ctr.Line_199
0008	intr_ctr.Line_220
0009	get_event

ISAL - State Analysis MicroTrace - ACQ Memory

Status: STOPPED at state (1) on measurement

t to c: 201.1 uS r to c: 201.1 uS

LOC	Address Hex	Mnemon
0764	000B6B9C	UD: (word) write
0765 c	000286E4	UP: LINK A6,#FFF4
0766	000B6B98	UD: (long) write
0767	000286E8	UP: NOP & NOP
0768	000B6B94	UD: (long) write
0769	000286EC	UP: MOVE.L A7,(FFFC,A6
0770	000286F0	UP: CLR.W (FFF8,A6)
0771	000B6B90	UD: (long) write
0772	000286F4	UP: CMPI.W #0003,(0008
0773	000286F8	UP:! (word) read & BN
0774	000B6B8C	UD: (word) write
0775	000B6B9C	UD: (word) read
0776	000286FC	UP: (word) read & CM
0777	00028700	UP: (long) read
0778	00028704	UP: BNE 00028768
0779	000B6B9E	UD: (word) read
0780	00028708	UP:? (word) read & MO

Performance Analysis

File Edit Display Setup Events Measurement Control Perf Analysis

SFA1 - Performance Analysis PGA Display - PGA.PAA

Status: STOPPED on command Run Time: 00:01:26

Event	Time	Number of Occur	Minimum Time	Maximum Time	Percent of Total T (0 10 15 25)
get_event	225.6 uS	60	2.000 uS	5.600 uS	28.6
put_light_stat	45.20 uS	13	3.200 uS	3.800 uS	5.7
set_light_stat	46.20 uS	13	3.200 uS	3.800 uS	5.8
valid_car_even	32.40 uS	7	4.400 uS	5.600 uS	4.1
threshold_reac	17.00 uS	4	4.200 uS	4.400 uS	2.1
valid_xing_req	66.60 uS	13	4.200 uS	6.800 uS	8.4
event_dispatch	353.4 uS	62	1.400 uS	8.600 uS	44.9
Total:	786.4 uS	172	--	--	100.0

Test Coverage

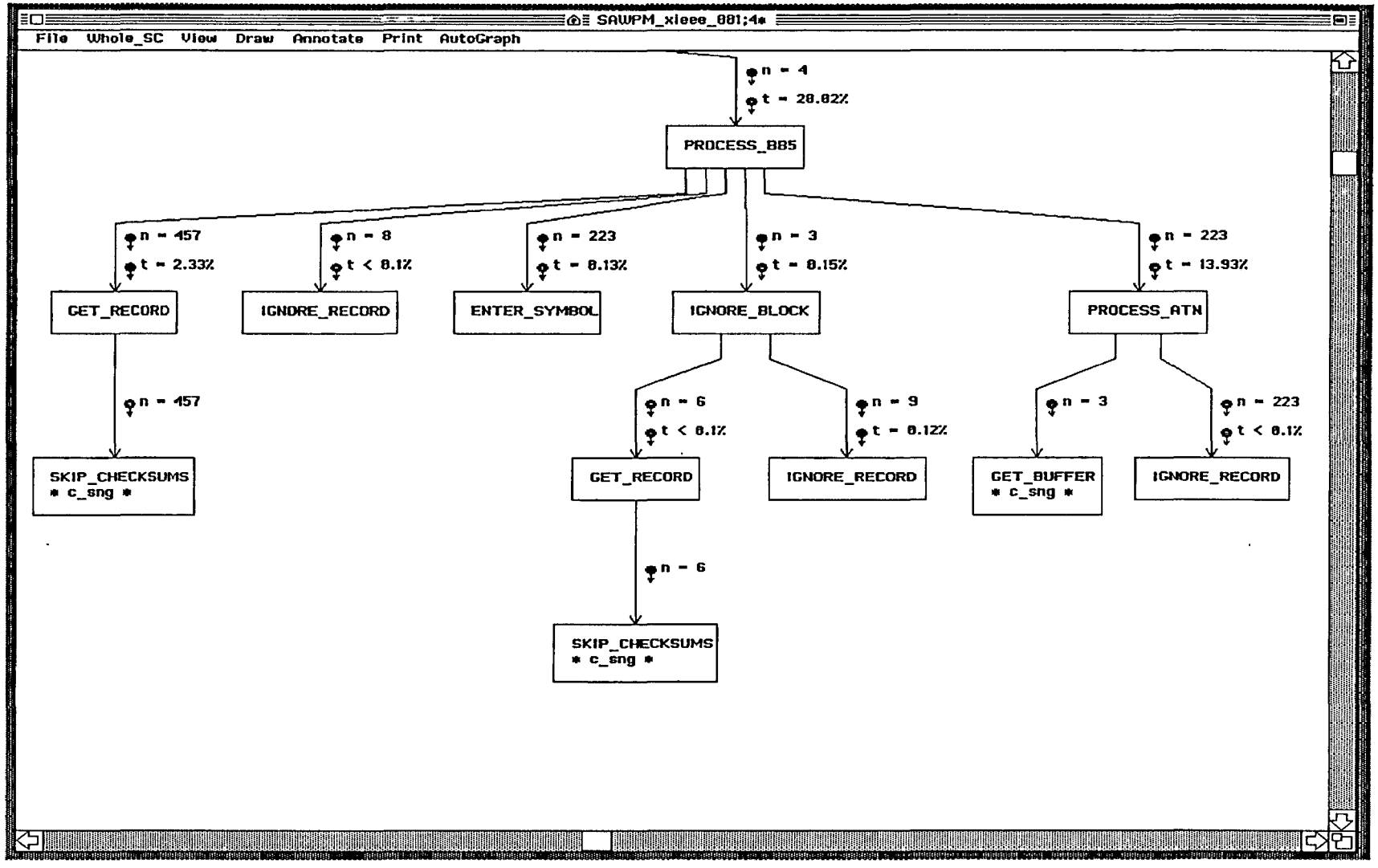
File Edit Display Setup Events Control CodeMap

SFA1 - CodeMap Table Mapped and Covered - CM_2.CMA

Status: STOPPED on command 79.6% Run Time: 00:00:47

Absolute	Program Symbol	
00028146-0002815F	event_pkg.event_dispatcher.Line_23	
00028160-00028187	event_pkg.event_dispatcher.Line_25	
000281E8-000281E9	event_pkg.event_dispatcher.Line_25	+ 000
000281EA-00028213	event_pkg.event_dispatcher.Line_27	
00028214-0002823D	event_pkg.event_dispatcher.Line_28	
0002823E-00028265	event_pkg.event_dispatcher.Line_29	
00028266-0002828F	event_pkg.event_dispatcher.Line_30	
00028290-000282B9	event_pkg.event_dispatcher.Line_32	
000282BA-000282E3	event_pkg.event_dispatcher.Line_33	
000282E4-0002830B	event_pkg.event_dispatcher.Line_34	
0002830C-00028337	event_pkg.event_dispatcher.Line_35	
000286E4-000286EF	control_pkg.valid_xing_req.Line_74	
000286F0-000286F3	control_pkg.valid_xing_req.Line_76	
000286F4-000286FD	control_pkg.valid_xing_req.Line_88	
000286FE-00028709	control_pkg.valid_xing_req.Line_89	
0002870A-00028713	control_pkg.valid_xing_req.Line_90	
00028714-0002871D	control_pkg.valid_xing_req.Line_92	
0002871E-00028727	control_pkg.valid_xing_req.Line_93	
00028728-00028731	control_pkg.valid_xing_req.Line_94	

Path Tracing & Performance Analysis



PathMap Benefits

- Behavioral Structure
- Design Verification
- Modification Assessment

TaskView

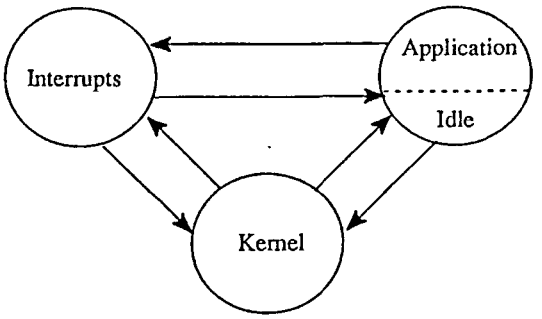
Goal: to provide system view of multi–tasking execution

Observe state transitions between:

- Application tasks
- Interrupts
- Kernel

TaskView

Run–time State Transition Model



- Record transitions – interrupts, kernel, application, idle task
- Durations in each state
- System calls to kernel
- Individual interrupt routines
- Procedures within application, nesting level
- Library calls
- A prerequisite for lower level analysis

TaskView Mechanics

- Use instrumented kernel to output transition points
  - In–line writes to a set of variables
  - Each variable represents specific state transition
  - Low overhead
- Use hardware to capture data and timestamp
- Hardware provides virtual trace to a disk file

TaskView Mechanics

Instrumentation solves many inherent trace problems:

- Prefetch – getting worse with burst modes
- Cache on – instr. overhead lower than with cache off
- Superscalar execution
- Logical to physical addressing
- Provides access to dynamic data
  - Tasks IDs
  - Heap information
  - Passed parameters
  - Messages
  - Data tagging

TaskView Release 1

Instrumented Ada kernel

Why Ada?

- Ada tasking model "restrictive"
  - Prioritized tasks queued in FIFO order rather than by priority
  - No cyclic executive
  - Tasks in Ada run non–deterministically
  - Task priorities cannot be changed at runtime
- Military/Aerospace projects have tough quality requirements
  - Designing many hard real–time systems
  - CPU utilization factor <50%
  - Customers have limited visibility into vendor run–time and its fit to their problem at hand

TaskView Release 1

- Sha/Goodenough of SEI/CMU
  - Developed rate monotonic scheduling theory; *Computer,* April 1990
  - Requires period and duration of each task
  - Blocking times caused by task inversion

TaskView Trace Display

Location in trace

System state

Current task name

Duration in each state

Symbolic name lookup

- Task names; optional IDs
- System calls by name
- Interrupt names
- Library calls

Additional trace data

- Ada exception source and handler found

  Memory allocation caller, size, pointer to mem

```
1  with unchecked_deallocation;
2  with text_io;
3  procedure example is
4
5    type buffer_ptr is access string;
6    subtype dist_range is integer range 0..12;
7
8    buffer : buffer_ptr;
9    size   : constant integer := (dist_range'last * 3);
10   middle : constant integer := (size / 2);
11   dist   : dist_range := 0;
12   add    : boolean;
13
14   procedure reset_dist;
15
16   task consumer is
17     entry deallocate;
18   end consumer;
19
20   task producer is
21     entry allocate;
22   end producer;
23
24   procedure reset_dist is
25     increment : exception;
26     decrement : exception;
27   begin
28     if (dist = dist_range'first) then
29       add := true;
30     elsif (dist = dist_range'last) then
31       add := false;
32     end if;
33
34     if (add) then
35       raise increment;
36     else
37       raise decrement;
38     end if;
39
40     exception
41       when increment => dist := dist + 1;
42       when decrement => dist := dist - 1;
43   end;
44
45   task body producer is
46   begin
47     loop
48       accept allocate do
49         buffer := new string'(1..size => ' ');
50         buffer(middle-dist..middle+dist) := (others => '*');
51       end allocate;
52       consumer.deallocate;
53     end loop;
54
55   end producer;
56
57   task body consumer is
58     procedure free is new unchecked_deallocation(string,buffer_ptr);
59     add : boolean;
60   begin
61     loop
62       accept deallocate do
63         text_io.put_line(buffer.all);
64         free(buffer);
65       end deallocate;
66       reset_dist;
67       producer.allocate;
68     end loop;
69   end consumer;
70
71 begin
72
73   text_io.put_line("Example program starts.");
74   producer.allocate;
75
76 end example;
```

#	S	TASK	DUR (uS)	EVENT NAME
9	ker		137.4	TS_FINISH_ACCEPT
13	app	consumer	8.0	
14	app	consumer	309.6	_A_reset_dist_14B13_ entry 1
15	app	consumer	24.4	_A_reset_dist_14B13_ exit 1
16	ker		259.6	TS_CALL
20	app	producer	19.6	
21	ker		128.8	TS_ACCEPT
25	app	producer	196.8	
26	int		64.6	Interrupt_0234
27	app	producer	318.4	
28	ker		139.0	TS_FINISH_ACCEPT
32	app	producer	21.8	
33	ker		257.0	TS_CALL
37	app	consumer	19.2	
38	ker		127.8	TS_ACCEPT
42	app	consumer	7.4	
43	app	consumer	19,190.0	put_line entry 1
44	int		64.4	Interrupt_0234
45	app	consumer	38,458.0	
46	int		63.8	Interrupt_0234
47	app	consumer	59,789.0	
48	int		63.6	Interrupt_0234
49	app	consumer	41,824.0	
50	int		63.6	Interrupt_0234
51	app	consumer	48,026.0	
52	int		63.4	Interrupt_0234
53	app	consumer	29,946.0	
54	int		63.8	Interrupt_0234
55	app	consumer	87.6	
56	app	consumer	123.2	put_line exit 1
57	ker		139.2	TS_FINISH_ACCEPT
61	app	consumer	7.8	
62	app	consumer	310.4	_A_reset_dist_14B13_ entry 1
63	app	consumer	24.6	_A_reset_dist_14B13_ exit 1
64	ker		260.6	TS_CALL
68	app	producer	19.8	
69	ker		129.6	TS_ACCEPT
73	app	producer	509.6	
74	int		64.2	Interrupt_0234
75	app	producer	19.4	
76	ker		138.4	TS_FINISH_ACCEPT
80	app	producer	21.6	
81	ker		257.6	TS_CALL
85	app	consumer	19.0	
86	ker		128.8	TS_ACCEPT
90	app	consumer	7.2	
91	app	consumer	27,597.0	put_line entry 1
92	int		64.4	Interrupt_0234
93	app	consumer	30,016.0	
94	int		63.4	Interrupt_0234
95	app	consumer	59,802.0	
96	int		63.8	Interrupt_0234
97	app	consumer	41,882.0	
98	int		64.2	Interrupt_0234
99	app	consumer	47,955.0	
100	int		64.0	Interrupt_0234
101	app	consumer	29,920.0	
102	int		64.0	Interrupt_0234

#	S	TASK	DUR (uS)	EVENT NAME
29	ker		260.2	TS_CALL
33	app	producer	19.4	
34	ker		128.6	TS_ACCEPT
38	app	producer	24.6	
42	lib	producer	75.8	alloc_entry pc=02011df6 size=00000034
44	lib	producer	37.8	semaphore_entry id=02014250
46	lib	producer	17.2	semaphore_exit id=02014250
48	app	producer	331.6	alloc_exit ads=020170cc
49	ker		139.4	TS_FINISH_ACCEPT
53	app	producer	21.2	
54	ker		257.8	TS_CALL
58	app	consumer	19.0	
59	ker		128.0	TS_ACCEPT
63	app	consumer	7.4	
64	app	consumer	19,149.0	put_line entry 1
65	int		64.0	Interrupt_0234
66	app	consumer	38,579.0	
67	int		64.6	Interrupt_0234
68	app	consumer	59,776.0	
69	int		64.0	Interrupt_0234
70	app	consumer	41,926.0	
71	int		64.4	Interrupt_0234
72	app	consumer	47,923.0	
73	int		63.8	Interrupt_0234
74	app	consumer	30,125.0	
75	int		63.6	Interrupt_0234
76	app	consumer	87.4	
77	app	consumer	18.4	put_line exit 1
79	lib	consumer	30.0	dealloc_entry ads=020170cc
81	lib	consumer	30.0	semaphore_entry id=02014250
83	lib	consumer	16.0	semaphore_exit id=02014250
84	app	consumer	21.6	dealloc_exit
85	ker		138.2	TS_FINISH_ACCEPT
89	app	consumer	7.6	
90	app	consumer	57.4	_A_reset_dist_14B13_ entry 1
94	lib	consumer	214.6	exception pc=02011d8e id=02013b88
96	app	consumer	26.2	exception_handler pc=02011d2e
97	app	consumer	24.8	_A_reset_dist_14B13_ exit 1
98	ker		259.4	TS_CALL
102	app	producer	19.2	
103	ker		129.4	TS_ACCEPT
107	app	producer	24.6	
111	lib	producer	76.4	alloc_entry pc=02011df6 size=00000034
113	lib	producer	38.4	semaphore_entry id=02014250
115	lib	producer	17.2	semaphore_exit id=02014250
117	app	producer	316.6	alloc_exit

TaskView Performance Summary

State level performance as a percentage of total time

- Application, kernel, interrupts
- Idle (CPU utilization)
- Instrumentation overhead

Task–level performance view; running, non running

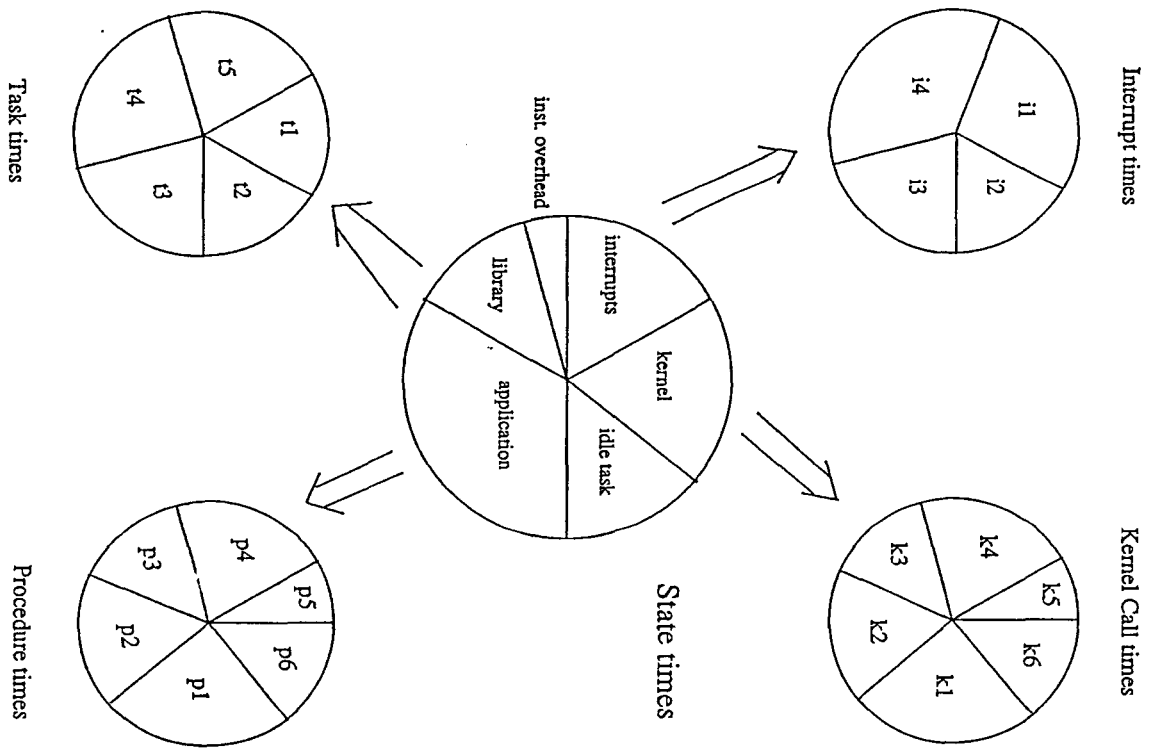
Interrupt durations: Min, Max, Avg

Kernel call durations: Min, Max, Avg

Libraries: memory alloc/dealloc, semaphore

Procedures within applications

User definable events



PERFORMANCE SUMMARY: (16,1000)

unknown	(unk)	time -	0.0%	0 uS
idle	(idl)	time -	0.0%	0 uS
interrupt	(int)	time -	0.2%	6,164 uS
kernel	(ker)	time -	0.4%	14,806 uS
library	(lib)	time -	0.2%	5,883 uS
application	(app)	time -	99.2%	3,300 mS
instrumentation		time -	<0.1%	2,583 uS
		TOTAL TIME -	100.0%	3,328 mS

RUNNING TASK TIME	%	OCCs	ACC	AVG	MIN	MAX
consumer	99.8%	15	3,299 mS	219,964 uS	331 uS	238,038 uS
producer	0.2%	14	6,692 uS	478 uS	404 uS	527 uS

NOT-RUNNING TASK TIME	%	OCCs	ACC	AVG	MIN	MAX
consumer	0.2%	14	6,770 uS	483 uS	431 uS	527 uS
producer	99.8%	13	3,091 mS	237,842 uS	237,540 uS	238,038 uS

PROCEDURE NAME	%	OCCs	ACC	AVG	MIN	MAX
_A_reset_dist_14B13_	<0.1%	14	1,176 uS	84 uS	81 uS	85 uS
put_line	100.0%	14	3,293 mS	235,220 uS	207,310 uS	237,565 uS

INTERRUPT NAME	%	OCCs	ACC	AVG	MIN	MAX
Interrupt_0234	100.0%	98	6,164 uS	62 uS	0 uS	65 uS

LIBRARY NAME	%	OCCs	ACC	AVG	MIN	MAX
exception	51.6%	14	3,035 uS	216 uS	214 uS	242 uS
alloc_entry	22.7%	14	1,333 uS	95 uS	92 uS	118 uS
semaphore_entry	15.8%	27	926 uS	34 uS	29 uS	39 uS
dealloc_entry	10.0%	13	588 uS	45 uS	44 uS	46 uS

KERNEL NAME	%	OCCs	ACC	AVG	MIN	MAX
TS_FINISH_ACCEPT	26.4%	28	3,908 uS	139 uS	137 uS	163 uS
TS_CALL	49.2%	28	7,290 uS	260 uS	256 uS	286 uS
TS_ACCEPT	24.4%	28	3,606 uS	128 uS	127 uS	130 uS

OTHERS NAME	%	OCCs	ACC	AVG	MIN	MAX
No entries						

PERFORMANCE SUMMARY:

idle	(idl)	time -	38.7%	15,328 mS
interrupt	(int)	time -	0.2%	67,729 uS
kernel	(ker)	time -	0.3%	132,729 uS
library	(lib)	time -	0.0%	0 uS
application	(app)	time -	60.7%	24,033 mS
instrumentation		time -	0.0%	10,800 uS
		TOTAL TIME -	100.0%	39,566 mS

TASK RUNNING TIME	%	OCCs	ACC	AVG	MIN	MAX
main	11.1%	20	2,666 mS	133,348 uS	6 uS	1,446 mS
rand_delay	0.0%	21	197 uS	9 uS	7 uS	18 uS
philosopher	0.0%	365	5,021 uS	13 uS	6 uS	34 uS
fork	0.0%	134	1,052 uS	7 uS	6 uS	18 uS
dining_room	0.0%	145	2,495 uS	17 uS	8 uS	21 uS
output	88.9%	425	21,358 mS	50,254 uS	1 uS	121,060 uS

TASK NOT RUNNING TIME	%	OCCs	ACC	AVG	MIN	MAX
main	88.9%	1968	21,366 mS	10,857 uS	0 uS	18,920 mS
rand_delay	100.0%	1958	24,033 mS	12,274 uS	0 uS	13,383 mS
philosopher	100.0%	1611	24,028 mS	14,915 uS	0 uS	4,142 mS
fork	100.0%	1827	24,032 mS	13,154 uS	0 uS	4,623 mS
dining_room	100.0%	1801	24,031 mS	13,343 uS	0 uS	4,142 mS
output	11.1%	640	2,675 mS	4,180 uS	0 uS	1,696 mS

PROCEDURE NAME	%	OCCs	ACC	AVG	MIN	MAX
_A_put_80B13_output	100.0%	344	3,887 mS	11,301 uS	5,218 uS	47,840 uS

INTERRUPT NAME	%	OCCs	ACC	AVG	MIN	MAX
interrupt_2	100.0%	2318	67,724 uS	29 uS	26 uS	50 uS

LIBRARY NAME	%	OCCs	ACC	AVG	MIN	MAX
No entries						

KERNEL CALL NAME	%	OCCs	ACC	AVG	MIN	MAX
AA_GLOBAL_NEW	0.2%	6	272 uS	45 uS	45 uS	45 uS
TS_INIT_ACTIVATE_LIST	0.1%	1	60 uS	60 uS	60 uS	60 uS
TS_CREATE_MASTER	0.1%	1	75 uS	75 uS	75 uS	75 uS
TS_CREATE_TASK_AND_LINK	3.4%	13	3,771 uS	290 uS	289 uS	290 uS
TS_ACTIVATE_LIST	0.8%	1	929 uS	929 uS	929 uS	929 uS
TS_ACTIVATION_EXCEPTION	0.1%	1	61 uS	61 uS	61 uS	61 uS
TS_TID	0.2%	13	276 uS	21 uS	20 uS	21 uS
TS_ACTIVATION_COMPLETE	0.5%	13	541 uS	41 uS	40 uS	53 uS
TS_ACCEPT	0.5%	8	531 uS	66 uS	52 uS	121 uS
TS_CALL	29.9%	268	33,536 uS	125 uS	61 uS	218 uS
TS_FINISH_ACCEPT	12.1%	183	13,613 uS	74 uS	58 uS	119 uS
TS_SELECT	16.6%	178	18,601 uS	104 uS	21 uS	108 uS
TS_COMPLETE_MASTER	0.1%	1	121 uS	121 uS	121 uS	121 uS
TS_FAST_ACCEPT	8.5%	119	9,585 uS	80 uS	75 uS	172 uS
TS_DELAY	24.3%	88	27,216 uS	309 uS	58 uS	381 uS
TS_TIMED_CALL	2.6%	30	2,953 uS	98 uS	87 uS	295 uS

Paper 2-M-1

SOFTWARE QUALITY ENGINEERING

Mr. J. Cashman
J&J Consulting

Mr. John T. Cashman is the Manager of Software Quality Engineering, responsible for all quality-related activites. Prior to this assignment, he was a Senior Staff Systems Engineer, performing a wide variety of engineering studies relating to various C3 applications. Mr. Cashman has over twenty years of experience in the definition, development, scheduling, and implementation of technical and management requirements for software and systems engineering programs. He holds and MS in Business Administration from Golden Gate University and a BSBA from the University of Southern Colorado.

SOFTWARE QUALITY WEEK

SAN FRANCISCO CA

15 - 18 MAY 1990

SOFTWARE QUALITY ENGINEERING (SQE)

A PRESENTATION BY

JOHN T. CASHMAN

SOFTWARE QUALITY ENGINEER

J & J QUALITY SERVICES

SOFTWARE QUALITY ENGINEERING

BACKGROUND

In general our industry is not paying attention to software quality, as evidenced by the following:

> Ashton-Tate, as quoted in *The Wall Street Journal*, dated October, 1988, stated that dBASE IV would meet the "quality standard". That same publication reported in its April 11, 1990, issue that the program still contained thousands of bugs and has not been not shipped
>
> According to *Time Magazine*, January 29, 1990, the AT & T Long distance telephone system failed due to a computer software logic error.

From the foregoing it would appear that there really isn't a great deal of interest in the software industry policing itself when it comes to quality.

It should be noted however, that the Federal Government in its role as the largest purchaser of software in the country has noticed the problem and is concerned. This concern is well documented in a Staff Study conducted by the Subcommittee on Investigations and Oversight, dated September 1989, transmitted to the Committee on Science, Space, and Technology, U.S. House of Representatives, One Hundred First Congress. The Introduction of this Study entitled, *Bugs in the Program*, in a discussion of software problems contains the following statement: "These concerns are heightened as computers perform more critical tasks where mistakes can cause financial turmoil, accidents or in the extreme cases, death".

The United States Air Force has been in the forefront of activities to improve software quality. Their activities include the establishment of the Software Engineering Institute at Carnegie Melon University, and active participation in the development of new DOD Standards for software design, implementation, and quality (MIL-STD-2167A and MIL-STD-2168). The concept of Software Quality Engineering, the subject of this paper, was enhanced as a result of a study conducted jointly by the Rome Air Development Center and Boeing Corporation. The message for the software industry would appear to be: "Either begin to police your quality or have the Federal Government do for you."

This paper is based on the RADC/Boeing study and the book, *Software Quality Engineering*, by Michael S. Deutsch and Ronald R. Willis, Prentis Hall and is presented as one means of addressing the software quality issue.

INTRODUCTION

Total quality is the result of three distinct processes: SQE, which engineers-in quality, testing to detect errors, and IV&V reviews to detect defects. The emphasis in all SQE activities is the specification, design, and implementation of good quality, as opposed to finding defects. SQE is an engineering process that is active throughout the software life cycle. That quality cannot be achieved until it is specified is the major premise underlying the SQE concept.

It is not enough to simply state that the software product will be a quality product for the following reasons: Achieving each succeeding level of quality incurs added cost. The same level of quality is not required for each functionality. Some Quality Factors are mutually exclusive, e.g., efficiency of processing may compete with efficient use of storage resources

Frequently software quality and the collection of software metrics are lumped together as a single activity. While SQE and the use of software metrics have much in common there are some subtle, albeit significant, differences. While both systems use metrics, many of which are the same, they are used differently. In a purely metrics driven approach the metrics are used as measures of such things as programmer productivity and other process related attributes. This tends to be an after-the-fact approach. SQE, on the other hand, takes a much more proactive approach. First it takes a set of metrics and separates them into two classes. One class, Quality Factors, represent the users "fitness for use" interests. The second group of metrics, Quality Criteria, is engineering oriented. The latter are directed toward implementing the user's Quality Factors by converting them to engineering attributes that can be specified, implemented, and monitored in the software during the design and coding process. Over time these activities can be used to measure characteristics like the productivity metrics do, however, the immediate objective of SQE is to initiate the improvement of software product quality early in the design phase.

In quality terms the software product consists of a design, computer executable code, and supporting documentation. The product quality is a matter of perception depending on a particular viewpoint. The procuring agent views quality in terms of how well the software meets the specifications. The software designer views quality in terms of freedom from defects and errors. The user has operational needs in terms of functionality and performance. The user also has maintenance needs in terms of changes and the management of change. Change, which in itself is a very high risk activity, addresses modifying the software to correct errors, to add functionality, or to adapt it to new environments. Management needs deal with planning and implementing change, controlling versions of the software, testing, and installation. SQE addresses all of these needs in whatever combination they exist, e.g a user who does not maintain software doesn't require the same quality level as one who does. It is important to note that quality is not the same for all applications. Because of this it is necessary to establish beforehand what quality level is required for a particular application.

SQE recognizes that there is a difference between the quality of the software products and the quality of the software development process. Product quality describes attributes of the products resulting from the software development process. Examples of product quality include clarity of the design document, traceability of the design to the specified requirements, reliability of the code, and the completeness of tests. Process quality focuses on the software engineering environmental elements such as: techniques, tools, people, organization, and facilities. Achieving a level of quality is a matter of first specifying the software quality attributes that are desired and then selecting the tools and techniques that can produce these attributes.

QUALITY SPECIFICATION

The fifteen Quality Factors described in Table 1 are the cornerstones of SQE. These factors define the boundaries of what may be required in any software product. All of these factors are not required in all software products. Consequently there must be an agreed-upon definition of which Quality Factors are necessary and desirable from the outset. This agreement ultimately becomes the quality specification.

Table 1
SOFTWARE QUALITY FACTORS
(User Viewpoint)

Correctness	What was produced is what was specified, and vice versa
Efficiency	The number of resources required is affordable
Expandability	It is easy to change the software to add new capabilities
Flexibility	The software is adaptable to a wide range of different environments
Integrity	The user is reasonably sure that the software and data are not being tampered with or stolen
Interoperability	The software produces or uses results that comply with industry standards
Maintainability	The software is productive during the maintenance life cycle
Manageability	The support environment is complete and easy to use
Portability	The software may be used on a wide range of operating systems and computers
Reliability	There is an acceptably long time between failures
Reusability	There is a large library of software building blocks
Safety	The software can be trusted
Survivability	Essential functions are still available even though some parts of the system are down
Usability	It is easier to use the software than not to use it
Verifiability	The software is productive during certification

Source: Rome Air development Study RADC-TR-85-37

QUALITY CRITERIA

Good quality, meaning "fitness for use", as defined by the user, is not necessarily understood, by the engineers who are required to produce the software. The process by which engineers can ensure quality is through the use of engineering based Quality Criteria; for example implementing the Anomaly Management criteria would involve designing the software in such a manner that all input data would be subject to validity and range checks. When the data failed these tests the system would deal with the error in a manner that rejected the data, notified the operator and continued to operate without the invalid data. Each of the 27 Quality Criteria listed in Table 2 is defined in terms of characteristics that the software will exhibit when it has been designed and developed to meet that criteria.

Table 2
SOFTWARE QUALITY CRITERIA
(Engineering Viewpoint)

CRITERIA	CONSIDERATIONS
Accuracy	Achieving required a precision in calculations and outputs
Anomaly Management	Nondisruptive failure recovery
Augmentability	Ease of expansion in functionality and data
Autonomy	Degree of decoupling from execution environment
Commonalty	Use of standards to achieve interoperability
Completeness	All software is necessary and sufficient
Consistency	Use of standards to achieve uniformity
Distributivity	Geographical separation of functions and data
Document Quality	Access to complete understandable information
Efficiency of Communication	Economic use of communications resources
Efficiency of Processing	Economic use of processing resources
Efficiency of Storage	Economic use of storage resources
Functional Scope	Range of applicability of a function
Generality	Range of applicability of a unit
Independence	Degree of decoupling from support environment
Modularity	Orderliness of design and implementation
Operability	Ease of operating the software
Safety Management	Software design to avoid hazards

Table 2
SOFTWARE QUALITY CRITERIA (Continued)
(Engineering Viewpoint)

CRITERIA	CONSIDERATIONS
Self-Descriptiveness	Understandability of design and source code
Simplicity	Straightforward implementation of functions
Support	Functionality supporting the management of changes
System Accessibility	Controlled access to software and data
System Clarity	Ensures the clear description of program structure
System Compatibility	Ability of two or more systems to work in harmony
Traceability	Ease of relating code to requirements and vice versa
Training	Provisions to learn how to use the software
Virtuality	Logical implementation to represent physical components
Visibility	Insight into validity and progress of development

Source: Rome Air development Study RADC-TR-85-37

Because the Quality Factors represent the user's viewpoint and the Quality Criteria represent the engineers viewpoint some method must be used to map these disparate viewpoints to a common understanding. The mapping provides the basis for specifying which of Quality Criteria should be engineered-in during the design and development process to meet selected quality requirements. What remains is to zero-in on an agreed-upon definition of "high quality" for each software product. In other words a means for specifying software quality. This should be completed as early in the development cycle as possible, in any case no later than the preliminary design review (PDR).

QUALITY NEEDS ANALYSIS

Quality needs are based on the statement of work, system requirements specifications, design specifications and any other source that provides clues identifying the end-users needs. Another source of information relating to quality needs is discussions with the current and prospective users. These discussions can do much to develop an understanding of the vague and legalistic terms used in contractual documents. Needs may also be determined through observation of the existing or similar systems for the purpose of identifying potential problems The problems observed in the existing systems may lead to additional knowledge and insights into the system being developed. As each need is identified it should be entered into the needs data base for subsequent review and analysis

Using the Quality Factors and Criteria described in Tables 1 and 2 it is possible to conduct a needs analysis to create a program specific quality requirements matrix. The Quality Criteria must then be examined to identify how each of the criteria selected will be implemented. A data base of the needs and selected criteria may be created to provide the basis for a quality specification. Converting the needs into quality requirements will result in a specification of the required software quality.

QUALITY LEVEL DETERMINATION

The objective of this process is to identify the optimum relationships between quality, cost, and schedule. As a result of this analysis all of the possible quality needs of the system will have been identified and described in a needs data base. This process examines these needs in terms of their affordability and practicality within the constraints of the overall project. The objective is to uniquely adapt the universal case to the specific needs of the software being developed. The relative importance of each factor to the software component should be determined.

As the design matures and the requirements are better understood it is anticipated that the needs levels will be adjusted as required. The level of needs should be reviewed throughout the design process at walkthroughs, design reviews etc. If there are current users of the system they should also be consulted in the needs level determination process.

It is likely that the results of the quality levels analysis will produce some conflicts in terms of user preferences, cost, and schedule. Initially the conflicts should be resolved through negotiation with all of the parties concerned. Although these

procedures are somewhat primitive they do provide considerable insight into the various tradeoffs that must be made, and they do so early in the development process.

CONVERT STATED NEEDS TO TESTABLE, OBJECTIVE, REQUIREMENTS

As noted earlier very often requirements will be described in vague terms that are meaningless to the software engineer, in other words they are not good requirements. For a requirement to be "good" means that it can be used to design software that possesses the required Quality Factor and that an independent observer can objectively verify that the implemented software exhibits the specified attribute.

There are no cookbook techniques for making each requirement objective and testable. There are quality attributes that may be impossible to state in an objective, testable manner e.g. Cohesion. Requirements exhibiting these characteristics should be stated in such a manner that acknowledges the fact and that there is a range of acceptable results. When requirements appear to exhibit these characteristics it is up to the SQE personnel to become inventive and test alternative means of stating the requirement or to attempt to perceive the requirement from a different aspect of the problem. In cases where invention is employed it is best to have someone else measure the objectivity of the solution.

ALLOCATING QUALITY REQUIREMENTS TO DESIGN COMPONENTS

The software design process results in several levels of design. Initially all of the software is partitioned into programs. Functional requirements may be partitioned and then synthesized into modules. The tasks are then partitioned into units. Allocation of quality requirements may be done at each level of design. The only criterion for making an allocation is the applicability of the requirement to the component in question. If it is applicable, it becomes a part of the requirements for that level of design. Requirements may be allocated to any level of design and allocation continues throughout the design process. The results of the allocation process may then be recorded in a manner which becomes a part of the design documentation. These documents provide the basis for checklists to be used during design reviews and code walkthroughs.

SUMMARY

Software quality improvement is a pressing need. Either the industry will improve it or the government will mandate it. This paper attempts to describe a rationale and engineering based approach to this goal.

Paper 2-M-2

STRUCTURE OF THIRD-PARTY SOFTWARE TESTING ORGANIZATIONS: PROGRESS ASSESSMENT

Dr. Vern Crandall
Brigham Young University

Prof. Vern Crandall is professor of Computer Science at Brigham Young University. He received his Ph.D. from University of Washington in Biomathematics. At BYU he developed a Software Engineering Curriculum in 1975 and later developed a series of Software Engineering Courses for IBM Technical Education. Crandall's courses at BYU reflect advanced teaching methodologies and require students to perform systems analyses, build software products, or test commercial software. In the 10 years that the testing course has been in operation his students have tested over 45 different products. He has just completed a two-year stay at Novell, Inc. of Provo, Utah, where he was Vice President of Software Development.

Structure of a Third Party Software Test Organization

An Assessment of Progress

Dr. Vern J. Crandall

Professor of Computer Science

BRIGHAM YOUNG UNIVERSITY

Provo, Utah

Quality Week

Wednesday

May 16, 1990

San Francisco, California

1:30 pm - 3:00 pm

Structure of Third Party Software Test Organizations
An Assessment of Progress

Dr. Vern J. Crandall
BRIGHAM YOUNG UNIVERSITY
Provo, Utah 84602
May 16, 1990

Third Party Testing, a concept which has been around for several years, appears to be an idea whose time has come. Organized correctly, it can be mutually beneficial for everyone involved. For many years, Brigham Young University has offered students the opportunity to learn Software Testing first-hand through actual test experiences with "real world" projects. These have often led to companies requesting students to continue testing their products after the course has ended--for pay. Recently, this concept was taken off-campus and a "for profit" testing institute, Softest, Inc., was created. This paper will address the nature of third party testing, the structure of the institute, its goals, and its directions. An assessment of progress will be presented as well.

1.0. Introduction.

For over 15 years, Brigham Young University has offered courses in the area of Software Engineering. These courses focus on Team Projects relating to "real world" projects with "real world" users. In 1980, as a part of that curriculum, a testing course, **Software Testing and Quality Assurance**, was started. The course was originally conceived as the testing of projects which were built in the earlier courses, but this proved impractical. It appeared most effective to test commercial software for companies that planned on distributing it nationally. When "off the shelf" software was tested, it tended to be too "clean." Students became discouraged when they could not find errors. Likewise, when companies offered "beta" or "post-beta" software, it also tended to be too clean. Over the years, companies began recognizing the BYU testing class as an excellent source of testing expertise (and potential test engineer new hires), and they provided software early enough that the testing proved effective. The major weakness was that most companies did not provide source code, so only **system testing** or **product verification testing** was possible. This meant that students needed two projects: one to handle commercial software at the **system** level and one to test source code at the **unit and integration** level. While this was a bit awkward and caused a lack of focus, it proved most effective. Since 1980, students in the BYU testing class have tested over 90 products for over 45 companies.

Sometimes companies recognized that student testing was extremely effective but that the semester end terminated their testing efforts. Also, it was difficult at times to get company software products to coincide with semester boundaries. This led companies such as IBM, IOMEGA, NOVELL, and MICROSOFT to propose that BYU set up what was referred to as **"On-Campus Cooperative Education"** projects, wherein the company contracted with students and faculty--through BYU--to continue testing software over several semesters. They would continually rotate projects through successive student teams. This proved very successful, and many of the students were hired by the companies providing these projects. This was discussed in a paper by the author [Crandall, 1989a], and more will be said about it in Section 2.

Two years ago, the author took a **professional development leave** to serve as *Vice President of Software Development* at **Novell**. Upon his return, he decided to resurrect the concept of **"On-Campus Cooperative Education"** projects, formalize them, and make them more effective, based upon his experiences at **Novell**. Part of the concept to be considered was the "certification" of software, according to pre-established criteria. Based on the political climate and changes in University policy at BYU, it was decided to take the project off-campus and create a "for-profit" corporation to manage acquired testing projects and provide the students to

work on the projects. This organization was originally to be called, the **University Institute for Software Testing and Research (UISTAR)**, to describe its function and direction. It was later decided to re-name it **Softest, Inc.**, which seemed more efficient as a corporate name.

This paper will discuss the nature of Third Party Testing, what makes it attractive to companies, and what its potential appears to be. It will also describe the background and learning models of the testing course at BYU and how and why it can be used to integrate students into the Third Party Test Organization, to be described in this presentation.

2.0. The Brigham Young University Testing Experience.

2.1. Background. In 1976, it was decided to prepare a course in Software Development at Brigham Young University. The course was to be based, initially, on the work of Stevens, Myers, and Constantine [Stevens, 1974] and later on Myers textbooks [Myers, 1976; Myers, 1978]. This course was later modernized to include the more up-to-date references in the field. There had already been a course in traditional systems analysis for several years which included a class project. To form a set of core courses, it was decided to modernize the systems analysis course and to teach a two-semester sequence of that course, followed by the software development course. The courses were to be "project-oriented," rather than "concepts-oriented," as is the case in the traditional university courses. Students were to perform a systems analysis in the first course and then build the product in the second. The projects were, for the most part, to be carried out for real users, not "professor-supplied, textbook-type" problems.

2.1.1. The Course Sequence Concept Rejected. Evaluation of the sequence concept quickly led to the modification of the courses to eliminate dependencies so that students could take the two courses in either order. The projects were made independent of each other. Systems analyses almost always led to "off-the-shelf" software, and users did not like to wait two semesters for their product in the few cases where it did not. Also, it was difficult to administer: keeping a team together for two semesters.

2.1.2. Student Teams. From the start, systems analyses were performed or software products were built using small student teams. Initially, 3-member teams were used for software development and 5-member teams for systems analyses. Later all teams came to consist of 4 - 6 persons, with 5 being the ideal. In addition to a team project, students were expected to create a personal project, to demonstrate their ability to work on their own and apply the same concepts they did in the team environment.

2.1.3. Real World Orientation. Because the courses were to prepare students for the real world, technologies which were in common usage in the real world were the ones taught and stressed in the courses. Experimental concepts and technologies were reserved for graduate courses. Later, the software design course was refined to reflect a development life cycle; project plan with a deliverables schedule; measures of software quality; software documentation with users manuals, screens and help files; software testing; installation considerations; and "user"-based beta testing.

2.1.4. Reference Libraries. It became obvious that there were many references that students needed access to in doing software development, so a library of appropriate reference books was established and maintained in the University's Reserve Library. A library of student projects was also maintained so that students could see the good ideas and examples which former student teams had created.

2.1.5. Team Presentations. Students were required to make an effective team presentation to the instructor and the rest of the class. This allowed all students to see what each other had done and gave students experience making formal technical presentations. [Suit and tie, heels and hose, were required and the presentation was to be made using overhead transparencies or slides. It was to be extremely formal rehearsed, and professional.] Students were evaluated on presentation skills as well as content, and every member of the team was to participate. They also were required to make a formal presentation to the user.

2.2. The Development of the BYU Testing Course. In 1980, it became obvious that there was a need for a testing course to round out the Software Engineering curriculum at BYU. The author attended two IEEE tutorials which had a major impact on the development of his course. These were Don Reifer's tutorial on Software Management [Reifer, 1979] and the Miller and Howden tutorial on Software Testing [Miller, 1977]. The textbook used was Myers, **The Art of Software Testing** [Myers, 1979]. Initially, the author was extremely naive about testing: What concepts should be taught? What could be tested? What should be tested? How should the test environment be organized? How should the students be graded? Most of the syllabus came from Miller and Howden and from Reifer, with Myers' text used as a supplement. The course was taught for the first time in 1981.

2.2.1. Benefits from Earlier Experience. It was obvious from the earlier experiences that this course should be project-oriented, not concepts-oriented. It should not be part of a sequence where one performed a systems analysis one semester, built a product the next, and tested it the next. A real live user could not wait that long for a product. Experience also showed that when professor-supplied problems were used, the students produced an inferior quality of work and did not learn as much. Also, students tended to be more "marketable" if the software they tested were being commercially marketed at the time of their job interviews and they had a copy of their testing report with them at their interview. For these reasons, an attempt was made to find real-world software to test instead of professor-supplied problems and projects.

2.2.2. Course Evolution. In 1983, Beizer's first book appeared [Beizer, 1983] and was immediately chosen as the text. The syllabus was expanded to include much of the Unit Testing material from Beizer.

The course continued to evolve and more and more companies were drawn to the testing program, allowing increased testing of "commercial-grade" software. A policy was established that code have national distribution to qualify for team testing. Student projects and faculty programs did not qualify. Students were required to sign non-disclosure statements before participating on specific projects.

Since most companies did not allow access to their source code, these commercial projects could not be "white box tested," which eliminated many, if not most, of the Unit Test Procedures. For this reason, small faculty or student projects were used to give students experience with Unit Testing.

The course syllabus was re-designed to reflect a System Test orientation in addition to the Unit Test orientation. At this time the main text became Beizer's second book [Beizer, 1984], which emphasized Systems Testing.

2.2.3. Reference Libraries. As other testing books have become available, student tastes and experience have led to a library of some 25 books [Crandall, 1989], which are made available to students. At present, the students' favorite reference is Hetzel [Hetzel, 1988], though there are some who prefer Perry [Perry, 1983; Perry, 1986]. Students also make extensive use of prior student projects.

2.2.4. Campus-Based Cooperative Education Projects. As was mentioned in the Introduction, one of the "enhancements" to this approach has come when companies have placed a project on campus outside the regular classroom experience and have paid students (and a faculty member) to continue testing either the product they tested for the class or other products. A major example of this was when IBM hired 21 students to test Release Three of the System/36 Office Product. The product size was around 780,000 lines of code and required three months to test. The students were divided into 4 teams of 5 persons, with one person designated as team leader. The 21st person was the project leader.

This approach to education has been extremely cost-effective. A company has essentially "hired" a group of high quality individuals to test their software, without incurring the overhead that would have been created had they been hired directly by the company. In addition, for a small fee, they have the faculty member's expertise and supervision provided to the project. It is this experience that has led to the Third Party Testing organization.

2.3. Professional Test Engineers. Although some students leave the testing course with little desire to do more than test their own code (if that), many others recognize the potential of becoming **Professional Test Engineers** (as opposed to testers or test technicians). Progressive companies such as Novell and Microsoft hire as many of these students as they can bring into their organizations each year. Many other companies, such as Borland, recruit specifically for hire test engineers each year at BYU.

There needs to be both a career definition and a career path for Professional Test Engineers. In the Tutorial on **Testing in a LAN Environment** [Newman, 1990], it was stressed that effective test engineers must be outstanding programmers--in C and Assembly, at least--having training and experience in working with operating systems and the various layers of the OSI model for Local Area Networks. Test organizations which pay their testing people less, call them "testers" or "Quality Assurance" people, and have them only work with user interfaces, are missing a great opportunity to build a strong, professional organization.

The testing course at BYU has as its goal to train Professional Test Engineers. Any one of these graduates could go into a company and start a test organization--should one not exist (one did exactly that at Novell several years ago). They are capable of starting--on day one--to create formal test plans and test strategies and to manage testing projects (one did at AT&T). If they are hired as programmers, they write their program specification, their test specification, software, and test cases and then Unit and Integration test their program (one did at Boeing).

3.0. The Concept of Third Party Testing.

Third Party Testing occurs when a company contracts with an outside company or individual to perform testing for that company. This may happen when the company does not have a test organization and needs a product tested or when it desires additional testing. It may also occur when outside "certification" is needed to provide a company's product with more "credibility."

3.1. Background. In the past, there have been organizations which have provided Third Party Testing. Some organizations, such as **Software Research, Inc.** and **National Testing Labs** have been in existence for some time and provide consulting and testing services in one form or another. There are some 15 or 20 other companies and organizations of which the author is aware, as well.

3.1.1. Certification of Compatibility. Third party testing is performed in the Novell environment, when Novell tests and certifies various drivers and cards and various types of hardware devices as being "Netware compatible." This allows companies to advertise with the Novell logo in their advertisements. Novell would run certification suites against a product to ascertain if it worked properly in the Novell environment.

A Third Party Test Organization might run software in different environments to certify interoperability, i.e., WORD PERFECT runs on Netware, WORD PERFECT and LOTUS 1-2-3 operate and communicate together, etc.

3.1.2. Certification of Completeness. Companies also may desire to show that an implementation of software running on their hardware is a complete implementation of a given product. Third party testing can be used in this context as well. Through CERTS and RAISE, two internal products used in the Novell test environment, Novell--and outside companies such as NCR, creating "Netware-like products"--can certify that their products are complete implementations of Netware [Newman, 1990].

3.1.3. Certification of Conformance to Standards. Similar types of certification could conceivably be performed against various GUI's (Graphical User Interfaces). Examples might be IBM's CUA (Common User Interface), Microsoft's Windows or Presentation Manager, Apple's Macintosh interface, or the NexT interface.

There is also a need for certification of conformance to certain technological standards. Examples might be data communications standards, data base standards, compiler standards, and other standards which are established by international bodies and by the US National Institute of Standards and Technology.

This certification could be done using the concepts, strategies, and automated tools in use at Novell [described in Newman, 1990] or using high-level language-based test suites, created over a period of time, to certify software against the published interface [Crandall, 1990a]. To make testing software in this environment effective, good software development methodologies should be employed relating to building test awareness and ease of testing into software products. A discussion of concepts in this environment is given in [Crandall, 1990b]. Ideally, a company wishing their software to be certified would submit that software to the Third Party Test Organization, which would then run its pre-existing test suites against the product and "certify" compliance. This compliance could be against any published standard--from government or industry.

3.1.4. Basic Software Testing. Third Party Test Organizations can also be used for standard testing in order to certify that the products are "error free." This is a much more dangerous, difficult type of testing. Yet this appears to be approach currently being taken in Italy. At last year's Quality Week, Ms Antonia Bertolino described the Software Certification program she was involved with, which has been in development for about 4 or 5 years. [Bertolino, 1989]. They are attempting to implement this program for all of Europe. It will be interesting to follow their progress.

3.2. Reasons to Use Third Party Testing. Is there a place for Third Party Testing? Is there a market for this kind of service? The answer to the first question is emphatically, YES. The answer to the second is more problematic. Many leading software developers do not have test organizations (no names will be mentioned); others have very poor test organizations. In several companies and organizations with which the author is familiar, the test organization is effective, but it is a "second class organization." This is normally the case where there is little or no management support and low salaries, as compared to development personnel.

With the cost of major software errors escalating, companies should put more and more emphasis on creating and building an effective test organization. The author gives a description of how to do this elsewhere [Crandall, 1990b]. But many companies still have not caught the vision of the cost/benefits of an effective test organization. Just how many companies will recognize the need for Third Party Testing is still not known.

3.2.1. No Corporate Test Organization. When a company does not have a test organization, it often does not know where or how to begin. Because of the relative weakness of many of the test organizations in existence, this question should be addressed. The June, 1990 presentation on *Creating an Effective Test Organization* addresses this question [Crandall, 1990a], but it takes time to build an effective organization--plus a great deal of management support. During the time that a company takes to build an effective test organization, hiring a Third Party to perform critical testing is an effective option.

3.2.2. Lack of Testing Resources. Often a company has spent all its software development budget before software testing can begin (or it may not even have budgeted for software testing). The company is left with a product to test and no budget or resources. Often, the solution is to ship the product and let the users test it. This will become less and less of an option as law suits start occurring when some users object to carrying the costs of lost data, loss of software connections, lost time, and lost customers. In this environment, one of the least expensive ways to solve the problem is to hire a Third Party Test Organization--especially one built around student labor, such as will be discussed in Section 4.

3.2.3. Lack of Testing Personnel. Another situation occurs when the cyclical nature of software development makes it so that there are times when there are too many testing staff and people are idle, waiting for a product to arrive for testing; the other situation is when there is too much product to test, not enough people, and management will not authorize any additional personnel. This "boom or bust" cycling makes it difficult to manage the test environment. Third Party Testing allows a company to operate with a smaller staff and hire the outside test organization to handle the "boom" times.

3.2.4. Lack of Testing Expertise. Sometimes a test organization exists, but the individuals who make up the organization lack the expertise and training to do an effective job of testing. Here again, the Third Party Test Organization can make available test engineers who are qualified to support and train the company's testing people. They perform part of the testing, help the company people to come up to speed on test technology and automated tools.

3.2.5. Second Opinion. Sometimes a company finds itself wondering if it is effective in testing its products. Too many errors are getting by, yet they seem to be doing an "adequate" job of testing. Is there anything wrong? Is their testing really effective? A independent test organization can do an audit of testing strategies, re-running critical test cases and evaluating the test environment. In addition, it can perform complementary types of testing to evaluate the state of the product.

3.2.6. Certification. Sometimes a company desires to have its software "certified" as meeting certain criteria. A discussion of certification was provided in Section 3.1. Certification--as described there--involved three areas.

Certification of Compatibility

Certification of Completeness

Certification of Conformance to Standards

This is becoming more and more of a practice as interoperability and connectivity become bigger issues. If a Third Party Test Organization becomes "expert" in a certain area and creates large, effective automated test suites, it becomes a fairly simple and inexpensive practice to "certify" that a particular software package conforms to some standard.

3.2.7. Software and Test Quality. Quality is defined in many different ways--error rate being only one. Other definitions of quality are provided in [Crandall, 1989b; Crandall, 1990c; Newman, 1990]. One use of a Third Party Test Organization is to certify that a product is of a certain quality, dependent upon what measure or measures are being considered. Some of this certification could be by citing "number (or percentage) of violations" of a particular quality measure; another part of it might be by actually running standard tests against the product; or still another part might possibly be running coverage analyzers against company-run test cases to come up with the certification.

3.2.8. Product Test Certification. Another area in which Third Party Test Organizations can be used is when the government or some other customer requires that a product be tested by someone other than the developer. In this situation, the independent test organization either re-runs all the tests of the developer and certifies that they all run without error or runs its own set of test suites and test cases against the product and certifies its correctness.

The organization might certify that error rates are below a certain rate. It might further establish that published functionality is present in the piece of software and that it is performing adequately. In this context, the organization might perform documentation testing--testing the documentation against the product to certify that the users manual, help files, help screens, error messages, etc., are all correct and clear to a person using the system.

As was mentioned in Section 2.1.4., this type of testing is somewhat dangerous, as one never "finishes testing; one quits testing." What happens--and who is responsible--for errors which are found after the product has been shipped? Who is liable? What must one do not to be liable? Are there insurance underwriters who will share some of the liability? What is the law beginning to look like in this area of software development and software testing? While these issues apply to all of the above situations in Section 3.2., they are especially critical when testing is performed in this last area. Yet this is the area where Third Party Test organizations are most likely to be used effectively.

4.0. Creation and Evolution of a Third Party Test Organization.

As has been mentioned in earlier sections of this paper, the Third Party Test concept tended to come about when IBM requested Brigham Young University to continue testing one of their products--after the end of a semester of testing. Other companies requested the same thing. This began as "campus-based cooperative education," and students were given the opportunity to work, under faculty supervision, on real world testing projects, for which they were paid. Later, as the author mentioned in the Introduction, the university requested that such projects be taken off campus.

4.1. University Institute for Software Testing and Research (UISTAR). Initially, it was felt that this entity should be a non-profit organization, and steps were taken to create such an official entity. It was felt that a corporation which hired students to perform testing for corporate sponsors would be easier to run, and it would be easier to recruit companies to provide projects if it were non-profit. However, as things progressed, it appeared that there would be too many problems trying to run the Institute as a non-profit entity.

The name was chosen to indicate the organization's university ties and to stress the fact that research, as well as contract testing would be carried out. It was stressed that UISTAR would not take on liability for certifying that products worked. It was not certain if companies could use the test organization's name or LOGO in its advertising--there are pro's and con's to this. The certification, testing, or evaluation would be a service provided to the company sending the software and funding the work; the results would be available--under non-disclosure--back to that funding company.

It was further decided that the company would provide research into areas of software testing, which information would be provided to industry through seminars, consulting, papers given at conferences, and papers published. Some of the research would be performed by the organization and would be marketed; other research would be funded through a university and would be in the public domain. Privately funded research--not being a part of a thesis or dissertation--would remain confidential to the funding organization or publicized at their discretion.

The company was to be established "off-campus" but would use university faculty and students. Later on, as it became stable and profitable, it would hire certain full-time employees.

4.1.1. Company's Basic Operating Strategy. Basically, the company would operate with a very low level of profitability, keeping the costs low. The company would keep operating overhead down by waiting until a contract had been signed. Then it would lease an appropriate amount of office space and office furniture and put in phones. If the client did not wish to furnish the equipment on which to test, this equipment would be leased locally, as well. The client would be billed directly for all out-of-pocket expenses--including insurance, should any be required as part of the contract.

Once everything was in place, students would be recruited, hired, and trained. The first choice would be students who had completed the university testing course the semester (or sometime) before, then would come students currently taking the course, then would come Computer Science majors with strong software backgrounds, then would be Computer Engineering students with strong software backgrounds, then students in the Business School MIS program, then any other intelligent students who could be recruited.

An IBM testing facility locally paid new hires $12/hr and experienced students $15/hr. It was decided to hire students and pay them (part time, up to 20 hours/wk) $13/hr initially, and after 2 months and acceptable progress, $15/hr. A project leader would be recruited for larger projects and would be paid $1 more per hour to manage the students. It was felt that this would allow the most intelligent, well-trained, students to be attracted to the project--regardless of their interest in software testing.

Space and furniture would be leased only for the length of the project; then the leases would be terminated and

the company shut down until the next contract were received. This would provide low operating capital needs and the ability to handle the "boom or bust" syndrome described earlier. But an attempt would be made to make the company a permanent entity, with permanent employees to allow for continuity of the technology and the operation.

4.1.2. Name Change. As the company struggled through its first few months, having several contracts postponed 6 to 9 months, it was decided to choose a new name, which would not be so complex. The name which was chosen was **Softest, Inc.** This has appeared to be a better choice. The company is also in the process of being revitalized with some dynamic management.

4.2. **Structure and Description of the Test Organization.** The company structure, goals, and operating strategy were defined. This led to a description of what would be its mission, how it would (and would not) operate, and how it would be organized. It was important that the company, or institute, not alienate vendors or others whose support it needed to succeed.

4.2.1. Mission Statement. **The mission of the institute is to grow the software testing industry, thereby promoting higher quality software.** This mission would be achieved through

- training students in testing technology;
- training software professionals in testing technology;
- funding research in areas where the testing technology is not fully developed;
- providing independent and unbiased test technology and services to interested companies;
- and increasing industry awareness of software quality.

This was to be done in partnership with and in cooperation with companies providing commercial, for profit, testing tools and services to the computer industry. In addition, an Advisory Group, made up of leaders in the Testing Industry, would be organized to provide for coordination of activities, direction in research, and leadership in other issues in which these advisors would like to play a role.

4.2.2. Operating Guidelines. It was decided that **Softest, Inc.** be managed according to the following guidelines:

- The institute will provide **evaluations** (as a product) to companies purchasing the institute's services. The institute will use commercially available tools, the client's own tools, and additional non-commercial tools and automation procedures developed by the institute.
- The institute will buy the best available commercial technology to furnish its clients with a high quality product. Where commercial technology is not available, appropriate, or practical, the institute will encourage vendors affiliated with the institute to develop appropriate tools. If in the vendor's judgement a tool is not commercially viable, or if the vendor does not have the time or resources to build such a tool, the institute may elect to build its own.
- The institute is not in business to market testing tools or other software products. Internally developed tools may be made available to clients and to commercial vendors who provide tools and services to the institute.

- The institute will use the best available commercial and non-commercial technology to provide its clients with a high quality product. It will not be bound unreasonably to any particular vendor.

- The institute will create its own automated test suites, as appropriate, to increase its competitive ability to certify software in the manner described above.

- The institute will use the services of consultants and advisors to enhance the quality of its products and services.

- The institute will preserve the confidentiality of the clients for whom it performs services.

- The institute will fund university professors and students to perform necessary research to support and enhance the services it provides.

4.2.3. Funding. Funding for the institute would come through three sources:

- Companies contracting for institute services, who would receive for their funding:

  - Copies of institute public research and any privately funded research they sponsored.

  - A specified amount of testing of their products.

  - A specified amount of consulting by consultants provided by the institute.

- Matching funds ($.50 for each $1.00 contracted) from the Utah Centers of Excellence.

- SBIR grants. An attempt would be made to interest the individuals contracting with the SBIR organization in funding Third Party Testing Organizations to make it easier for their organizations to obtain independent testing and to provide a facility for certification of their contracted software.

4.2.4. Affiliate Vendors. Because of the need for automated tools in the testing effort, vendors of automated tools would be encouraged to become Affiliate Vendors. These vendors would either provide their tools at a substantial discount to the institute or free to the institute, for use in testing products. In return, the institute would use their tools, when appropriate, to automate the testing process. In other circumstances, they would provide them--full-cost--to the client, should the client be willing to purchase them, for use by the institute.

The institute would take care not to recommend any given test tool over any other, but would use the tools the faculty supervisor and students felt were most appropriate for testing a given product. But the institute would indicate that a particular tool was used in the testing effort. Should this prove politically unfair or inappropriate, to make things more fair to various affiliate vendors, a list of available tools, with their specifications and descriptions, could be made available to the client, who then could choose the appropriate tool(s) for the automation of the institute's testing effort.

But it is obvious that the competitive advantage of the institute lies in the already-automated test scripts which

are captured and maintained from project to project and used in the certification process for successive clients.

4.2.5. Staffing. Leadership of the institute would be through a Board of Directors, a Chief Executive Officer, and a President. The Board of Directors would act as advisors to the institute, but would be paid for their time. The CEO and President would both be paid for their time, as funds became available. As it became economically feasible--with projects coming in on a regular, stable basis--a project manager, executive assistant, and secretary would be hired as full time employees. Later on, full-time testing experts would also be hired to provide continuity and enhance the quality of the testing effort.

University faculty would be hired on a consultant basis to work for the institute; students would be hired as part-time employees to work a maximum of 20 hours per week for the life of the project or until they graduate. These students then would be available for hire by the client, should they desire.

4.3. Current Status. The institute is currently in process of acquiring contracts. A million-dollar contract from a major computer company was put on hold for 6 to 9 months because of unavoidable delays in the product; another smaller contract has been delayed 3 to 6 months for similar reasons. It is hoped that ultimately these will be the contracts which get the institute off the ground.

It was hoped that one or both of these contracts--or some other contract--would be in place by the time this paper was presented, but that is not the case. There are several interested organizations with whom the institute is in the process of obtaining contracts, but none is in place at present.

5.0. Conclusions.

The nicest part about building an organization in the way the institute has been organized, is that when there are no projects, the company does not sit around quietly going broke. It can be started up and stopped as projects become available and end. When similar projects were run in the past as part of the **On-Campus Cooperative Education**, there were times when 3 or 4 projects were in place simultaneously--and other times when there were none or perhaps one.

Third Party Testing needs to be sold to companies. In the Novell environment, third party certification is a way of life--accepted by all the players not only as being necessary, but also as being extremely valuable for all concerned. Currently government-mandated Third Party Testing appears to be among the most viable opportunities.

It will be interesting to see where Third Party Testing is next year at this time. The author believes that the institute will be well on its way at that time.

6.0. References.

[Crandall, 1989a] CRANDALL, Vern J., *The Development of an Industry-Education Relationship in the Test Environment: The Novell-Brigham Young University Experience*, **Sixth International Conference on Testing Computer Software**, May 22 - 24, 1989, Washington, D.C. [1989].

[Stevens, 1974] STEVENS, Wayne, CONSTANTINE, Larry, and MYERS, Glenford, *Structured Design*, **IBM Systems Journal**, Vol. 13, No. 2, 1974, pp. 115-139 [1974].

[Myers, 1976] MYERS, Glenford, **Reliable Software Through Composite Design**, Pertrocelli/Charter [1976].

[Myers, 1978] MYERS, Glenford, **Composite/Structured Design,** Van Nostrand/Reinhold [1978].

[Reifer, 1979] REIFER, Donald, **Software Management,** IEEE Computer Society Tutorial (EHO 146-1) [1979].

[Miller, 1977] MILLER, Edward, and HOWDEN, William, **Software Testing and Validation Techniques (Second Edition),** IEEE Computer Society Tutorial (EHO 180-0) [1981]. The first edition was 1977.

[Myers, 1979] MYERS, Glenford, **The Art of Software Testing,** John Wiley & Sons [1979].

[Beizer, 1983] BEIZER, Boris, **Software Testing Techniques,** Van Nostrand-Reinhold [1983]. A new edition is due out this Spring/Summer.

[Beizer, 1984] BEIZER, Boris, **Software System Testing and Quality Assurance,** Van Nostrand-Reinhold [1984].

[Hetzel, 1988] HETZEL, Bill, **The Complete Guide to Software Testing (Second Edition),** QED Information Sciences, Inc. [1988].

[Perry, 1983] PERRY, W. E., **A Structured Approach to Systems Testing,** Prentice-Hall [1983].

[Perry, 1986] PERRY, W. E., **How to Test Software Packages: A Step-by-Step Guide to Assuring They Do What You Want,** John Wiley & Sons [1986].

[Newman, 1990] NEWMAN, Jan, HACKING, Brian, and CRANDALL, Vern J., *Testing in a LAN Environment: A Summary of Practical Experience*, **Quality Week,** May 15 - 18, 1990, San Francisco, California [1990].

[Crandall, 1990a] CRANDALL, Vern J. and HAYES, Linda G., *Creating an Effective Test Organization*, **Seventh International Conference on Testing Computer Software,** June 20, 1990, San Francisco, California [1990].

[Crandall, 1990b] CRANDALL, Vern J., *Building Testable Software: Integrating Test Awareness into Software Development*, **Second International Software Testing Methods, Tools and Techniques Conference,** March 1, 1990, Orlando, Florida [1990].

[Bertolino, 1989] BERTOLINO, Antonia, *Software Certification: The Ongoing Experience of Italian Government Controlled Fiscal Meters*, **Quality Week,** May 16 - 19, 1989, San Francisco, California [1990].

[Crandall, 1989b] CRANDALL, Vern J., *Testing in a Local Area Network Environment*, **Quality Week,** May 16 - 19, 1989, San Francisco, California [1989].

[Crandall, 1990c] CRANDALL, Vern J., *Product Oriented Development Methods*, **Software Development '90,** February 6 - 9, 1990, Oakland, California [1990].

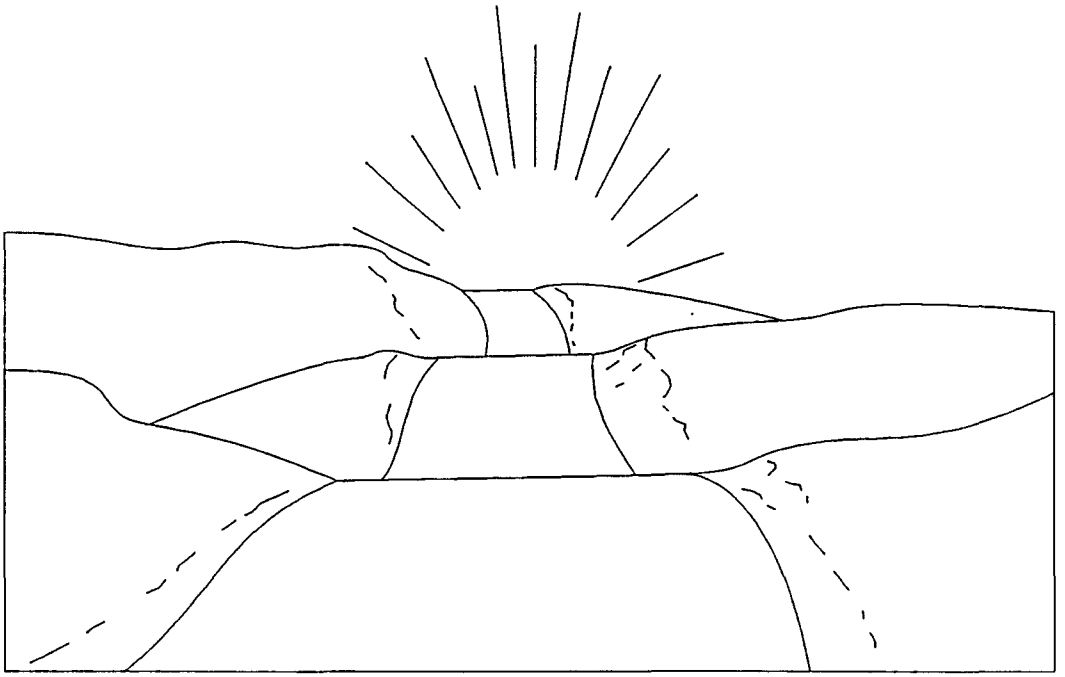
Paper 2-M-3

THE EMERGING SOFTWARE QUALITY PROFESSION: AN ASQC PERSPECTIVE

Mr. Taz Daughtrey
Chair, ASQC Software Division
B&W Nuclear Service Company

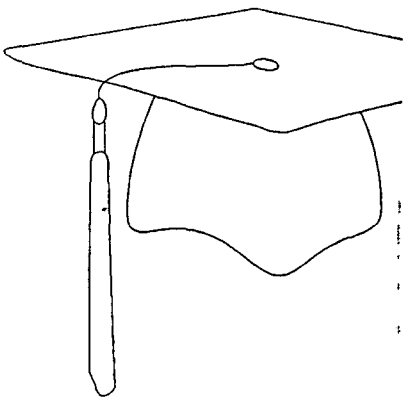
Mr. Taz Daughtrey chair of the ASQC Software Division, is a software quality specialist with the B&W Nuclear Service Company in Lynchburg, Virginia. He has been active in developing professional standards such as the IEEE Standard for Software Verification and Validation Plans, as well as several standards specific to the nuclear power industry. His work involves consulting, training, researching, and managing projects related to software quality measurement and procedures.

THE EMERGING SOFTWARE QUALITY PROFESSION



Taz Daughtrey Chair, Software Division
American Society for Quality Control

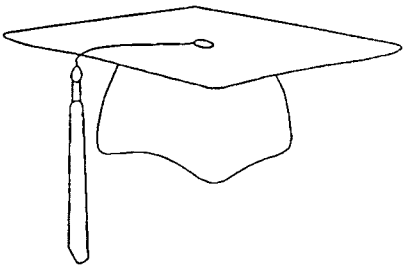
capture



widesp

control

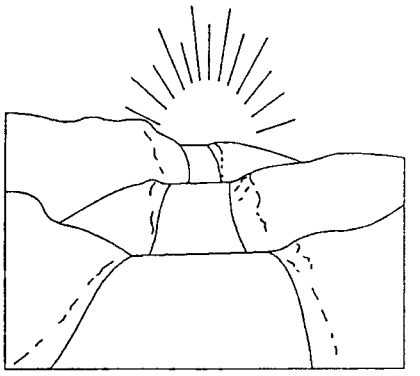
-- estab
-- requir
-- licens
-- code
-- accou

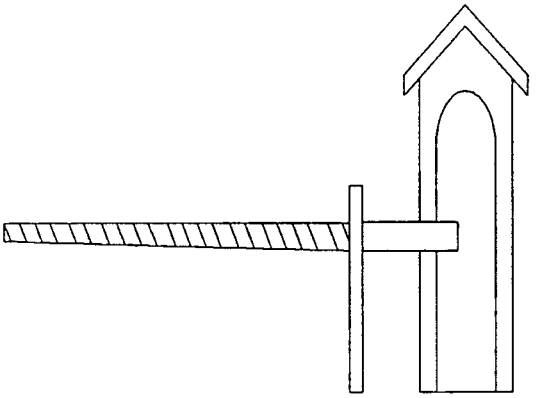


theory underlying practice

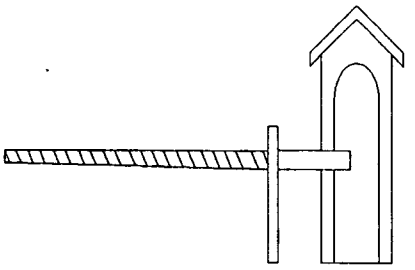
conceptualization

control by: accreditation





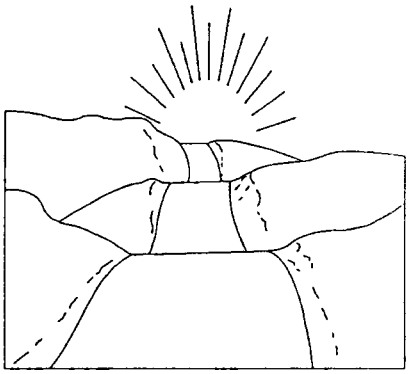
licensing

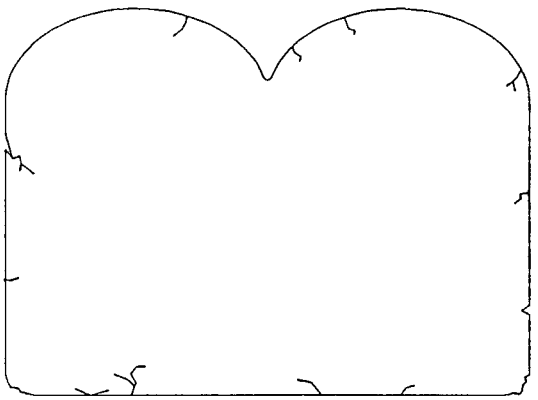


regulatory body

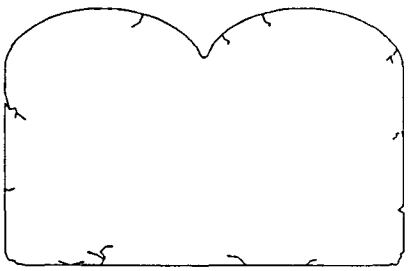
apprenticeship possible

control by: certification





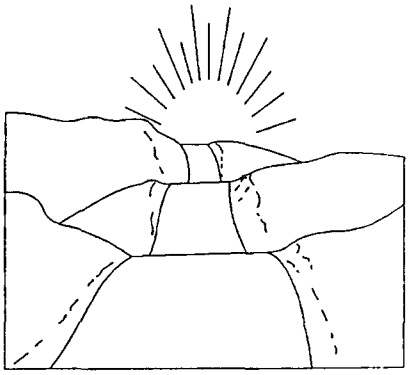
code of conduct

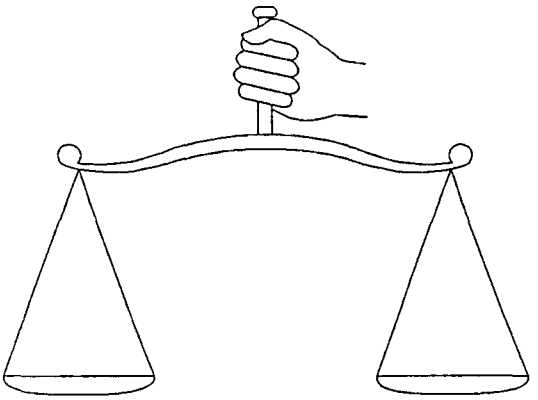


acceptable behavior

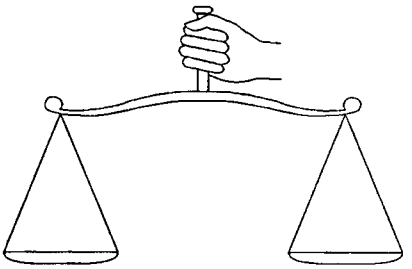
expectations, ethics

control by: discipline





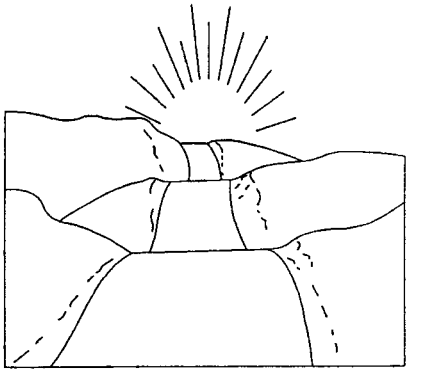
liability



expert opinion

malpractice

control by: adjudication



We will have a SOFTWARE QUALITY PROFESSION when we have ...

... common vocabulary and toolset.

... recognized certification.

... software malpractice suits.

Taz Daughtrey
B&W Nuclear Service Company
P.O.Box 10935
Lynchburg, Virginia 24506

phone: (804) 385-2869
fax: (804) 385-3663
COMPMAIL+: H.DAUGHTREY

Paper 2-M-4

EDP OUTSOURCING: QUALITY ASSURNACE AND RISK MANAGEMENT ISSUES

Mr. Richard F. Eng
MITRE Corporation

Mr. Richard F.T. Eng is a Group Leader in the Advanced Information Systems Division of the MITRE Corporation. He has provided Systems Engineering and Program Management support to various Federal agencies for nearly a decade. He is currently supporting the test and acceptance of a telecommunications service oversight facility for a government agency. Prior to joining the MITRE Corporation, Mr. Eng was an Engineer/Scientist at IBM working on the test and integration of an automated satellite command and control system. Mr. Eng holds a M.S. in Bioengineering and is a member of the IEEE and the Association for Computing Machinery.

Paper 2-A-1

USING CASE TECHNOLOGY IN A LARGE ORGANIZATION

Mr. James R. Hagedorn
Senior Vice President
Chartway Technologies

Mr. James R. Hagedorn is Vice President of the Technical Operations Division for Chartway Technologies, Mr. Hagedorn is their primary authority for development, direction, and review of all Chartway Technologies applications projects. He has over 15 years of highly varied and responsible ADP experience. He has worked in and managed all phases of a computer system's life-cycle. Mr. Hagedorn is the primary innovator of Chartway Technology methodology, which has been constructed to take the greatest advantage of the potentials inherent in the Application Productivity System (APS). He guides the ongoing refinement of the methodology and directs the technical enhancements being developed for the APS product. His experience also includes final management review authority for all Chartway Technologies applications projects.

Chartway Technologies

USING CASE TECHNOLOGY IN A LARGE ORGANIZATION

James R. Hagedorn
Chartway Technologies
Rockville, Maryland

THE CHALLENGE OF CASE IMPLEMENTATION

Chartway Technologies

- THE INDUSTRY HAS EXPERIENCED DIFFICULTY IN IMPLEMENTING CASE LARGELY BECAUSE OF UNANTICIPATED CULTURAL IMPACTS WITHIN THE ORGANIZATION

THE CHALLENGE OF CASE IMPLEMENTATION

Chartway Technologies

"OUR EXPERIENCE CAN BE SUMMED UP BY SAYING THAT ONE NEEDS TO LOOK AT CASE IN TERMS OF THE CASE STRATEGY, AND TO THINK ABOUT ALL THE THINGS THAT GO INTO CASE, NOT JUST THE TOOLS. TOOLS ARE PROBABLY JUST 10% TO 20% OF THE EQUATION. THINK ABOUT HOW YOU ARE GOING TO USE THEM. PREPARE TO MAKE SOME DRASTIC CHANGES IN THE ORGANIZATION, IF YOU ARE GOING TO MAKE CHANGES IN TOOLS AND METHODOLOGIES. THIS REALLY CHANGES THE WAY THAT MIS DOES BUSINESS."

– Tom Schwaninger, MIS Director, The Bekins Company

DEFINITION OF CASE

Chartway Technologies

CASE MEANS:

THE USE OF AUTOMATED TOOLS

IN SUPPORT OF STRUCTURED SOFTWARE ENGINEERING

CORE CASE TECHNIQUES

Chartway Technologies

- USE OF STRUCTURED ANALYSIS AND DESIGN APPROACHES TO AID REQUIREMENTS GATHERING AND TO STRENGTHEN SYSTEM DESIGN ARCHITECTURE
- USE OF PROTOTYPING TO INCREASE END-USER UNDERSTANDING OF THE SPECIFICATION AND DESIGN PROCESS
- USE OF A CENTRAL DICTIONARY TO CAPTURE, CONTROL AND RELATE REQUIREMENTS AND DESIGN DATA
- USE OF A CODE GENERATOR TO INCREASE PRODUCTIVITY DURING SYSTEM CONSTRUCTION
- USE OF A RELATIONAL DATA BASE

CORE CASE TOOLS

Chartway Technologies

- A GRAPHIC MODELLING TOOL TO SUPPORT STRUCTURED ANALYSIS AND DESIGN (IEW)
- A DICTIONARY (IEW)
- A PROTOTYPING TOOL (APS)
- A CODE GENERATOR (APS)
- A RELATIONAL DATA BASE (DB2)

LONG TERM GOALS FOR CASE IMPLEMENTATION

Chartway Technologies

1. INCREASE OVERALL USER SATISFACTION BY:
   A. WORKING WITH USERS MORE EFFECTIVELY TO DETERMINE USER NEEDS
   B. IMPROVING USER UNDERSTANDING OF WHAT WILL BE DELIVERED
   C. DELIVERING WHAT THE USER NEEDS MORE QUICKLY AND WITH HIGHER QUALITY
2. INTEGRATE CASE SMOOTHLY INTO THE CIS ORGANIZATION IN ORDER TO:
   A. INCREASE PRODUCTIVITY OF DEVELOPMENT STAFF
   B. SUBSTANTIALLY REDUCE MAINTENANCE COSTS

INCORPORATION OF OTHER CASE TECHNIQUES AND TOOLS

Chartway Technologies

THE IMPLEMENTATION OF THIS CORE WILL ALLOW THE REASONABLE INCORPORATION OF OTHER SUPPORTING TECHNIQUES AND TOOLS, E.G.:

- TESTING TOOLS
  - SOURCE CODE DEBUGGERS
  - TEST DATA GENERATORS
  - TEST SCENARIO RECORDERS
- CONFIGURATION MANAGEMENT TOOLS
- PROJECT MANAGEMENT TOOLS

SHORT TERM GOALS FOR CASE IMPLEMENTATION

Chartway Technologies

1. IMPLEMENT A PROJECT OR PROJECTS USING CASE
2. DEVELOP A COMPREHENSIVE TRAINING PLAN TO SUPPORT THE USE OF CASE
3. RAISE OVERALL CIS CONSCIOUSNESS REGARDING CASE AND ITS IMPACT ON THE LOWE'S WAY OF BUILDING SYSTEMS

TRADITIONAL APPROACHES TO CASE IMPLEMENTATION SUPPORT

Chartway Technologies

- VENDORS CONCENTRATE ON SELLING HARDWARE/SOFTWARE AND TRAINING IN SPECIFIC TOOLS
- THE MAJORITY OF CONSULTANTS WORK WITH INDIVIDUAL TOOLS AND, IN MANY CASES, WITH A SOFTWARE ENGINEERING APPROACH
- A FEW CONSULTANTS DISCUSS SYSTEM MANAGEMENT AND CONTROLS ISSUES

THE CHARTWAY MODEL FOR THE APPLICATION DEVELOPMENT ENVIRONMENT

Chartway Technologies

- CHARTWAY DIVIDES THE ADE INTO FOUR DISTINCT, THOUGH INTERRELATED AREAS:
  - THE HARDWARE/SOFTWARE PLATFORM
  - THE SOFTWARE ENGINEERING APPROACH
  - THE SYSTEM MANAGEMENT AND CONTROLS AREA
  - THE ENVIRONMENT SUPPORT AREA
- THESE AREAS ARE NOT STATIC BUT EVOLVE IN A CONTROLLED MANNER IN A CONTINUING PROCESS OF IMPROVEMENT

THE CONTEXT FOR CASE IMPLEMENTATION

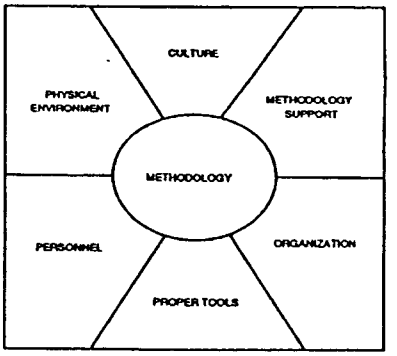
Chartway Technologies

- THE APPLICATION DEVELOPMENT ENVIRONMENT (ADE) IS THE PROPER CONTEXT FOR CASE IMPLEMENTATION
- THIS ENVIRONMENT CAN BE INTENTIONALLY STRUCTURED IN WAYS THAT FACILITATE THE INCORPORATION OF NEW TECHNOLOGY
- UNLESS THE ADE IS SO STRUCTURED, THE INCORPORATION OF NEW TECHNOLOGY (AND CASE) WILL BE JEOPARDIZED

A PRODUCTIVE DEVELOPMENT ENVIRONMENT INCLUDES:

Chartway Technologies

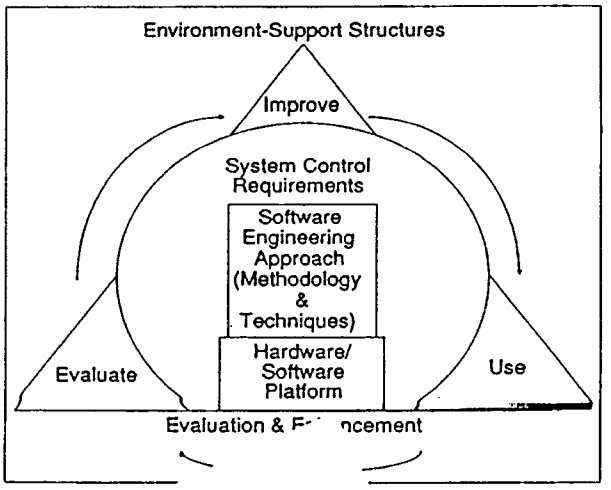
- A methodology
- A supportive culture
- Methodology support structure
- An organization that supports the methodology
- The right personnel
- A good physical environment
- The proper tools



THE CHARTWAY MODEL

Chartway Technologies

APPLICATION DEVELOPMENT ENVIRONMENT



SOFTWARE DEVELOPMENT PROJECT COMPLETED

Chartway Technologies

ORGANIZATION	SYSTEM NAME
Naval Sea Systems Command (NAVSEA)	• Customer Order Documentation System • Shipyard Automated Budget and Reporting System • Automated Funding Document System • Program Support Data Automated Reporting and Tracking System • Electronic Funds Transfer System • Chief of Naval Operations Gaming System
Army Personnel Center (CIVPERCEN)	• Headquarters Level - Army Civilian Personnel System (ACPERS)
Air Force Logistics Command (AFLC)	• Mode and Carrier Selection Module • Shipment Planning Address and Labeling System
Naval Military Personnel Command (NMPC)	• Officer Assignment Information System • Enlisted Assignment Information System

DOES THIS APPROACH WORK?

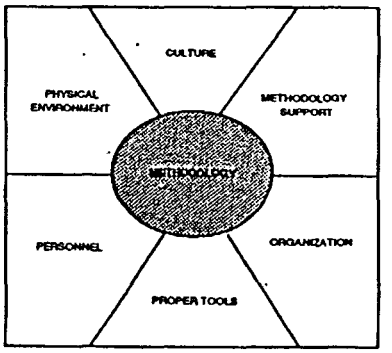
Chartway Technologies

- Based on CASE technology
- Over 15 years success in building applications this way
- Applies to small as well as large efforts
- Integrated into a staff of 200 developers located in 5 cities

PRINCIPLES OF A SUCCESSFUL METHODOLOGY

Chartway Technologies

- Establish standards
- End user involvement
- Solve problems through automation
- Incremental development
- Consistent productivity measures
- Should include re-engineering



ESTABLISH STANDARDS

Chartway Technologies

- All shops have "established standards"
- Management support and education needed to achieve buy-in to standards benefits
- Mechanism needed to support adherence
- Standards must not be restrictive but allow for creative application

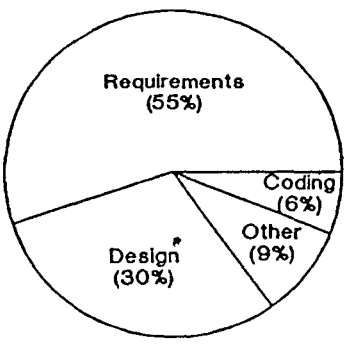
END-USER INVOLVEMENT

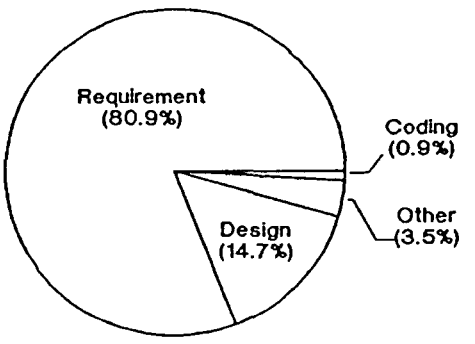
Chartway Technologies

- Get requirements right the first time
- Control risk
- Build iterative models
- Establish team roles

END-USER INVOLVEMENT CRITICAL

Chartway Technologies



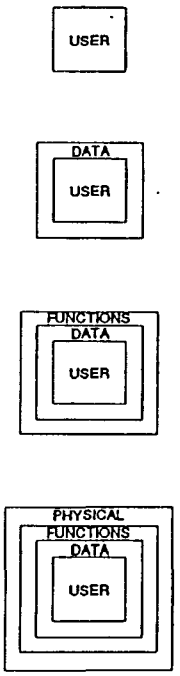


% Errors by Area (Maintenance Problems) ' Cost % of Errors for Each Segment

* from Prototyping Revisited, Jaime DeJesus, 1988

FOUR LEVELS OF PROTOTYPING

Chartway Technologies



Level 1: Requirements Definition Prototyping
- Screens
- Reports

Level 2: Functional Validation Prototyping
- Elements
- Tables
- Files
- Functional Baseline

Level 3: Functional Description
- Processing Logic
- Algorithms

Level 4: Physical Development
- DB/DC Environment

THE INCREMENTAL DEVELOPMENT AND DELIVERY

Chartway Technologies

THE PROBLEM

- Increasing number of large complex systems over budget and behind schedule

THE SOLUTION

- Review requirements to identify logical functional breakpoints
- Produce a tangible product at least every six months
- Establish core capability to demonstrate to end user
- Use an evolutionary development strategy

SOLVING PROBLEMS THROUGH AUTOMATION

Chartway Technologies

CURRENT PRACTICES

- Can not just throw people at the problem
- Today technology is thrown at problem

THE ANSWER

- Evaluate and select CASE tools based on gaps in life-cycle
- Build in absence of an available tool

PRODUCTIVITY MEASUREMENT

Chartway Technologies

THE PROBLEM

- Most tracking systems for software costs inaccurate
  - 30% to 50% error rate
- Frequent omissions: early planning, user costs, unpaid overtime, and certain overhead functions

THE SOLUTION

- Standard 25-activity chart of accounts
- Provides accountability at project and corporate level

STANDARD CHARTS OF ACCOUNTS

Chartway
Technologies

1. Requirements
2. Prototyping
3. System Architecture
4. Project Planning
5. Initial Analysis and Design
6. Detail Design and Specification
7. Design Reviews/Inspections
8. Coding
9. Reusable Code Acquisition
10. Purchased Code Acquisition
11. Code Reviews/Inspections
12. Independent Verification and Validation (Military Projects)
13. Configuration Control
14. Integration

STANDARD CHARTS OF ACCOUNTS (Continued)

Chartway
Technologies

15. User Documentation
16. Unit Testing
17. Function Testing
18. Integration Testing
19. System Testing
20. Field Testing
21. Acceptance Testing
22. Independent Testing (Military Projects)
23. Quality Assurance
24. Installation
25. Development Project Management

DEVELOPED BY S WARE PRODUCTIVITY RESEARCH, INC.

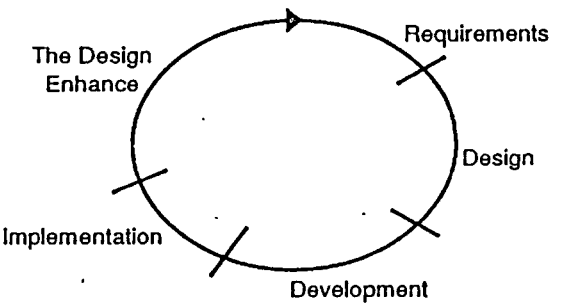
INCLUDE RE-ENGINEERING/MODERNIZATION

Chartway Technologies

TRADITIONAL LIFE-CYCLE

REQUIREMENT	DESIGN	DEVELOPMENT	IMPLEMENTATION	MAINTENANCE

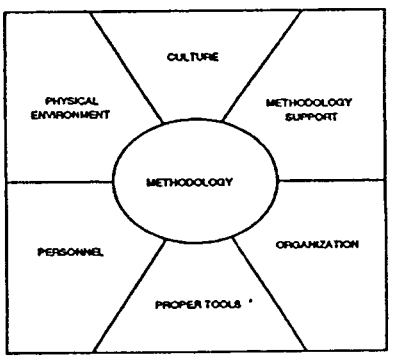
TODAY



OTHER ELEMENTS THAT SUPPORT THE METHODOLOGY

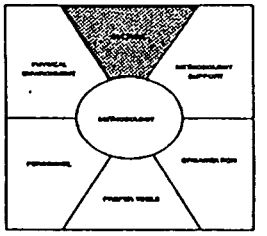
Chartway Technologies

- Culture
- Methodology Support
- Organization
- Tools Evaluation
- Personnel
- Physical Plant



CULTURE

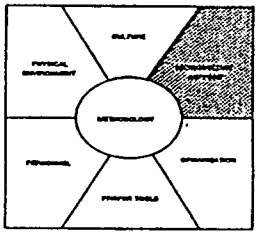
Chartway Technologies



- Must provide psychological support for methodology and the use of CASE
- Culture must:
  - support a corporate commitment to solving problems through automation
  - encourage the "better idea" and champion the "champions"
  - foster openness and sharing among project teams
  - support controlled flexibility
  - be incorporated in the corporate strategy
- Implementation issues
  - establish a process to generate new ideas
  - establish programs and procedures to share ideas
  - extend the focus of marketing department
  - monitor the culture regularly

METHODOLOGY SUPPORT

Chartway Technologies

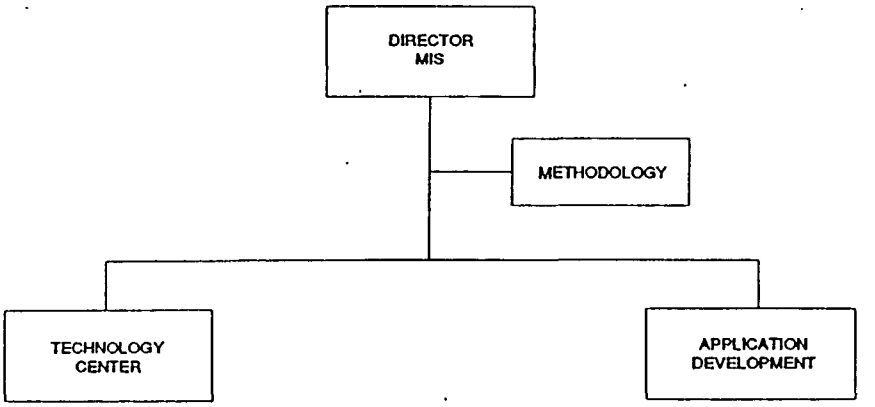


- The methodology should:
  - be living, not mechanical
  - be formative, rather than restrictive
  - emerge from experience
  - have formal procedures and structure to support and perpetuate it
- Structures which can focus support of the methodology

  Establish methodology support structures:
  - a methodology coordinator
  - a methodology control board
  - a methodology committee
  - formal levels of tool evaluation and integration
  - a[illegible] review of methodology

CORPORATE MIS ORGANIZATION

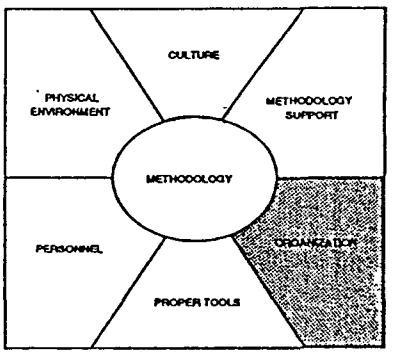
Chartway Technologies



- FACILITATES APPLICATION DEVELOPER ACCESS TO TOOLS AND METHODS
- PROVIDES STRUCTURES FOR LEVERAGING PROJECT LEARNING
- DEVELOPS AND IMPLEMENTS TRAINING
- EVALUATES NEW TECHNOLOGY
- PROVIDES ON-GOING TECHNICAL SUPPORT TO APPLICATION PROJECTS

ORGANIZATION

Chartway Technologies



- Organization structure will separate application projects from technical support group
- Organizational structure must support the methodology
- Technical support staff and project staffs must have open communication at every level
- Independent technology center established
- Application pr[illegible]t teams established around the [illegible]logy

TOOL EVALUATION GOAL

Chartway Technologies

- Determine
  - What functions
  - When
- To Find
  - The least number of tools to handle the most functions in the most phases of the process

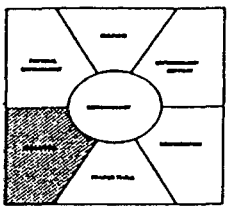
Chartway Technologies

TOOLS	Software Audit	Re-engineering			Maintenance		New Development
		Restructure	Repackage	Modernize	COBOL	CASE	
Data Analyzer	▒	▒			▒	▒	▒
Code Analyzer	▒	▒	▒	▒	▒		▒
Test Data/Path Generator	▒	▒	▒	▒	▒	▒	▒
Testing Record/Playback		▒	▒	▒	▒	▒	▒
Test Coverage Monitor	▒	▒	▒	▒	▒	▒	▒
Documentation Generator	▒	▒	▒	▒	▒	▒	▒
Cross Referencer	▒	▒	▒	▒	▒	▒	▒
Text and File Comparator		▒	▒	▒	▒	▒	▒
Restructurer		▒			▒		▒
Reformatter	▒	▒			▒		▒
Data Name Standardizer	▒	▒					▒
Translator		▒					
Screen Importer			▒	▒			
Data Normalizer			▒	▒			▒
Code Generator			▒	■		■	■
Prototyper				■		■	■
Reusable Code Facility			▒	▒	▒	■	■
Optimizer		▒	▒	▒	▒	▒	▒
Screen/Report [illegible]er				■		■	■

■ CASE-Level

PERSONNEL

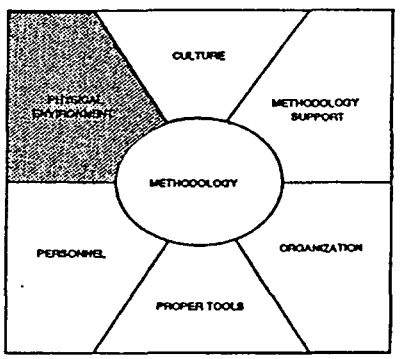
Chartway Technologies



- Technical staff must believe proper application of methodology means
  - corporate success
  - personal success
- Technical staff must cross-train
- The technical staff must regard itself as a team
- Suggestions for professional development of your technical staff:
  - hire staff with the methodology in mind
  - require at least 40 hours training per year
  - include team building activities into environment
  - design generous, purposeful reward structure

THE PHYSICAL ENVIRONMENT

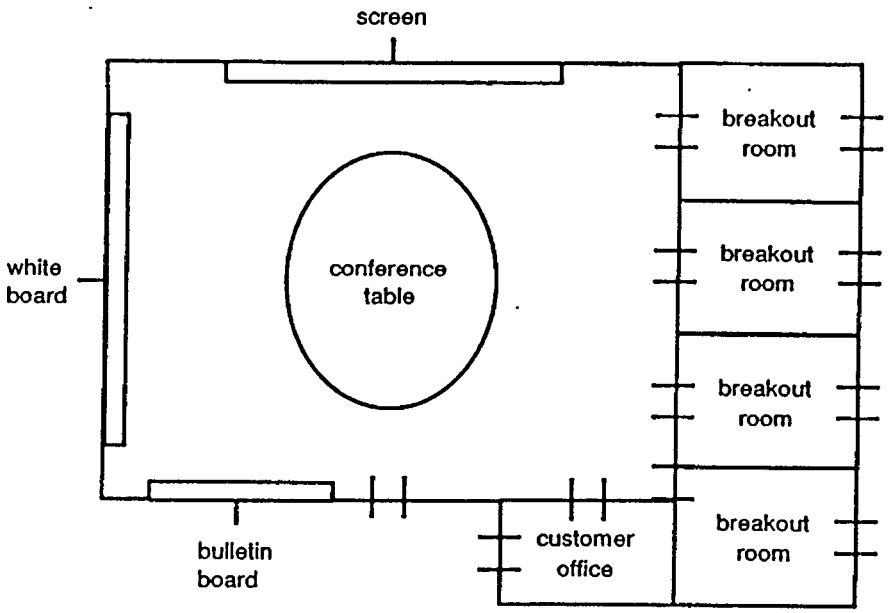
Chartway Technologies



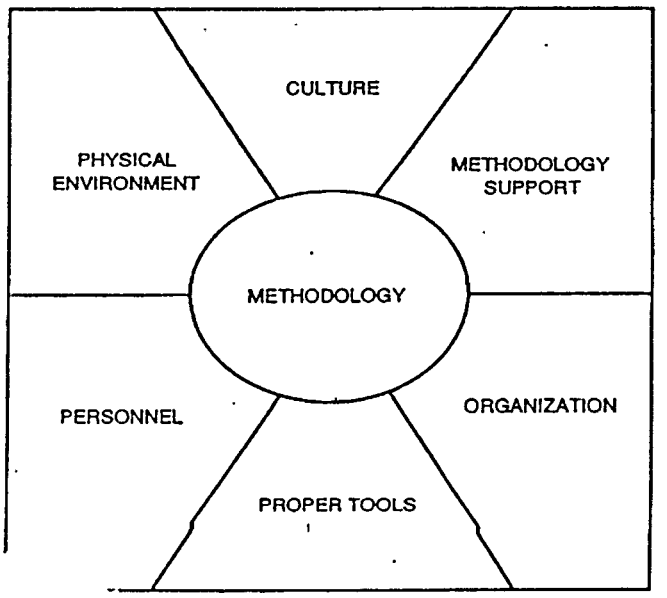
- Physical design must allow staff to function as professional software engineers
- Design layout to support methodology
- Give developers enough room, air, and lighting
- Give developers the right tools

PROTOTYPING LABORATORY

Chartway Technologies



Chartway Technologies



Paper 2-A-2

RISK REDUCTION THROUGH REQUIREMENTS TRACING

Mrs. Jane Huffman Hayes
Member of Technical Staff
SAIC

Mrs. Jane Huffman Hayes received the Master of Science degree in Computer Science from the University of Southern Mississippi in May of 1987. She currently works for Science Applications International Corp. as the Assistant Principal Investigator for the Independent Verification and Validation (IV&V) of the Tomahawk Mission Planning Center upgrade. Mrs. Hayes is a member of the ACM and the IEEE Computer Society. Her interests include software maintenance, documentation, metrics, software testing, and requirements analysis.

Risk Reduction Through Requirements Tracing

20 April 1990

Jane Huffman Hayes
Science Applications International Corporation
1213 Jefferson Davis Highway
Suite 1300
Arlington, Virginia 22202
(703) 553-6149

* Work described in this paper was partially funded by the Cruise Missiles Project of the U.S. Navy.

Risk Reduction Through Requirements Tracing

Why Perform Requirements Tracing?

It has been estimated that a problem identified during testing costs 20 - 40 times more to correct than a problem detected during the preparation of requirements documents [1]. One of the most costly errors to correct is the omission of requirements from a system. Basili and Weiss [2] found that thirty-one percent of the requirements errors detected on a project were incomplete (missing) requirements. Such an error, discovered during testing, necessitates the update of many requirements and operational documents (Software Requirements Specifications/Interface Requirements Specifications, Software Design Documents, System User's Manual, etc. under DOD-STD-2167A) and the development, integration, and testing of additional source code. Requirements tracing, a technique which detects requirement omissions and incomplete or incorrect requirement satisfaction, greatly reduces cost, schedule, and technical risks to a software development effort.

Requirements tracing verifies the completeness, traceability, and consistency of a system. Completeness refers to a parent requirement (in an A level document, for example) being fully satisfied by all the children requirement (in a B level document, for example) it has linked to it. Traceability refers to every child requirement (in a B level document, for example) linking back to a parent requirement (in an A level document, for example). Consistency refers to the uniformity of requirements across a document level. For example, the Software Requirements Specification for each Computer Software Configuration Item should address reliability and maintainability comparably. Computer-Aided Software Engineering tools have not tackled this problem successfully.

Many Computer-Aided Software Engineering tools allow a designer to ensure consistency and "balancing" of one level of design (requirements specification or preliminary design, for example), but do not allow consistency or completeness checks between levels of design (did all the requirements flow down into the preliminary design?). This is further complicated when different design techniques are used by the tool, for example data flow diagrams may be used for requirements analysis, Ada structure graphs may be used for preliminary design, and Ada Development Language may be used for detailed design. The task of verifying traceability for such a "mixed media" design becomes a formidable one.

In light of this, manual requirements tracing for a system of any significant size becomes arduous. The problem of "mixed media" (data flow diagrams, structure graphs, Program Development Language, etc.) can be resolved by utilizing the requirements documented in the Software Requirements Specifications/Interface Requirements Specifications, Software Design Documents, etc. rather than attempting to use the diverse design media. Text processing is a mature capability which has further been enhanced by technologies such as hypertext. Unfortunately, the extraction of requirements from requirements and design documents and the process of mapping requirements from one document level to another is manpower intensive and error prone. Two tools, SuperTrace and SuperTrace Front End Processor (SFEP), have been used successfully in combination to perform requirements tracing as part of the Independent Verification and Validation (IV&V) of the upgraded Theater Mission Planning Center.

SuperTrace and SFEP - Requirements Tracing Tools

SuperTrace was developed by the Command Systems Division of the Systems Technology Group of Science Applications International Corporation (SAIC) under contract to the Cruise Missiles Project of the U.S. Navy. SuperTrace is written in SuperCARD (C) and runs on an Apple Macintosh II. Information can be entered, retrieved, and/or modified for requirements/design documents using SuperTrace. For requirements within these documents, the section number, page number, security classification, requirement text, keywords, forward links, backward links, and the reviewing analyst's name can be entered/retrieved/modified. Once the requirement text and keywords have been entered, SuperTrace can be used to generate a list of candidate forward links (System Specification requirement #1 is a candidate link to Software Requirements Specification requirements # 75, 77, and 79) for all requirements in a document. These candidate links are verified by an analyst and are then entered/accepted in SuperTrace. Finally, a list of anomalous requirements (System Specification requirement with no "children" requirements in the Software Requirements Specification, Software Requirements Specification requirement with no "parent" requirement in the System Specification) is generated.

After using SuperTrace for two levels of documents (System Specification and Software Requirements Specifications/Interface Requirements Specifications) for the IV&V of the Tomahawk Land Attack Missile Planning System (a subset of the upgraded Theater Mission Planning Center), the need to accelerate the entry of requirements became obvious. To meet this need, the SuperTrace Front End Processor (SFEP) was developed by the Command Systems Division of Science Applications International Corporation. SFEP is written in Pascal and runs on IBM personal computers and compatibles. The SFEP partially automates the process of extracting requirements from

documents. The analyst edits a softcopy of the requirements document, annotating ("marking") requirements with predefined symbols. Macros are used to further expedite the process of marking requirements. SFEP parses the document, extracts marked requirements, and builds several data files which can then be directly loaded into the SuperTrace database. SFEP also builds two "human readable files" which allow the analyst to ensure that all requirements were extracted from the document (ensure that the analyst "marked" all requirements). SFEP uses Microsoft Word (C) for all word processing functions.

The Development Project

SuperTrace and SFEP have been used by Science Applications International Corporation to perform requirements tracing as part of the IV&V of the Digital Imagery Workstation Suite, Tomahawk Land Attack Missile Planning System, and Mission Data Distribution System segments of the upgraded Theater Mission Planning Center for the Cruise Missiles Project. The design and test requirements have been traced using SuperTrace and SFEP to ensure that all system requirements are met and that an adequately tested system will be delivered. The upgraded Theater Mission Planning Center provides the Unified and Specified Commanders of the U.S. Navy with the capability to develop Mission Data and supporting Command and Control information for the Tomahawk Land Attack Missile [3].

The Digital Imagery Workstation Suite provides the image management and precise mensuration needed to exploit imagery in support of the overall upgraded Theater Mission Planning Center operation. The Tomahawk Land Attack Missile Planning System provides the production planning, control, and monitoring capabilities for the entire Theater Mission Planning Center as well as the capability to plan missions for the

variants of the Tomahawk Land Attack Missile. The Mission Data Distribution System provides services for the preparation and certification of Data Transport Devices, preparation and distribution of Mission Data Updates, preparation and distribution of Tomahawk Land Attack Missile Command Information, and maintenance of accountability data [4].

Cruise Missiles Project began development of the Digital Imagery Workstation Suite well over a year prior to contract award for the Tomahawk Land Attack Missile Planning System and Mission Data Distribution System segments to reduce technical and schedule risk. As a result, the three segments are now at different phases of the software development life cycle. The Digital Imagery Workstation Suite has completed Critical Design Review and is completing coding and unit testing. Tomahawk Land Attack Missile Planning System has completed Preliminary Design Review and is completing detailed design. Mission Data Distribution System has completed Preliminary Design Review and is performing detailed design.

Figure 1 (3 parts) shows the design document hierarchy for the Digital Imagery Workstation Suite, Tomahawk Land Attack Missile Planning System, and Mission Data Distribution System development projects. To date, the Digital Imagery Workstation Suite requirements have been traced (using SuperTrace) to the Software Detailed Design Document level and will soon be traced to source code. The Tomahawk Land Attack Missile Planning System requirements have been traced to the Software Requirements Specification level using SuperTrace and to the Software Design Document (Preliminary) level manually. The Mission Data Distribution System requirements have been traced to the System/Segment Specification level using SuperTrace. Unfortunately, data was not consistently gathered on these requirements tracing activities. The requirements tracing activities performed to evaluate the test effort for Tomahawk Land Attack Missile

Planning System, however, have been carefully monitored and tracked. Figure 2 shows the test document hierarchy for the Tomahawk Land Attack Missile Planning System. This article will focus on the requirements tracing of the test requirements of the five Computer Software Configuration Items (CSCIs) that comprise the Tomahawk Land Attack Missile Planning System (Process Control (PC), Data Management (DM), Mission Route Planning (MRP), Detailed Independent Quality Verification Check (DIQVC), and Standard Application Environment (SAE)) using SuperTrace and SFEP.

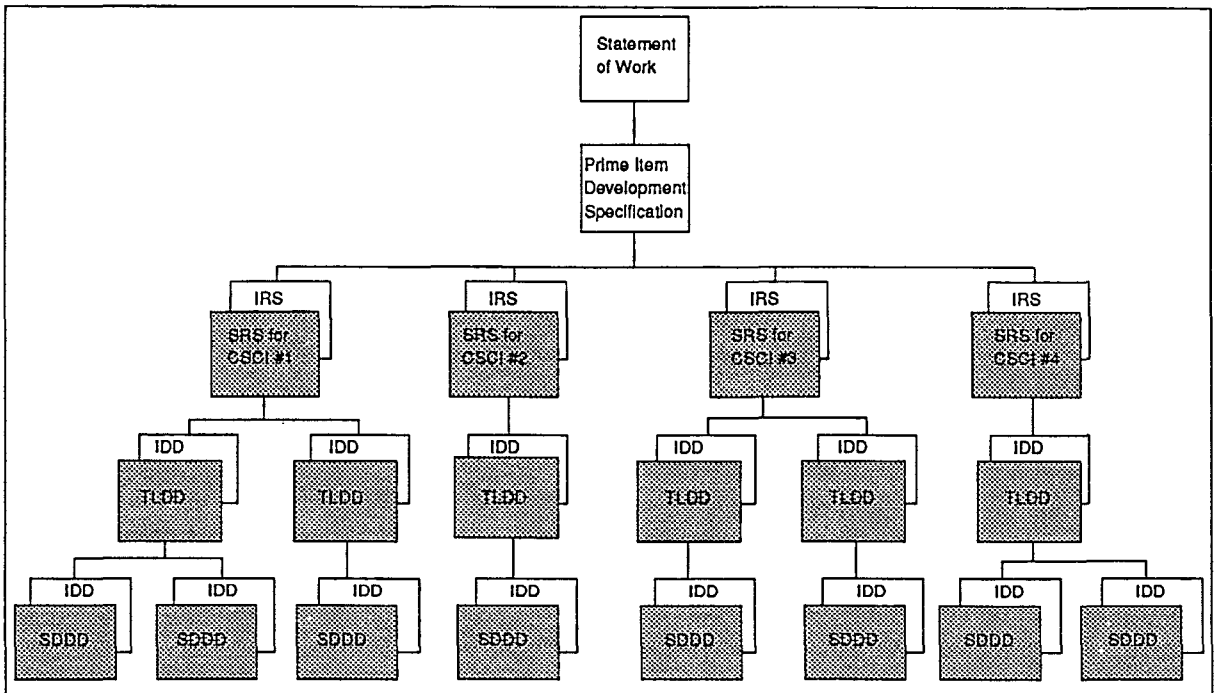


Fig. 1. Digital Imagery Workstation Suite Design Document Hierarchy (DOD-STD-2167).

leeefig1

1 of 3

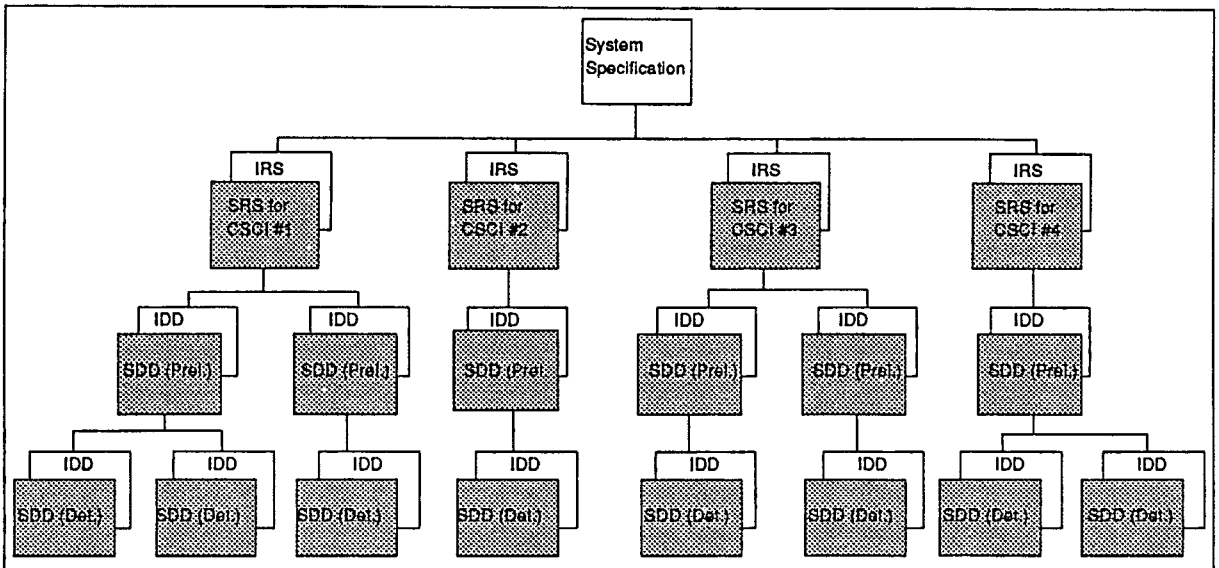


Fig. 1. Tomahawk Land Attack Missile Planning System Design Document Hierarchy (DOD-STD-2167A).
2 of 3

ieeefg1a

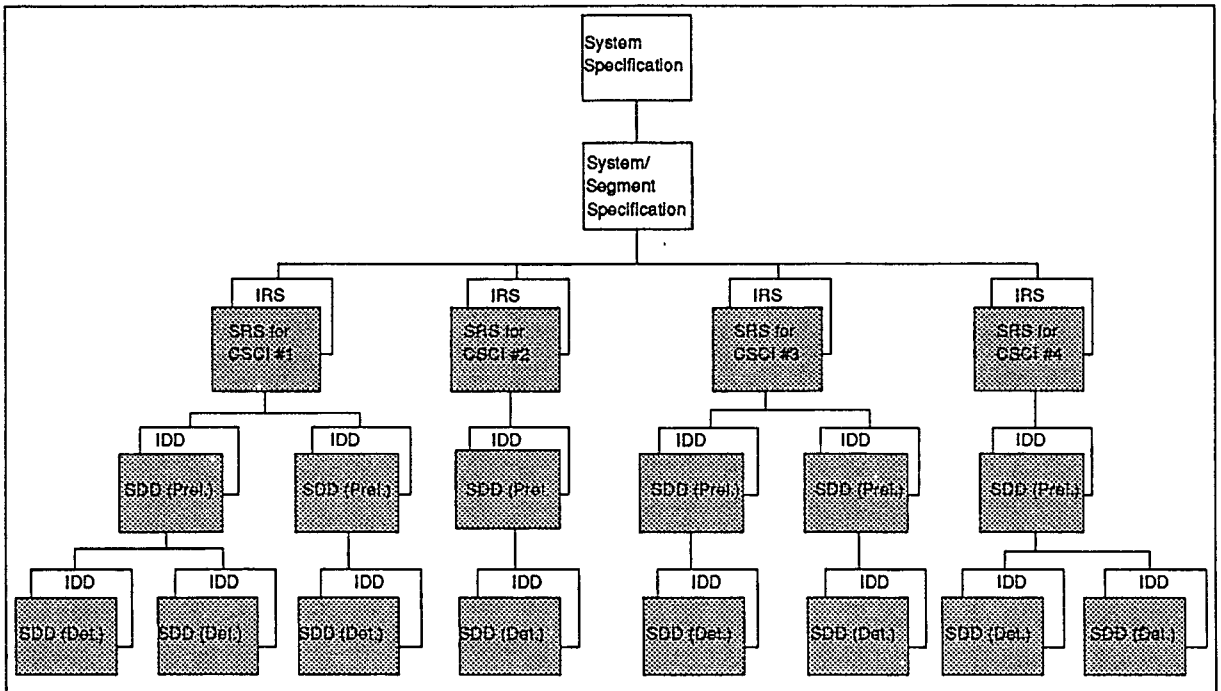


Fig. 1. Mission Data Distribution System Design Document Hierarchy (DOD-STD-2167A).
3 of 3

ieeefg1b

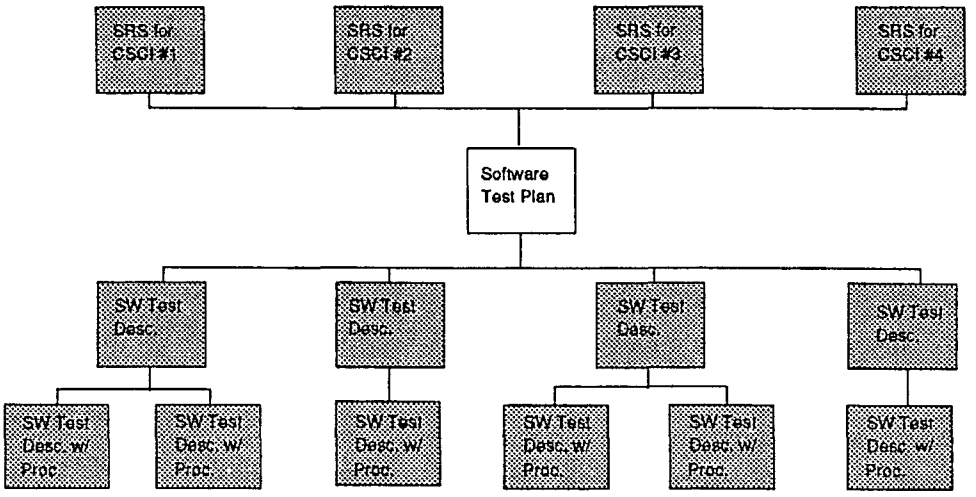


Fig. 2. Test Document Hierarchy for Tomahawk Land Attack Missile Planning System (DOD-STD-2167A).

ieeefig2

Requirements tracing for the Tomahawk Land Attack Missile Planning System test effort began in October of 1989. The learning curve for SuperTrace and SFEP had been overcome during Digital Imagery Workstation Suite and Tomahawk Land Attack Missile Planning System design document tracing activities. Requirements Tracing Data Collection Sheets were distributed (see figure 3) to each team member. The sheets list the possible requirements tracing activities that could be performed as well as the various document levels. Some tracing activities were performed manually, others using SuperTrace. Certain automated activities were divided with half being performed manually and half using SuperTrace for comparison purposes. The tracing information was gathered, and then entered into a Lotus 1-2-3 (C) spreadsheet for analysis. The analysis results are presented here.

REQUIREMENTS TRACING DATA SHEET

DATE:_____

CSCI/SEGMENT/SYSTEM:

DM_____ IDI_____SR _____IDS _____

MRP_____ SAE_____PC _____WS _____

DIQVC_____ TPS (Segment)_____DIWS (Segment) _____

SCI Isolation (Segment)_____ MDDS (Segment) _____

TMPCU (System)_____Other (Specify) _____

DOCUMENT(S):

System Spec _____ SOW_____ PIDS____ SRS _____

IRS _____SDD (Prel.)____ SDD (Det.)____IDD _____

TLDD____ SDDD_____ Other (Specify) _____

ACTIVITY (Check one from each column):

Marking and extracting requirements _____
Entering requirements _____
Assigning major keywords _____
Entering major keywords _____
Generating candidate link report_____
Verifying candidate links _____
Entering hard links _____
Generating flowdown report _____
Verifying completeness _____
Updating hard links _____
Documenting anomalies _____
Other (Specify) _____

Manual_____
SFEP _____
Supertrace _____
MS WORD _____
KEYWORD _____

NUMBER OF REQUIREMENTS: _____

TIME SPENT ON ACTIVITY (e.g., hours, days): _____

IS THIS AN ESTIMATED OR ACTUAL VALUE? _____

Fig. 3. Requirements Tracing Data Collection Sheets.

fig3

The Requirements Tracing Process

Entry of Requirements

The requirements tracing process is composed of several different activities shown in figure 4. First, requirements must be entered into SuperTrace. To eliminate interactive entry of textual requirements into SuperTrace, SFEP is utilized. Specified delimiters are placed in the textual document to "tag" requirements. Since many of the design documents present requirements in a standard format , entry of these delimiters has been accelerated by the use of macros. For example, in "marking" requirements in a Software Requirements Specification, macros can be used to locate and annotate with

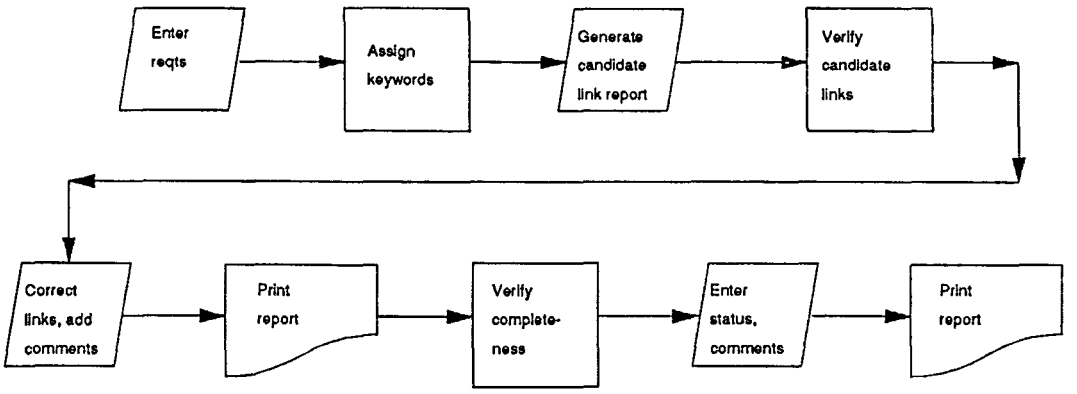


Fig. 4. Requirements Tracing Activities.

rqtflow

delimiters the Input, Processing, and Output sections. Other sections, such as Adaptation Requirements and Quality Factors, would be "delimiterized" manually by the analyst utilizing Microsoft Word (C). Also, macros can be built to key on phrases such as "shall," "will," etc. Once the document has been "marked," SFEP parses the embedded delimiters and generates the following flat files containing the extracted requirements:

FILE.DF1	flat file for requirement IDs and for later keyword entry
FILE.DF2	flat file for document specific information
FILE.DF3	flat file for requirement ID, section number, section title, text
FILE.DF4	flat file of all requirement IDs (index built from this)
FILE.HRF	lists each requirement in a readable format
FILE.TXT	lists all the text that was not marked as a requirement

Figure 5 (5 parts) shows the extraction process.

SuperTrace Front End Processor ◘ Selection Menu File: FILE.DOC

Select an option by NUMBER:

1 ◘ Mark / Re-Mark a document

2 ◘ Extract requirements from a completely marked document

3 ◘ Review/ Print extracted requirements in Microsoft Word

4 ◘ Quit the SuperTrace Front End Processor

Your selection >

Fig. 5. Requirements Extraction Process - SFEP Main Menu.
1 of 5

sfep1

SuperTrace Front End Processor ◘ Document Information Screen

Information for FILE.DOC

System Name: Beginning Req. ID:

Document Title:

Document Publisher:

Document ID: Document Version:

Document Date: Document Type:

Document Entry: Document Desig.: Cust. ID:

Satisfied with Information (Y/n) ?

TAB/BACKTAB to change fields ◘ ESC for Menu

Fig. 5. Requirements Extraction Process - SFEP Document Information Screen.
2 of 5

sfep2

```
L[ | | 1 | | | 2 | | | 3 | | | 4 | | | ] | | |
<~3.2.1.1.1.2.1 Mission Data Distribution. The M/TDDS shall provide Block
III TLAM mission data and related TCI to launch platforms, by specifically
providing:~>

[[~a. Time of arrival (TOA), information~]

[~b. Multiple Operational Flight Programs (OFPs) on production DTDs. A default
 OFP shall be identified on a DTD~]

[~c. GPS Flight Software (GFS) data (conventional only)~]

[~d. DSMAC IIA Flight Software (DFS) data (conventional only)~]

[~e. Modifications to the TCI and TCIPs.~]]
                                                    MDDSSSS1.DOC
COMMAND: Copy Delete Format Gallery Help Insert Jump Library
         Options Print Quit Replace Search Transfer Undo Window
Edit document or press Esc to use menu
Pg12 Co2        {}              ?                   Microsoft Word
```

Fig. 5. Requirements Extraction Process - Inserting Delimiters.

sfepword

3 of 5

```
L[ | | 1 | | | 2 | | | 3 | | | 4 | | | ] | | |
                B00002
            1
                B00002
            2
                B00002
            3
                B00002
            4
                B00002
            5
                B00002
            6
                B00002
            7
                B00002
            8
                B00002
            9
                B00002
           10
                B00003
            1
                B00003
                                                    MDDSSSS1.DF1
COMMAND: Copy Delete Format Gallery Help Insert Jump Library
         Options Print Quit Replace Search Transfer Undo Window
Edit document or press Esc to use menu
Pg12 Co2        {}              ?                   Microsoft Word
```

Fig. 5. Requirements Extraction Process - Sample Extraction File.

sfepdf1

4 of 5

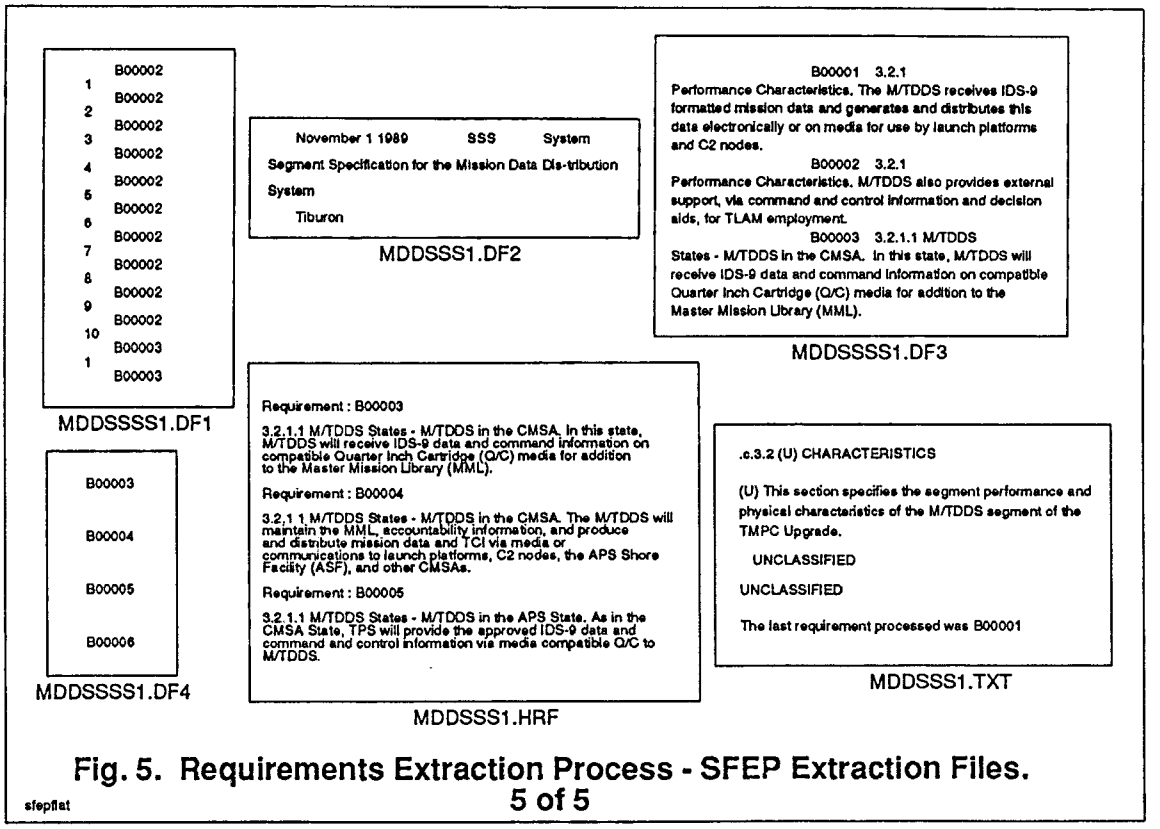


Fig. 5. Requirements Extraction Process - SFEP Extraction Files.
5 of 5

sfepflat

Assignment of Keywords

The next requirement tracing activity is the assignment of keywords to requirements. For each project, a hierarchical keyword tree is designed by the analysts to represent the logical decomposition of the design. These keywords are then assigned to the extracted requirements. The keyword tree is expanded to represent the detail of the design as the life cycle progresses. SuperTrace uses these keywords to make candidate links between parent and child requirement levels (System Specification to Software Requirements Specification). Figure 6 shows a subset of the keyword hierarchy for the Tomahawk Land Attack Missile Planning System.

```
1   TMPCU
2     T-CAPACITY
3     T-VARIANTS
4     T-FLEX-LCH-PT
5     T-RAPID-RETGT
6     T-FLEX-TGT
7     T-RESPONSE
8     T-PRODUCTION
9     T-REPLAN
10    T-DATA-PREC-ACCURACY
11    T-INTEROPERABILITY
12    T-SECURITY
13    T-DB-STORAGE
14    T-STATUS
70        T-STATUS-PC
15    T-REPORTS
16    T-CMD-REVIEW
17    T-MULTI-ROUTE
18    T-OPER-EMPLOY
19    T-STRIKE-PKG
20    T-TGT-WEAPON
21    T-INPUT-DATA-SOURCE
22    T-C2-NODE-INTERFACE
23    T-OTHER-INTERFACE
```

Fig. 6. Subset of Keyword Hierarchy.

Generation of Candidate Link Report

Using the assigned keywords, SuperTrace generates a candidate link report. This report presents a list of the parent requirements and their candidate children requirements. For example, in tracing from the System Specification (A level document) to the Software Requirements Specification (B level document), the candidate link report would have this format:

A00001	B00230, B00234, B00436
A00002	
A00003	B00001, B00003, B00006, B00023, B00024

In this case, requirement A00001 of the System Specification has as candidate children requirements B00230, B00234, and B00436 of the Software Requirements Specification.

Requirement A00002 of the System Specification has no candidate links to the B level requirements. A00003 is potentially linked to B00001, B00003, B00006, B00023, and B00024.

Verification of Candidate Links

After generating the candidate link report, an analyst must verify the candidate links. This is also referred to as verifying traceability. In other words, the analyst must ensure that all children requirements (B level) link back to a parent requirement (A level) or are derived requirements. This can be accomplished manually (analyst leafs through A level document and B level document) or interactively using SuperTrace. SuperTrace allows the analyst to examine the A level requirement and the B level candidate links using a split screen (multiple windows). The analyst can accept, reject, or modify candidate links. SuperTrace also provides a "Keyword in Context Search" that allows the analyst to search on key phrases to find candidate links for requirements without candidate links (such as A00002 in the example above).

Acceptance/Correction of Candidate Links

If the analyst uses SuperTrace to verify the candidate links, as described above, the acceptance and correction of links would be accomplished interactively. If the analyst performs this activity manually, however, the "correct" (verified) candidate links must still be entered into SuperTrace. Two approaches can be used: 1) the analyst can use SuperTrace to interactively update the candidate links (figure 7 shows the SuperTrace display for this activity); or 2) the analyst can use Microsoft Word (C) (or some alternate word processing package) to modify a softcopy of the candidate links report. The softcopy data is then loaded back into SuperTrace as a flat file (much like the SFEP data is loaded into SuperTrace initially).

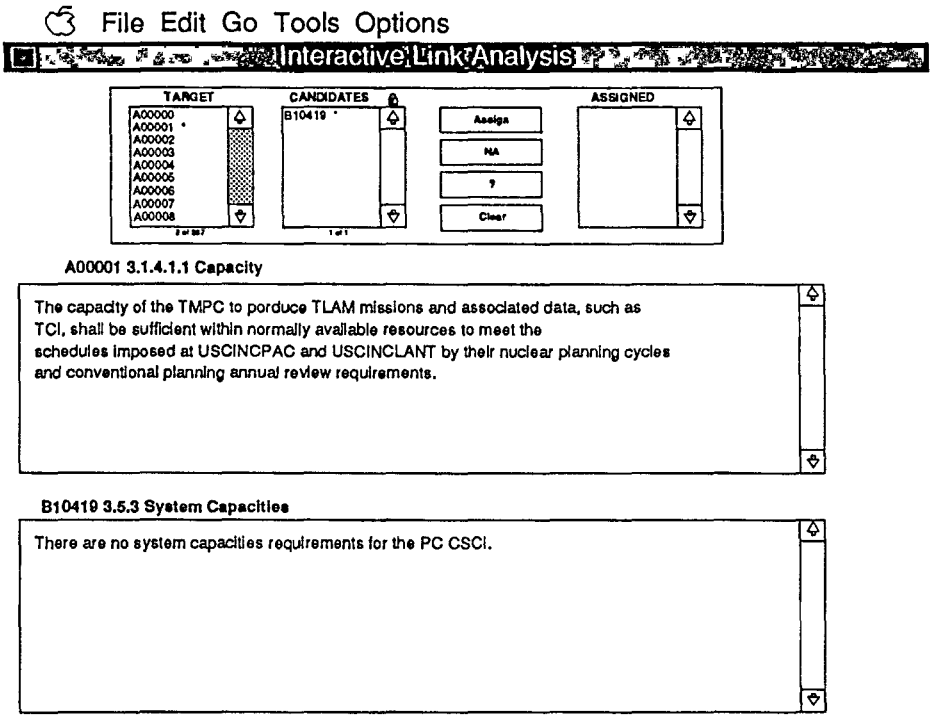


Fig. 7. Acceptance/Correction of Candidate Links.

Report Generation

In addition to generating the candidate links report, SuperTrace can be used to generate many other reports. For example, after updating the candidate links the analyst may generate a report of the "hard" links between the A and B level documents. The analyst can print reports listing all the requirements for a particular document level. A "table of contents" for a document level can be printed which presents the section numbers, section titles, and page numbers for a document. The analyst can also print a report of any requirement anomalies or analyst comments that have been entered. SuperTrace generates reports and then places them in the Microsoft Word (C) directory for later printing.

Verification of Completeness

Now that traceability has been verified (B level requirements have a parent requirement in the A level document), completeness must be verified. Completeness refers to an A level requirement being fully satisfied by all the B level children requirements linked to it. Verification of completeness is performed much the same way that candidate links were verified. The analyst can perform the activity manually utilizing paper copies of each document. SuperTrace can also be used to view the A level parent requirement and one B level child requirement at a time. Status information (parent requirement is completely satisfied, partially satisfied, or omitted by the child document) is entered by the analyst (either using SuperTrace or by generating a flat file in Microsoft Word (C)). In the case of partially satisfied requirements, the analyst should enter a comment in SuperTrace that describes the portion of the requirement that has not been addressed by the child document. This status information and analyst comments can then be used to generate reports. Such reports will help the software developer correct requirement anomalies at an early stage of the design process. Figure 8 shows a sample anomaly report.

entry, some requirements were processed using SuperTrace's keyword entry tool (Keyword) and some using a flat file generated manually in Microsoft Word (C). The comparison of manual and automated processes showed that SFEP and SuperTrace greatly expedited the entry of requirements and the entry of keywords. Figure 9 shows a bar chart of the comparison.

The data collected for all the requirements tracing activities were stored in a Lotus 1-2-3 (C) spreadsheet. A subset of this spreadsheet is shown in table 1. The data in the spreadsheet was used to calculate mean values for all requirement tracing activities and to calculate the Point-Biserial for activities performed both manually and automatically. The Point-Biserial equation (r_{pb}) is shown below [5]:

$$r_{pb} = \frac{M_2 - M_1}{S} \sqrt{\frac{n_1 n_2}{N(N-1)}}$$

where:

n_1 = number of scores in category 1 (Manual/Word)

n_2 = number of scores in category 2 (SFEP/SuperTrace)

$N = n_1 + n_2$

S = standard deviation of all N scores

M_1 = mean score of category 1

M_2 = mean score of category 2

DIQVC SRS Reqts Partially Addressed by SW Test Plan

B50016	The SW Test Plan does not address "User requests for...PC."
B50017	The SW Test Plan only addresses this requirement superficially.
B50018	The SW Test Plan does not address "The GFS executes from...CSCIs."
B50019	The SW Test Plan only addresses this requirement superficially.
B50020	The SW Test Plan does not address "the DIQVC interfaces...tasking requirements."
B50039	The SW Test Plan does not address "the DIQVC Executive acts as a control interface...other DIQVC functions."
B50042	The SW Test Plan does not address "the file IDs for the VEM/RLS...Mission Data Log file."

Fig. 8. SuperTrace Anomaly Report.

The Findings

The requirements tracing process described above was utilized for two levels of test documents for the Tomahawk Land Attack Missile Planning System (the Software Requirements Specifications and the Software Test Plan). All activities with the potential for automation were performed using SuperTrace and SFEP, with the exception of requirement entry and keyword entry. These activities were divided so that half of the requirements were processed automatically using SuperTrace and SFEP, and half were processed manually. For requirement entry, SuperTrace and SFEP were used to process some requirements, the remainder were entered with Microsoft Word (C). For keyword

TABLE 1. SUBSET OF LOTUS 1-2-3 SPREADSHEET FOR REQUIREMENTS TRACING ACTIVITIES.

VERIFYING CANDIDATE LINKS

DATE	# OF RQTS/HR	# OF REQTS	TIME SPENT (HOURS)	CSCI	SEGMENT	DOCUMENT LEVELS	METHOD	EST./ACT
10/5/89	16.46875	527	32	MRP	TPS	A-SPEC to SRS	Manual/WORD	Estimate
12/11/89	128.5714	36	0.28	PC	TPS	SRS to SW Test Plan	Manual	Actual
12/14/89	25.18518	340	13.5	MRP	TPS	SRS to SW Test Plan	Manual	Actual
12/15/89	156.4220	341	2.18	SAE	TPS	SRS to SW Test Plan	Manual	Actual
12/15/89	47.91666	46	0.96	DIQVC	TPS	SRS to SW Test Plan	Manual	Actual
12/15/89	4.047619	85	21	DM	TPS	SRS to SW Test Plan	Manual	Actual

ENTERING HARD LINKS

DATE	# OF RQTS/HR	# OF REQTS	TIME SPENT (HOURS)	CSCI	SEGMENT	DOCUMENT LEVELS	METHOD	EST./ACT
11/3/89	160	800	5	N/A	DIWS	PIDS to SRS	SUPERTRACE	Actual
11/27/89	66.66666	200	3	(Critical)	DIWS	SRS to SDDD	SUPERTRACE	Actual
12/18/89	88.88888	400	4.5	IDI/IDS	DIWS	SRS to SDDD	SUPERTRACE/WORD	Actual

VERIFYING COMPLETENESS

DATE		# OF REQTS	TIME SPENT (HOURS)	CSCI	SEGMENT	DOCUMENT LEVELS	METHOD	EST./ACT
11/1/89	20.66666	93	4.5	N/A	TMPCU	A-SPEC TO SRS	Manual	Actual
11/2/89	9.846153	64	6.5	N/A	TMPCU	A-SPEC TO SRS	Manual	Actual
12/8/89	18.33333	88	4.8	N/A	TMPCU	A-SPEC TO SRS	Manual	Actual
12/27/89	83.33333	50	0.6	DIQVC	TPS	SRS to SW Test Plan	Manual	Actual
12/27/89	53.84615	7	0.13	DM	TPS	SRS to SW Test Plan	Manual	Actual
12/29/89	103.8461	27	0.26	DM	TPS	SRS to SW Test Plan	Manual	Actual
1/2/90	17.64285	247	14	MRP	TPS	SRS to SW Test Plan	Manual	Actual
1/5/90	13.38095	281	21	SAE	TPS	SRS to SW Test Plan	Manual	Actual
1/8/90	62.09375	1987	32	DM	TPS	SRS to SW Test Plan	Manual	Actual
1/11/90	13.30434	306	23	PC	TPS	SRS to SW Test Plan	Manual	Actual

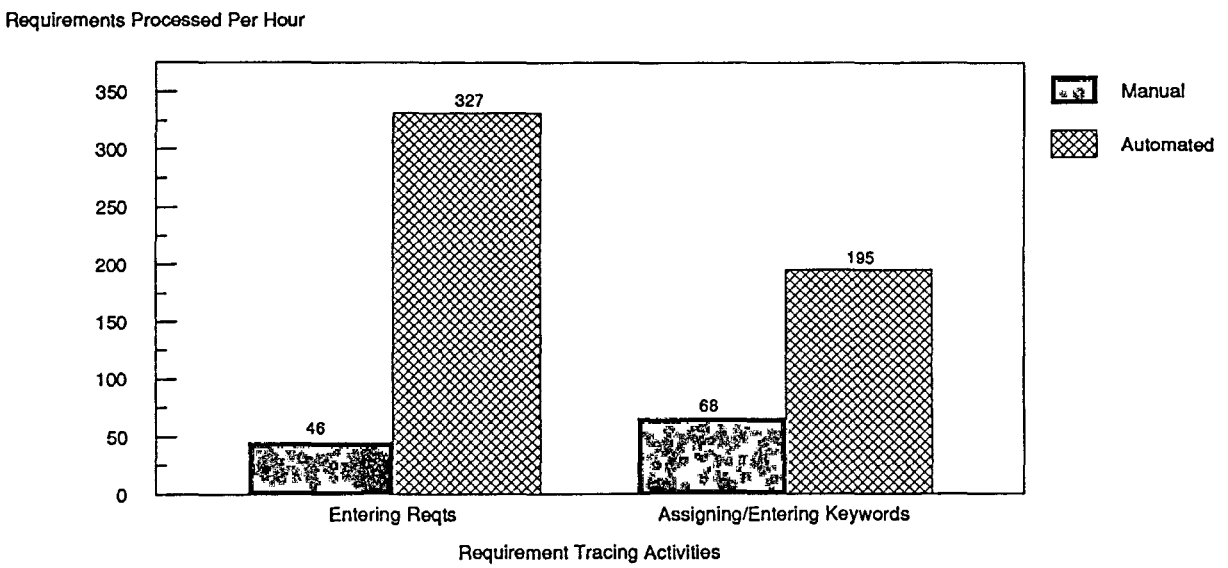


Fig. 9. Comparison of Manual and Automated Requirements Tracing.

rqts

In addition, for each r_{pb} calculated, uncertainty was also calculated. The student t distribution ($t_{d,p}$) was used for this calculation. The equation is shown below [6]:

$$t_{d,p} = r_{pb} \sqrt{\frac{N-2}{1-r_{pb}^2}}$$

Using the $t_{d,p}$ table [7], probability was obtained. Uncertainty (1 - probability) was then calculated. The results of these calculations are presented in table 2.

TABLE 2. STATISTICAL DATA FOR REQUIREMENTS TRACING ACTIVITIES.

ACTIVITY	CATEGORICAL VARIABLE (METHOD)	NUMERICAL VARIABLE (REQTS PROCESSED PER HOUR)	r_{pb}	CORRELATION	UNCERTAINTY
MARKING AND EXTRACTING REQUIREMENTS	NA	62.35 MEAN	NA	NA	NA
ENTERING REQUIREMENTS			0.9687	VERY HIGH	< 0.05%
	MANUAL/WORD	46.626 MEAN			
	SFEP/SUPERTRACE	327.01 MEAN			
ASSIGNING/ENTERING MAJOR KEYWORDS			0.5804	MODERATE	11.57%
	MANUAL	68.649 MEAN			
	KEYWORD	195.619 MEAN			
GENERATING REPORTS	NA	2065.57 MEAN	NA	NA	NA
VERIFYING CANDIDATE LINKS	NA	63.10 MEAN	NA	NA	NA
ENTERING HARD LINKS	NA	315.55 MEAN	NA	NA	NA
VERIFYING COMPLETENESS	NA	39.629 MEAN	NA	NA	NA
DOCUMENTING ANOMALIES	NA	104.42 MEAN	NA	NA	NA

The result of the requirements tracing analysis is that the tracing of the 5141 SRS requirements to 1173 Software Test Plan requirements (and vice versa) took approximately 260 man-hours. Figure 10 shows this graphically. Of these requirements, over 20% were found to be anomalous (SRS requirement was not fully verified by a test requirement or an SRS requirement was not addressed by any test requirements).

Based on these findings, it is estimated that the use of SFEP and SuperTrace allowed approximately 3 times more requirements to be processed per hour than manual methods. Also, improved accuracy was obtained using the automated methods. No attempt was made to measure improved accuracy. Using an estimate of $75/instruction for software development costs versus $4000/instructon for maintenance costs (as reported by TRW in [8]), it is estimated that the requirements tracing effort discussed in this article resulted in an approximate savings of several million dollars to the Government.

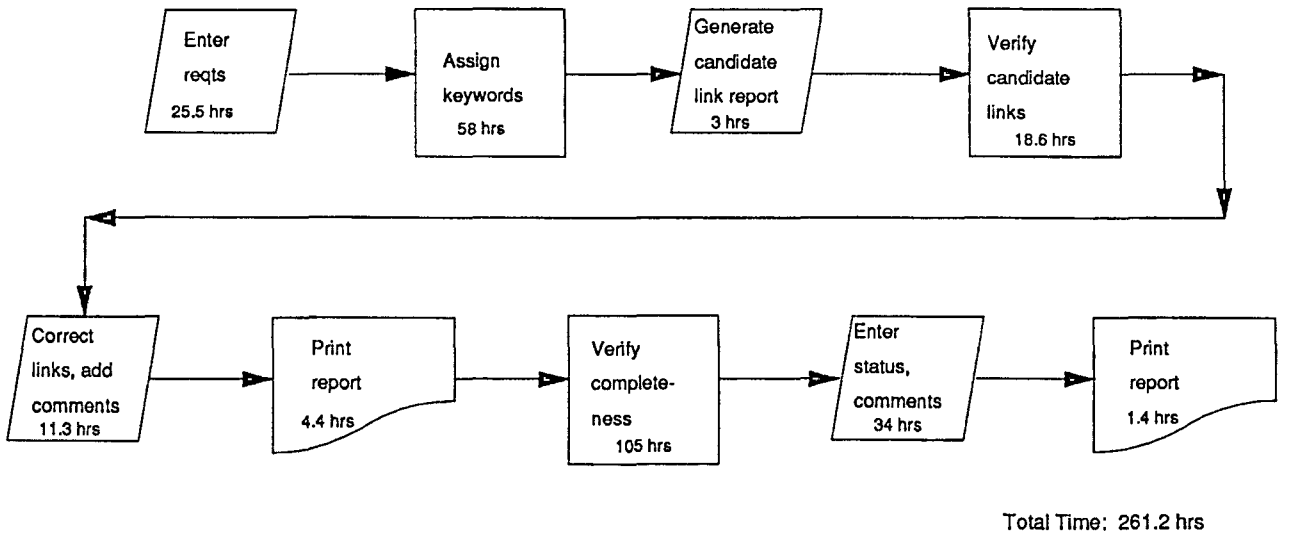


Fig. 10. Duration of Each Requirements Tracing Activity.

ieeefg10

REFERENCES

[1] Robert O. Lewis, "Software Verification and Validation," in Software Quality Management, Petrocelli Book, 1987, NY, NY, p. 240.

[2] V. Basili and D. Weiss, "Evaluation of a Software Requirements Document by Analysis of Change Data," in Fifth International Conference on Software Engineering, Computer Society Press of IEEE, 1981, Washington, D.C., pp. 314 - 323.

[3] Theater Mission Planning Center Upgrade System Specification, Cruise Missiles Project, U.S. Navy, February 1988, p. 1.

[4] Theater Mission Planning Center Upgrade Concept of Operations, Cruise Missiles Project, U.S. Navy, August 1989, p. 5-2.

[5] Richard B. Darlington and Patricia M. Carlson, Behavioral Statistics: Logic and Methods, The Free Press, New York, NY, 1987, p. 159.

[6] Bernard Rosner, Fundamentals of Biostatistics, Duxbury Press, Boston, Mass., 1988, pp. 383-409.

[7] Statistical Table for Biological, Agricultural, and Medical Research, Longman Group Limited, London, England, 1989.

[8] Barry W. Boehm, "Software Engineering," IEEE Transactions on Computers, 25, 12 (December 1976), p. 1236.

Paper 2-A-3

PRACTICAL EXPERIENCE IN IV & V: CASE STUDIES FROM ITALY

Dr. Mario Fusani
Consiglio Nazionale delle Richerch
Instituto de Elaborazione dell'Infromazione

Dr. Mario Fusani, born in 1945, graduated cum laude in Electronic Engineering at the University of Pisa, Italy, in 1971. Since 1971, he has been working at the Instituto di Elaborazione della Informazione of the Italian National Research Council (CNR), Pisa, involved in researches concerning operating and distributed systems. He has also worked as a research associate at the Dept. of Electrical Engineering and Computer Sciences of UC Berkeley. His current research interests include fault tolerant techniques and software validation.

A PRACTICAL EXPERIENCE IN IV&V: CASE STUDIES FROM ITALY

A. Bertolino, C. Carlesi, M. Fusani
IEI - CNR, Pisa, Italy

ABSTRACT:

Validation is usually defined as the process to establish whether or not (or how well) a system or computer program complies with its specified requirements. Only, that may be impossible to establish and the "specified requirements" themselves should often be validated against some other thing (user's secret will?) that is not there.

We have been working for a few years as an Independent Laboratory and research Institute being asked for this kind of service, and we think that the above definition does not catch just the point.

So, we are trying two opposite approaches.

The first one would impose that a system is specified using formal, algebraic techniques. They are based on the LOTOS language because we have the compiler and some tools about. This approach, however, seems to be good for research purposes, but not for providing a service: the number of states actually "explodes" when real cases have to be specified.

In the other approach we have applied a process in which a defined set of characteristics of both the software product and the developing project are submitted to a defined Validation Suite. This process ends, of course: a formal declaration about the actions performed and their results is issued, and this is the service.

The talk is intended to show what the Validation Suite is about, what is its value and what are the problems met in applying it to actual cases.

1. BASIC CONCEPTS ON IV&V:

Verification and Validation (V&V) of software technology products is acquiring increasing importance in the field of software engineering. In fact, V&V is a must to achieve product integrity.
V&V is usually defined [IEEE/729] as the process to establish whether or not (or how well) a system or computer program complies with its specified requirements. Only, that may be impossible to establish and the "specified requirements" themselves should often be validated against some other thing (user's secret will?) that is not there.

From the customer's viewpoint, V&V is addressed to demonstrate compliance with his/her needs; from the complementary manufacturer's viewpoint, V&V is the means to maintain (quality) control on software as it is developed (so that an "error-free", maintanable product is delivered). Independent V&V (IV&V) can be simply defined as V&V performed by an independent third party, i.e. by a team which is different and intermediate between customer and manufacturer.
IV&V is commonly employed when particular product complexity and criticity are involved. The reasons could be the following:

- An independent party is more successful in discovering defects (by performing inspections, for instance).

- An independent party is not interested in asserting the "good" results of the developing process.

- Some customers begin to realize that "good" software is not so easy to have at one's will. So, as they get aware not to have the necessary expertise, they wish to have the validity of a product (process, producer) checked by an independent organization.

- Performing IV&V is expensive (e.g., there is an extra-work, for the independent party getting acquainted with the object).

Incidentally, we observe that, in a sense, all the brand-new software products are critical, because they are known to contain defects.
However, IV&V can be applied with varying extensions, and this can be regulated according to the economy of the project.
Our thesis is that IV&V can be made flexible, easy and not too much expensive, so that it can be possibly applied to any class of software. To this purpose, we first recall some V&V schemes, then sketch the profile of our organization.

There are various levels of application of an independent V&V party [Deutsch88]. We illustrate 3 possible schemes in the following Figures 1 - 3.

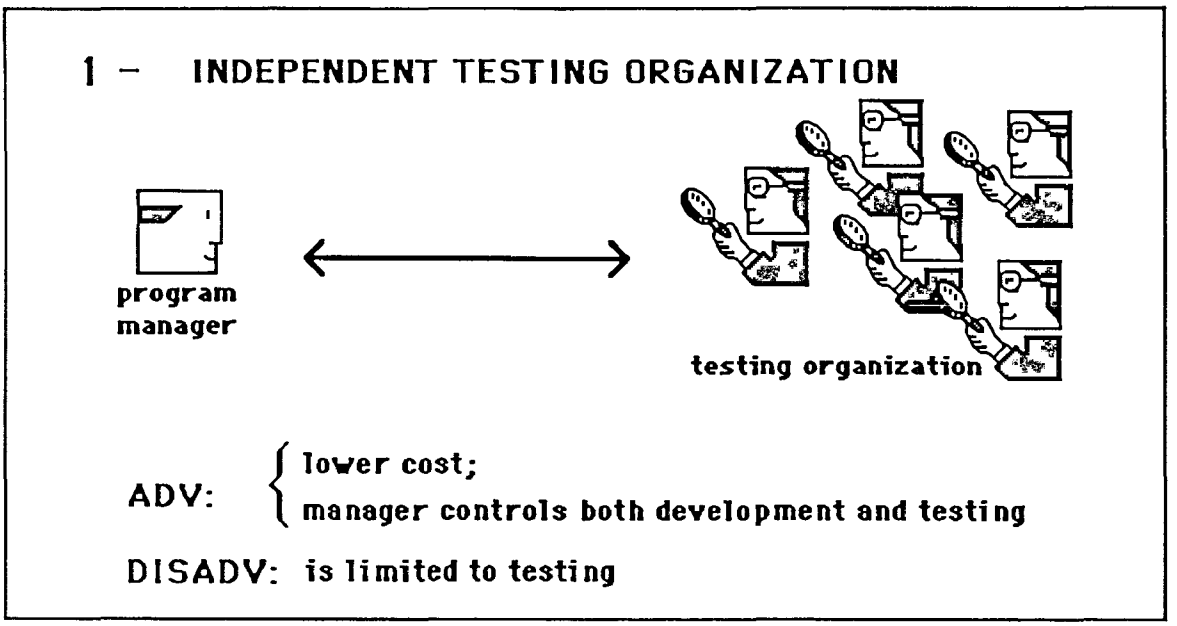


Figure 1.

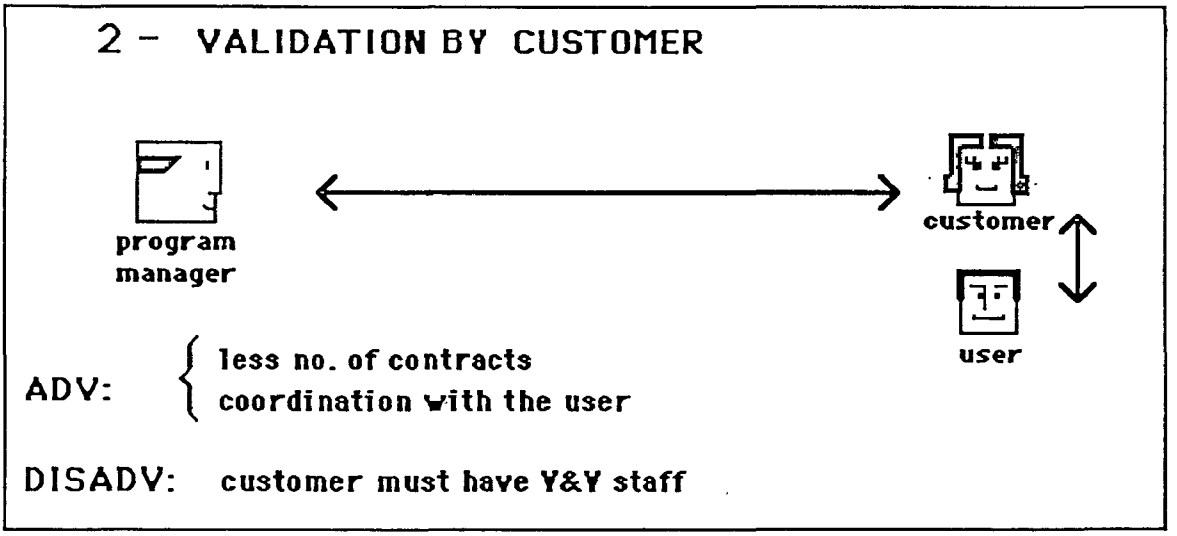


Figure 2.

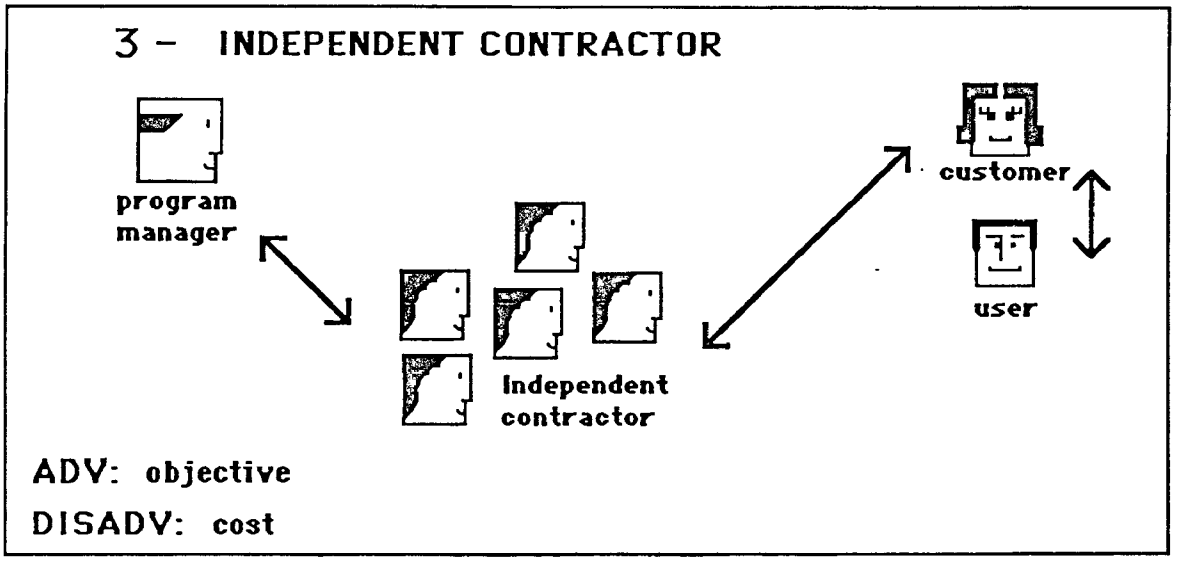


Figure 3.

We can be classified more or less in the scheme in the third figure. Besides, in some particular cases (the so called "Fiscal Meters" job), we extend our work to the issue of "certificates". In general, when the independent organization is officially authorized (i.e. accredited) to approve or refuse a submitted product for release, the process of IV&V terminates by issue of a certificate and more properly we speak of Certification.

We emphasize that this certificate, or however the successful termination of the V&V process, should by no means be mistaken as a "guarantee".

There is universal agreement on the urgency of applying V&V and possibly certification to software technology.

On the other side, the state of the art is still at a primitive stage. An activity has been started within the European Community to establish a certification scheme [Tretmans90]; the scheme is almost ready from the organizative point of view, but the technical contents have still to be defined. Also, as known, a lot of effort is devoted to establish appropriate standards.

Yet, the current situation is that some laboratories have already been accredited, by private (voluntary) organizations, as testing laboratories for certain classes of software products, about defined standards; yet no validation process has been formalized.

In this paper, we present our experience in IV&V; according to above premise, we had to conceive first a suitable process for IV&V. And, in fact, we started an activity just applying good-sense and the technologies of SQC.

To understand our experience, it is fundamental to describe first who we are, and this is what we shall treat in the next section. In fact, we could say that, due to the peculiarity of our organization, we represent an ideal interlocutor both for software customers and for software producers.

2. PROFILE OF OUR ORGANIZATION:

Our organization comes out from a *Public* research & service Institute working in Information Technology (IT), joined to an Independent *private* Testing Laboratory.

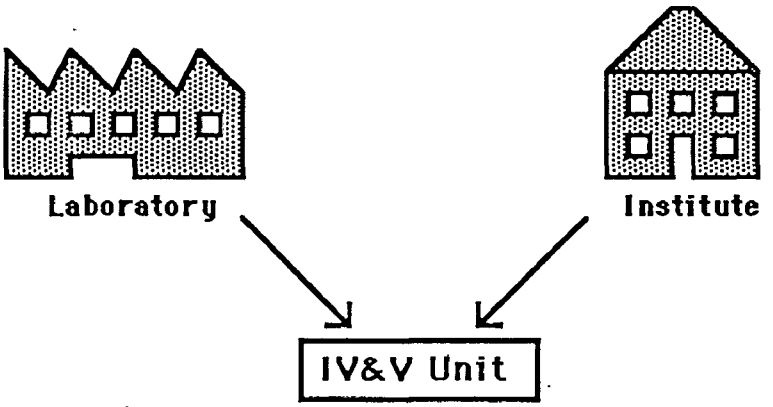


Figure 4.

Including a public-money financed entity would add some advantages with respect to the above presented models:

Adv:
- financial stability (the Institute can also choose more private labs as partners);
- bigger inventory of tools;
- easier access to project information;
- technical, state-of-the-art updated support to the service (EEC relationships and other international scientific connections)

and some disadvantages, too:

Disadv:
- complex organization;
- lack of flexibility;
- non specifically business-oriented.

Our efforts are to manage to get advantages by both organizations (a private company *must* be business oriented, for instance).

If we want to perform IV&V, we have to face many open problems, as sketched in Section1. We have set a number of long-term goals and of short-term goals.
In the first class, we include research objectives. In fact, as a research Institute, we have started some projects related to software V&V.

Particularly, a research project which attempts to put together formal verification and testing is described in Section 3. Another investigation effort is addressed to conceive testing tools in which the aspect of usability is stressed; this is sketched in Section 4.
In parallel, we had to face with spontaneous and urgent demand for validation service, for training and for consultation on SQC methodologies. Finding an effective way to answer this demand constitutes our short-term goal. Particularly, we conceived a Validation Suite, which is general and is fitted on the particular application. This shall be described in Section 5.

3. "FORMAL TESTING":

Being a research Institute helps in searching for solutions to problems we are facing with. We do that by trying to use results of the work of our researcher collegues.
One problem is incompleteness and ambiguity of the functional specifications of the programs to be validated. That often prevents to write good test plans. We started a research to investigate how much, if any, formal (algebraic) techniques help in deducing a test plan, starting from high-level specifications and refining them stepwise until we get to a specification level very close to the detailed design.
We use the LOTOS formal specification language, plus a LOTOS to C compiler and some tools to check equivalences between the various levels of specification (like observational and test equivalences). This work is related to an ESPRIT programme, called LOTOSPHERE [Tretmans90].
The work is still at the very beginng, and we are finding that, in the observational equivalence, some simple (but real, not just educational) cases yeld a low level specification with a big number of states (thousands), but still manageable by a mainframe computer. Testing equivalence (a testing process runs concurrently with a set of processes representing the application, synchronizes itself with it waiting for some expected results) could be better promising, but we don't have good results yet. Anyway, it is too early to say that formal methods are not yet feasible.

4. EASY-TO-USE TESTING TOOLS:

A sub-task of V&V is testing; since testing a not-banal software implies launching and recording a great number of test cases, automatic tools are used which take on themselves the clerical and repetitive part of the job. Many tools [Bertolino90] are today available.
More than about their functionality, we are interested into their usability: in a scenario of an independent test organization (such as ours),

penetrating the application under test with little effort is the key objective. Another objective we have in mind is to make software testing more and more simple.
Therefore, we are studing a tool which is provided with a suitable graphical interface. It performs a phase of reverse engineering on the program to be tested, so to alleviate the task of collecting information necessary to develop the test plan; then it illustrates the fundamental structure of the program, with regard to control flow and data flow. This project is still under development.

5. VALIDATION SUITE:

The validation suite was based on the state of the art for V&V.
As known, in order to validate a software product, the best approach is to check both the product and the process.
Essentially, we apply inspection on a required subset of documentation and testing on the executable product. Just this affirmation supposes that:

a) an adequate documentation is available
b) the product is testable.

This point coincides with the property of certifiability, i.e. IV&V has a cost; a product is certifiable when this cost is acceptable.
In the following table we summarize the Validation Suite.

VALIDATION SUITE

1 Request for software product

1.1 Probably exsisting documents

1.1.1 Functional specifications
1.1.2 Design schemata and flow graphs
1.1.3 Source code listings
1.1.4 Documents describing the developing environment of the product to be validated: Hardware, Operating System, Applications, Approaches, Tools.
1.1.5 Documents describing the actions performed during product development and testing
1.1.6 Reference standards for 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5 documents

1.2 Documents to be prepared

1.2.1 Missing documents from point 1.1
1.2.2 Filled Forms

1.3 Product instance

1.3.1 On a support suitable for available host hardware
1.3.2 With its target hardware

2 Acceptance analysis (Certifiability)

2.1 Adequacy criterion
2.2 Completeness criterion

3 Inspections

3.1 Document and Form inspection

3.1.1 Check lists for documents (various levels)

** REMARK: ** a set of documents may refer to an internal V&V (concerning both planning and results reporting). This is the case of an Independent V&V about an internal V&V activity.

3.1.2 Check lists for Forms (various levels)

3.2 Code inspection

3.2.1 Check lists for code

4 Tests (execution)

4.1 Functional tests

4.1.1 Unit tests
4.1.2 Subsystem, System tests

4.2 Structural tests

4.2.1 Branch Coverage

The Validation Suite would be a great deal of work if totally applied. In practice it represents all of the possible actions, of which only some are reasonably worth to perform in a single case of IV&V. The point is to devise each time the best subset to exercise, depending on product criticality, availability of documentation, cost constraints.

6. SOME FINAL REMARKS:

The first experience of our IV&V Unit has been a work (still in progress), performed upon request by the Government, to "certify" the software of the Fiscal Meters (cash registers storing the sales amounts into an unforgeable memory [Bertolino87] against a defined set of requirements.

The idea to put into the certificate just the results of the Validation Suite, as well as the idea of introducing the forms in the documentation, came from this experience.

One interesting fact about the follow-up of the certification activity is the following:

We are willing to act as an independent testing laboratory, so we have been offering such a service to the industrial community. What we get as demand from the customer's side is this: 'OK, that is very interesting, and we will be probably using your testing service, but now we have this big problem ..... '. And the problem is typically a need of assistance during the developing phase of the project, to check if the producer is actually giving what the user is willing.
From the manufacturer's side the situation is not much better: It is, in our opinion, even worse: almost everybody speaks of quality, but schedule urge and marketing needs often cause a product to be shipped without an adequate level of *functional* testing, not just structural.

REFERENCES:

[Bertolino90] Bertolino A, "An overview of Automated Software Testing", to appear on *The Journal of Systems and Software* , North-Holland.

[Bertolino87] Bertolino, A., Fusani, M., "Software Validation: A Government-imposed Challenge to the State-of-the-Art in Certification", *Computers & Standards*, Vol. 6, 1987, pp. 433-436.

[Deutsch88] Deutsch, M. S., Willis, R. R., *Software Quality Engineering: A Total Technical and Management Approach*, Prentice Hall, 1988.

[IEEE/729] ANSI/IEEE Std 729-1983, IEEE Standard Glossary of Software Engineering Terminology.

[M-IT-03/87] CEN, CENELEC, CEPT, Memorandum M-IT-03 on Certification of Information Technology Products, 1987.

[Tretmans90] Tretmans, J., "Test Case Derivation from LOTOS Specifications", LOTOSPHERE Workshop, Leidschendam, 3-4 May 1990.

Paper 3-T-1

COMPUTER AIDED TESTABILITY FOR SOFTWARE (SoftCAT)

Mr. Paul E. Janusz
Product Assurance Engineer
U.S. Army AMCCOM

Mr. Paul E. Janusz is an engineer in the Software Quality Assurance group of the Product Assurance Directorate, involved with research and development activities to develop tools and techniques which will assist the Independent Verification and Validation (IV & V) effort. These activities involve the development and implementation of automated tools for software development, testing, and maintenance, the incorporation of metrics to assess and quantify software preformance, and the application of artificial intelligence technology to solve common SQA problems. He recieved a B.S. in Civil Engineering fom SUNY at Buffalo in 1981 and a B.S. in Mathematics from SUC at Oneonta in 1980.

COMPUTER AIDED TESTABILITY OF SOFTWARE (SOFTCAT)

PAUL E. JANUSZ
US ARMY AMCCOM
(201)724-4849
DSN 880-4849

OUTLINE

- BACKGROUND
- SOFTCAT MODEL
- TEST HOOK PLACEMENT
- APPLICATION
- SUMMARY

SOFTCAT BACKGROUND

- SMALL BUSINESS INNOVATIVE RESEARCH (SBIR) CONTRACT

  ATAC, MOUNTAIN VIEW, CA
  BRAD ASHMORE, PRINCIPLE INVESTIGATOR

- DEVELOP A TEST HOOK DESIGN WHICH ANSWERS:
  - WHAT SHOULD A TEST HOOK CONSIST OF?
  - WHAT FEATURES ARE AVAILABLE TO BE INCLUDED IN THE TEST HOOK DESIGN CONCEPT?
  - WHERE SHOULD TEST HOOKS BE PLACED?
  - HOW CAN TEST HOOKS TIE INTO BUILT-IN TEST?
  - WHAT WILL BE THEIR IMPACT ON PROGRAM TIMING?
  - SHOULD THEY BE PART OF THE PRODUCTION VERSION OF THE SOFTWARE OR A TEST VERSION?

APPLICATIONS OVER LIFE CYCLE

- DESIGN
  - TESTABILITY FIGURES OF MERIT
  - SYMBOL TABLE AND OTHER REPORTS
  - TEST HOOK PLACEMENT GUIDANCE
- OPERATION
  - ERROR HANDLER
  - HIGH RESOLUTION DUMP
- MAINTENANCE
  - INTELLIGENT INTERACTIVE DEBUGGER

- WHILE SOFTCAT CAN BE USED THROUGHOUT THE LIFE CYCLE, ALL FEATURES CAN BE EFFECTIVELY APPLIED TO DEBUGGING DURING CODE DEVELOPMENT

CYCLOMATIC COMPLEXITY

- BASED ON PROGRAM LOGIC FLOW
- MODULE COMPLEXITY LIMIT OF 10
- RESTRUCTURE COMPLEX MODULES

SOFTCAT MODEL

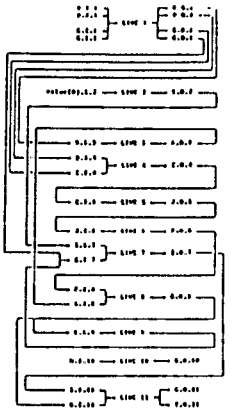
- BASED ON PROGRAM / DATA FLOW (PDF)
- INTRA-LINE PDF'S
  - FAULT PROPAGATION MODEL OF A SOURCE CODE LINE
  - REPRESENTS DEPENDENCIES WITHIN COMPLEX EXPRESSIONS
  - ONE LINE OF SOURCE CODE MAY CONTAIN MANY PDF'S
- INTER-LINE PDF'S
  - CONNECT TOGETHER THE MODELS OF SOURCE CODE LINES

SoftCAT model

- MODELS DATA FLOW SUBJECT TO CONSTRAINTS OF PROGRAM FLOW

```
 1   READ B, D, E, G
 2   I = 0
 3   A = B
 4   C = D + E
 5   J = C
 6   F = J
 7   I = I + 1
 8   G = G + A + F
 9   IF G < 100 THEN GO TO 7
10   B = H
11   WRITE G, I
```

SOURCE CODE



MODEL

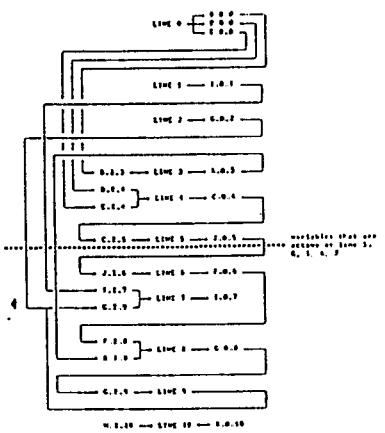
- PROGRAM FLOW MOVES FROM LINE 2 TO LINE 3 BUT DATA FLOW DOES NOT (BECAUSE VARIABLE I IS NOT IN LINE 3), THEREFORE, SoftCAT MODEL SHOWS NO CONNECTION FROM LINE 2 TO LINE 3
- SoftCAT MODEL IS A FAULT PROPAGATION MODEL BECAUSE IT CORRECTLY MODELS THE FLOW OF CORRUPTED DATA

Identification of active variables

O A VARIABLE IS ACTIVE AT A SOURCE CODE LINE IF IT IS AN OUTPUT OF AN UPSTREAM LINE AND AN INPUT OF A DOWNSTREAM LINE

- 'UPSTREAM' AND 'DOWNSTREAM' DEFINED IN TERMS OF PROGRAM FLOW ('DOWNSTREAM' MAY ACTUALLY BE ABOVE THE LINE IF A GO TO REDIRECTS PROGRAM FLOW)

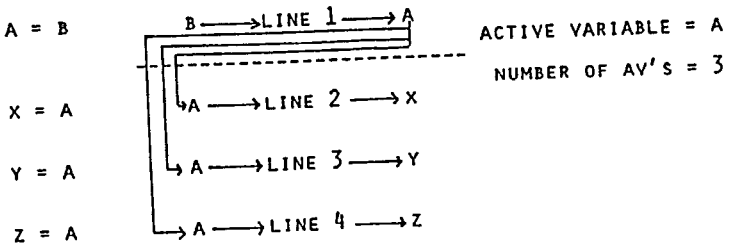
```
 0   READ B, D, E
 1   I = 0
 2   G = 0
 3   A = B
 4   C = D + E
 5   J = C
 6   F = LN (J)
 7   I = I + 1
 8   G = G + A + F
 9   IF G < 100 THEN GO TO 7
10   B = H
```



O ' ENTIFY ALL ACTIVE VARIABLES AT A LINE, THE SoftCAT MODEL IS TO REVEAL THE 'CROSS SECTION'

CROSS SECTION (CSEC) FUNCTION OUTPUTS

- LIST OF ACTIVE VARIABLES
- COUNT OF THE NUMBER OF VARIABLES ACTIVE AT A LINE
  MAY BE HIGHER THAN THE COUNT OF UNIQUE ACTIVE VARIABLES



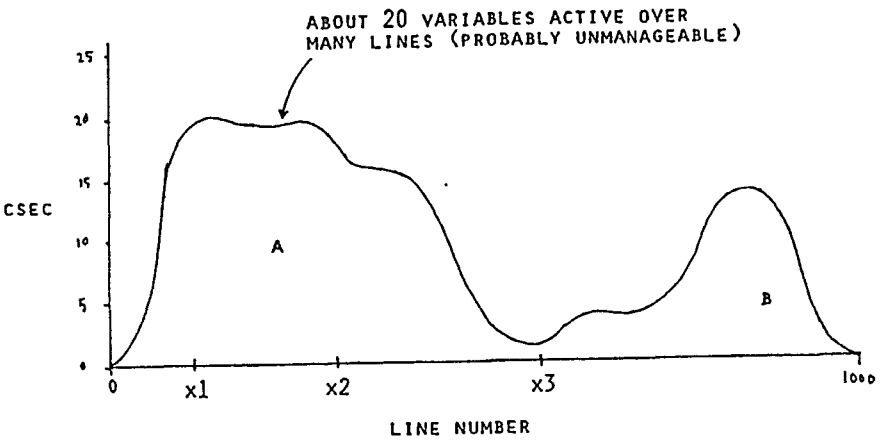
DESIGN STAGE APPLICATIONS

- TESTABILITY FIGURES OF MERIT
  - CSEC PROFILE
  - TEST PROGRAM COMPLEXITY PREDICTION
- SYMBOL TABLE AND OTHER REPORTS
  - LIST OF VARIABLES USED BUT NOT ASSIGNED (AND VICE VERSA)
  - LIST OF INTER-LINE PDF'S
- TEST HOOK PLACEMENT GUIDANCE
  - TEST HOOK LOCATION
  - DESIGN OF TESTING MACRO

TESTABILITY FIGURES OF MERIT (TFOM)

- CSEC PROFILE
  - A PLOT OF THE NUMBER OF ACTIVE VARIABLES AT EACH LINE IN THE PROGRAM
  - ILLUSTRATES THE DEGREE TO WHICH A SECTION OF CODE IS LOGICALLY DEPENDENT ON THE REST OF THE CODE
  - ILLUSTRATES THE 'DEPTH' OF LOGIC
  - IDENTIFIES OPTIMAL LOCATIONS FOR BREAKING A BLOCK OF CODE INTO SMALLER PIECES

CSEC PROFILE

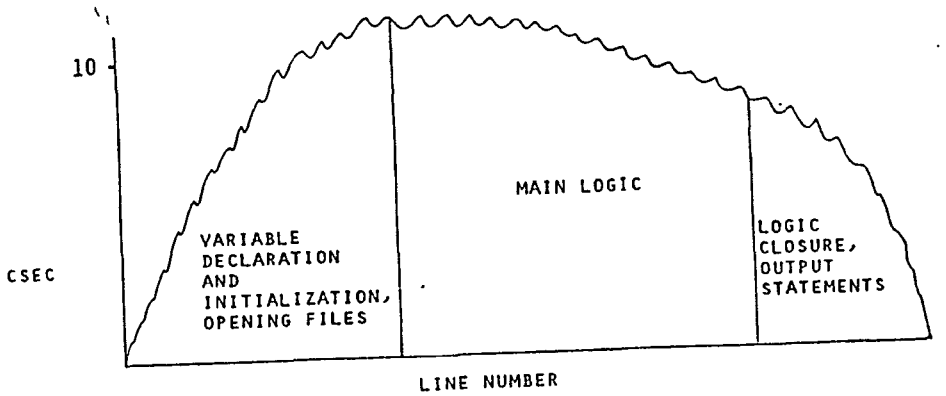


DEGREES OF LOGIC DEPENDENCE

- AT x3, CSEC = 1, THUS, ONE ACTIVE VARIABLE SEPARATES A FROM B AT LINE x3. THIS MEANS THAT B ONLY NEEDS 1 UPSTREAM VARIABLE, THEREFORE, B COULD BE A SUBROUTINE WITH THAT VARIABLE PASSED FROM A
- IF CSEC WERE ZERO AT x3, THEN B WOULD BE COMPLETELY INDEPENDENT OF A
- LOGIC AT x2 IS VERY DEPENDENT ON LOGIC AT x1

CSEC PROFILE

TYPICAL PROFILE OF A MANAGEABLE CODE



AREA UNDER THE CURVE IS SIGNIFICANT:

- HIGH CSEC IS PERMISSIBLE FOR A SMALL NUMBER OF LINES
- LOW CSEC IS PERMISSIBLE FOR A LARGE NUMBER OF LINES

TEST PROGRAM COMPLEXITY PREDICTION (TFOM)

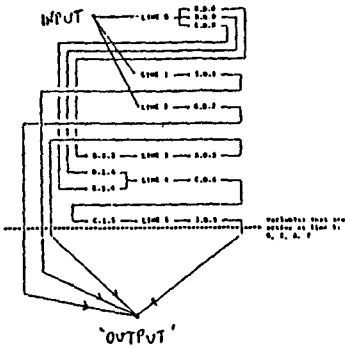
- COMPLEXITY = NUMBER OF BASIS PATHS THROUGH SOFTCAT MODEL

  DIFFICULTY OF TESTING A SEGMENT OF CODE

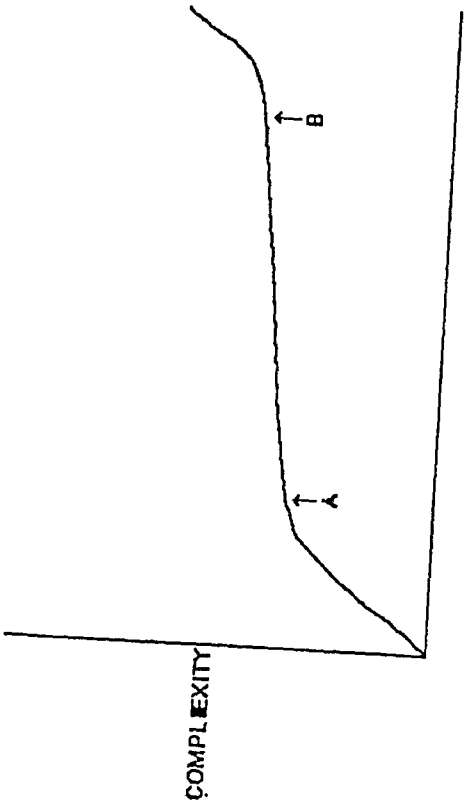
- CYCLOMATIC COMPLEXITY METRIC IS USED, BUT APPLIED TO SOFTCAT MODEL
  - MUST BE SCALED DIFFERENTLY BECAUSE IT'S MEASURING A DIFFERENT KIND OF COMPLEXITY
  - COMPLEXITY OF A REASONABLE TEST SHOULDN'T EXCEED ABOUT 100 (THIS IS A CURRENT ESTIMATE, IT WILL CHANGE AS WE TRY DIFFERENT TEST CASES)
- PROFILE GENERATED TO SHOW COMPLEXITY OF TESTING EACH LINE

TEST PROGRAM COMPLEXITY PREDICTION

- O METHOD FOR CALCULATING THIS TFOM USES CSEC
  - ACTIVE VARIABLES ARE IDENTIFIED AT LINE
  - SUBGRAPH EXTENDS FROM PROGRAM INPUTS, THROUGH THE MODEL, AND TO A NODE THAT THE ACTIVE VARIABLES ARE CONNECTED TO



- O COMPLEXITY (C = E - N + 2) IS CALCULATED FOR SUBGRAPH



Complexity plot

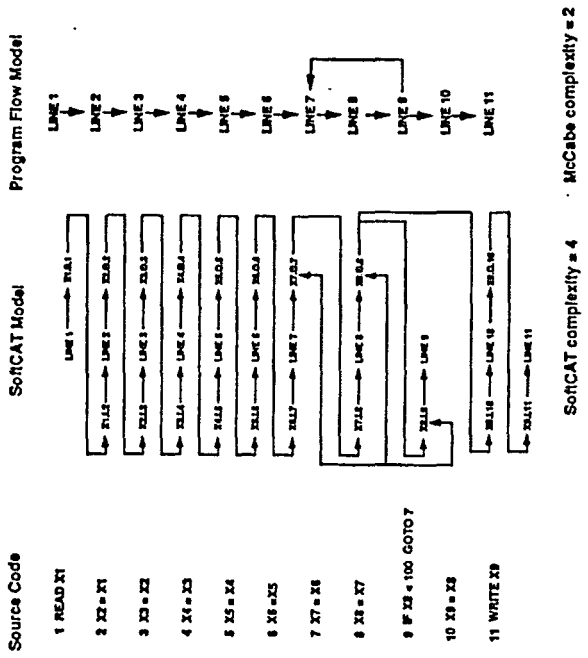


Figure 9a.

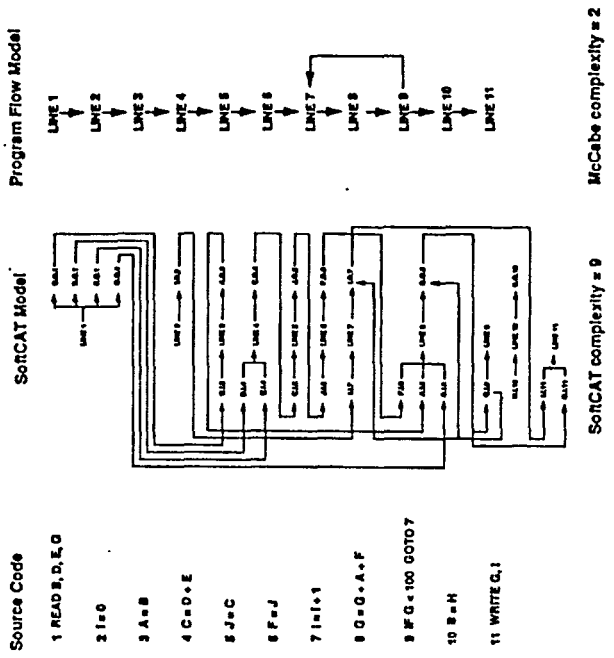


Figure 9b.

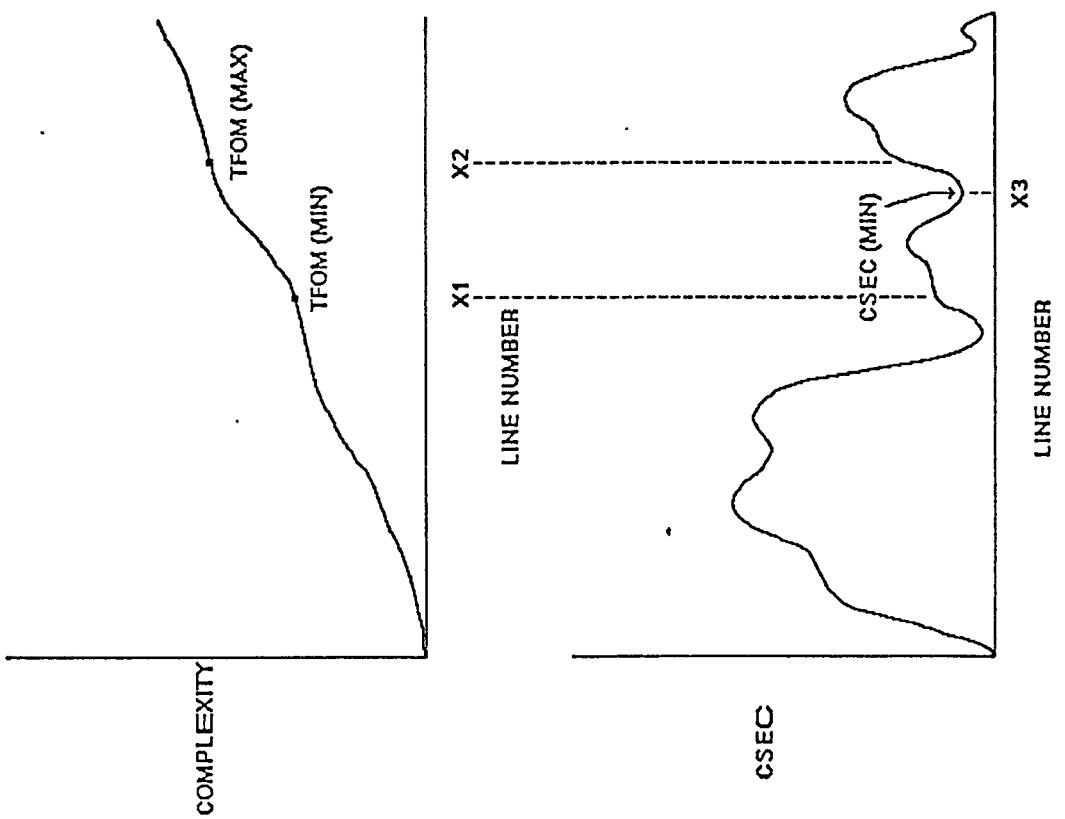
TEST HOOK PLACEMENT GUIDANCE

- A TEST HOOK IS A MACRO DESIGNED TO ASSESS THE VALIDITY OF ALL ACTIVE VARIABLES AT A LINE
- AUTOMATIC PLACEMENT STRATEGY IS BASED ON CSEC AND TEST COMPLEXITY PROFILES
- SOFTCAT FINDS WHERE TO PLACE TEST HOOKS AND WHAT VARIABLES TO TEST. USER DEFINES THE MACRO
- USER SPECIFIES A RANGE OF MAXIMUM PERMISSIBLE TEST COMPLEXITY VALUES

  - DEFAULT TFOM(MIN) = 75, TFOM(MAX) = 100

TEST HOOK PLACEMENT GUIDANCE

- SOFTCAT STARTS AT LINE 1 AND MOVES DOWN THE CODE UNTIL IT FINDS THE LINE CORRESPONDING TO TFOM(MIN)
- CSEC IS STORED STARTING AT THIS LINE
- WHEN THE LINE CORRESPONDING TO TFOM(MAX) IS REACHED, SEARCH STOPS
- FOR THE CSEC VALUES STORED, THE LINE WHERE THE MINIMUM VALUE OCCURS IS THE RECOMMENDED TEST HOOK SITE
- SEARCH RESUMES FROM THAT LINE
- EFFECT IS TO PLACE TEST HOOK SUCH THAT TEST COMPLEXITY IS NEAR ITS MAXIMUM REASONABLE VALUE BUT REQUIRES WRITING A MACRO WITH THE FEWEST POSSIBLE VARIABLES



Test hook placement guidance

TEST HOOK PLACEMENT GUIDANCE

- SOFTCAT GENERATES A REPORT INDICATING WHERE TO PUT TEST HOOK
- USER DEFINES MACRO
  - COULD BE A SUBROUTINE OR JUST AN IF-STATEMENT
  - ALL THE ACTIVE VARIABLES MUST APPEAR IN MACRO

    NO OTHER VARIABLES MAY APPEAR

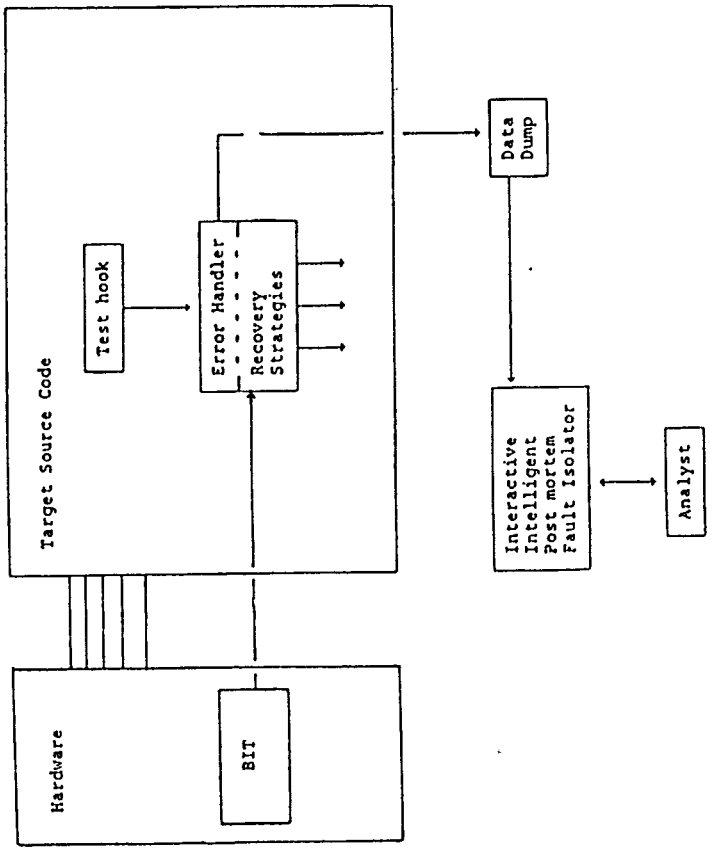
    VALID MACROS INCLUDE ANY RELATION BETWEEN THE ACTIVE VARIABLES

TRAIL REPORTING STATEMENTS

- SPECIAL STATEMENTS AUTOMATICALLY PLACED AT EACH BRANCH POINT
  - PERMITS RECONSTRUCTION OF SEQUENCE OF LAST N LINES EXECUTED BEFORE CRASH
  - FIFO STACK IS MAINTAINED TO PRESERVE SEQUENCE THAT BRANCH POINTS WERE TRAVERSED: LOW OVERHEAD

SOFTCAT REAL-TIME ERROR HANDLER

- O PERMITS FOR RECOVERY OR GRACEFUL SHUTDOWN
- O USER DESIGNS SPECIFIC RECOVERY STRATEGY INVOKED BY VIOLATION OF A GIVEN TEST MACRO
- O PERMITS INTEGRATION WITH HARDWARE BIT



INTEGRATION WITH HW BIT

- DURING OPERATION, A FAILURE MAY BE DETECTED IN HW BY HW BIT
- EFFECTS OF THE HW FAULT COULD PROPAGATE TO SW THROUGH I/O PORTS

INTEGRATION WITH HW BIT

- SPECIAL SOFTCAT MODULE RESIDING WITHIN SW CAN RECEIVE AN INTERRUPT FROM HW BIT WHEN FAULT IS DETECTED
  - HW BIT TRANSMITS DATA THAT SOFTCAT MODULE USES TO DEDUCE WHICH I/O PORTS MAY START RECEIVING CORRUPTED DATA
  - SOFTCAT ENABLES ALL TEST HOOKS DOWNSTREAM OF THE SOFTWARE SIDE OF THE AFFECTED PORT(S)
  - THIS PUTS THE TEST HOOKS ON 'FULL ALERT' AND WILL SLOW DOWN EXECUTION
  - A MORE RADICAL OPTION WOULD SHUT DOWN THE SYSTEM, BUT PENALTY FOR FALSE ALARM WOULD THEN BE GREATER
- SW FAULT PROPAGATING TO HW IS ALSO POSSIBLE
  - THIS IS ADDRESSED BY THE SW ERROR HANDLER

HIGH RESOLUTION DUMP

- USED BY THE INTELLIGENT DEBUGGER TO FACILITATE FAULT ISOLATION
- VERY COMPACT DATA DUMP WHICH CAN BE WRITTEN TO DISK CONTAINS:
  - LOCATION OF THE VIOLATED MACRO AND VALUES OF ALL MACRO VARIABLES
  - LIST OF LAST N BRANCH POINTS REACHED BEFORE ERROR DETECTION (N CAN EASILY BE SEVERAL HUNDRED)

MAINTENANCE STAGE APPLICATIONS

INTERACTIVE POST MORTEM DEBUGGER

- PORTION OF SOFTCAT THAT WILL PROBABLY GET USED MOST
- USES SOURCE CODE AND HIGH RESOLUTION DUMP
- HAS SPECIAL 'REPLAY' FEATURE
  - CAN STEP THROUGH THE LAST N LINES ONE AT A TIME OR AUTOMATICALLY
  - VALUES OF VARIABLES ON PREVIOUS LINES WILL BE DEDUCED WHERE POSSIBLE

SUMMARY

- A NEW MODELING AND ANALYSIS METHOD WAS DEVELOPED WHICH IS SENSITIVE TO BOTH PROGRAM FLOW AND DATA FLOW
  - CORRECTLY CAPTURES SOFTWARE FAULT PROPAGATION
  - DEPICTS DEGREE OF LOGIC COUPLING WITHIN PROGRAM
- TEST HOOK DESIGN AND PLACEMENT METHODOLOGY DEVELOPED
- STRATEGY FOR INTEGRATING WITH HW BIT DEVELOPED

Paper 3-T-2

TEST ENGINEERING TECHNOLOGIES FOR REAL-TIME APPLICATIONS

Mr. Robert Troy
Verilog, Inc.

Mr. Robert Troy is President and CEO of Verilog S.A. since 1984. He is also President of Verilog USA since 1986. He has a Doctorate in Computer Science from Paul Sabatier University in Toulouse, France. He is a member of IEEE, AFCET, and AFCIQ. His scientific publications deal with Computer Simulation, Reliability Evaluation, Quality Assurance, and Fault Tolerant Computing.

VERILOG

BEAUREGARD SQUARE, SUITE 300
6303 LITTLE RIVER TURNPIKE
ALEXANDRIA, VA 22312

TEST TECHNOLOGIES FOR REAL TIME APPLICATIONS

Robert TROY

VERILOG *SOFTWARE DEVELOPMENT PRACTICES*

RATP – METRO automatic pilot

THE TRAIN DOORS CAN ONLY OPEN WHEN THE TRAIN HAS COME TO A COMPLETE HALT.

ROCKWELL COLLINS – aircraft equipment

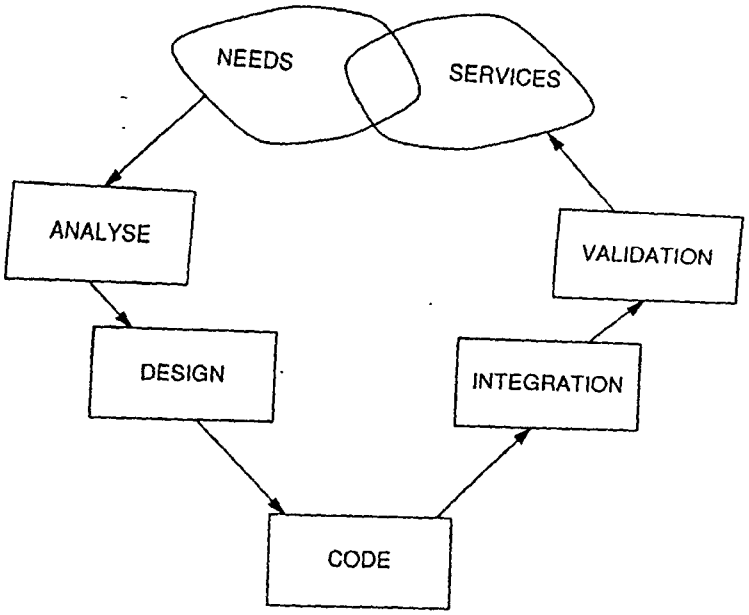
WHEN A PILOT PROGRAMS TRANSMISSION CHANNEL FREQUENCIES, THE OTHER PILOT'S EQUIPMENT COMMANDS MUST NOT INTERFERE WITH THAT PROGRAMMING ... PROGRAMMING CANNOT BE MONOPOLIZED BY ONE OF THE PILOTS ... THE PILOT INSTRUCTOR HAS AN INHIBIT COMMAND AT HIS DISPOSAL ...

AEROSPATIALE – break system control unit AIRBUS A320

AUTOMATIC MODE MUST BE DISENGAGED WHEN THE PILOT ACTIVATES THE BREAK PEDAL ...

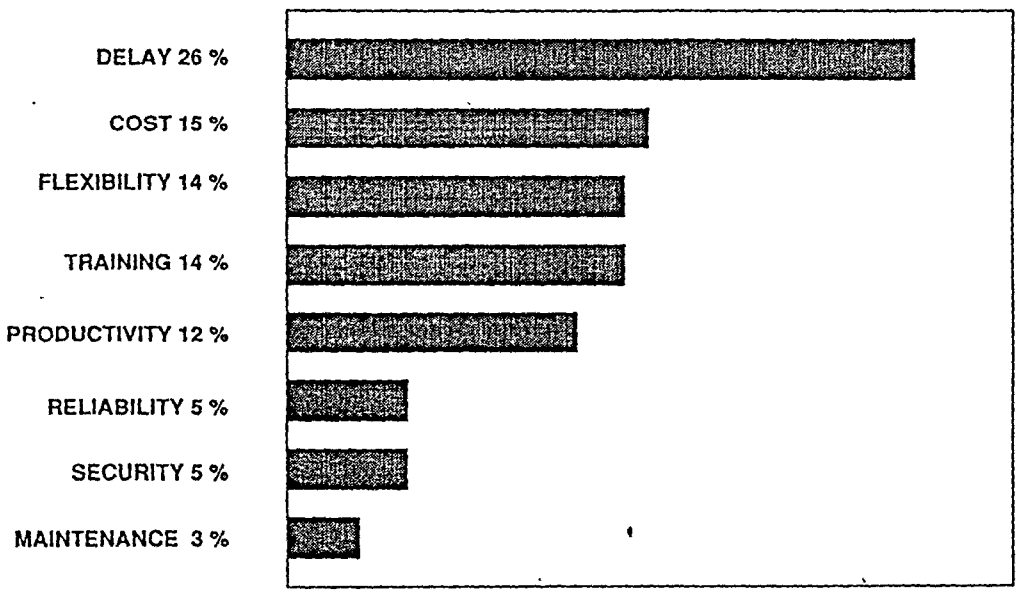
VERILOG

SOFTWARE DEVELOPMENT PRACTICES



VERILOG

SOFTWARE DEVELOPMENT PROBLEMS



MAIN CAUSE

WEAKNESS OF TEST POLICY

as a consequence of

PROGRAMMING ENVIRONMENT PRIORITY (limited)

THE ESSENTIAL QUESTIONS ARE :

ON THE PRODUCT ⟶ FIT WITH NEEDS ?

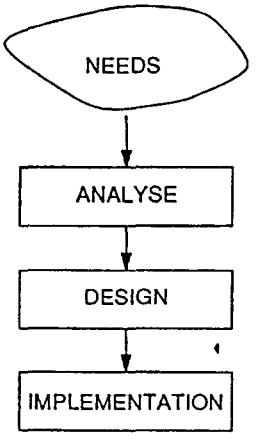
ON THE PROCESS ⟶ MASTERING ?

REMEDY

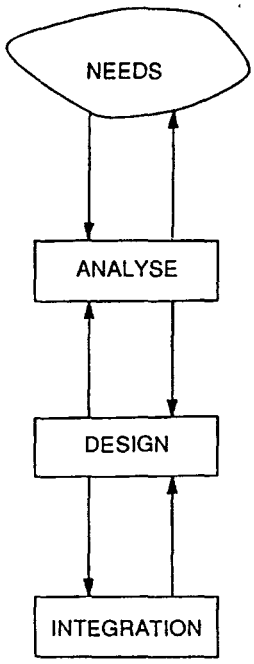
STRONG TEST POLICY

impose

CONSTRUCTIVE PRODUCTION METHOD

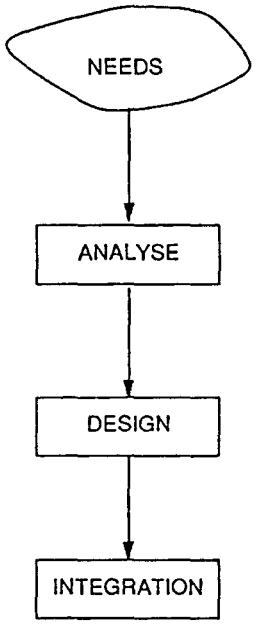


VERILOG — REAL TIME CHARACTERISTICS



- INTERACTIONS between Application and Environnement
- CONSTRAINTS Specification : Time, Performances, Dependability
- REFINED CONSTRAINTS
- COMMUNICATION schemes : Protocols, Asynchronisms
- REAL TIME FEATURES in programming languages

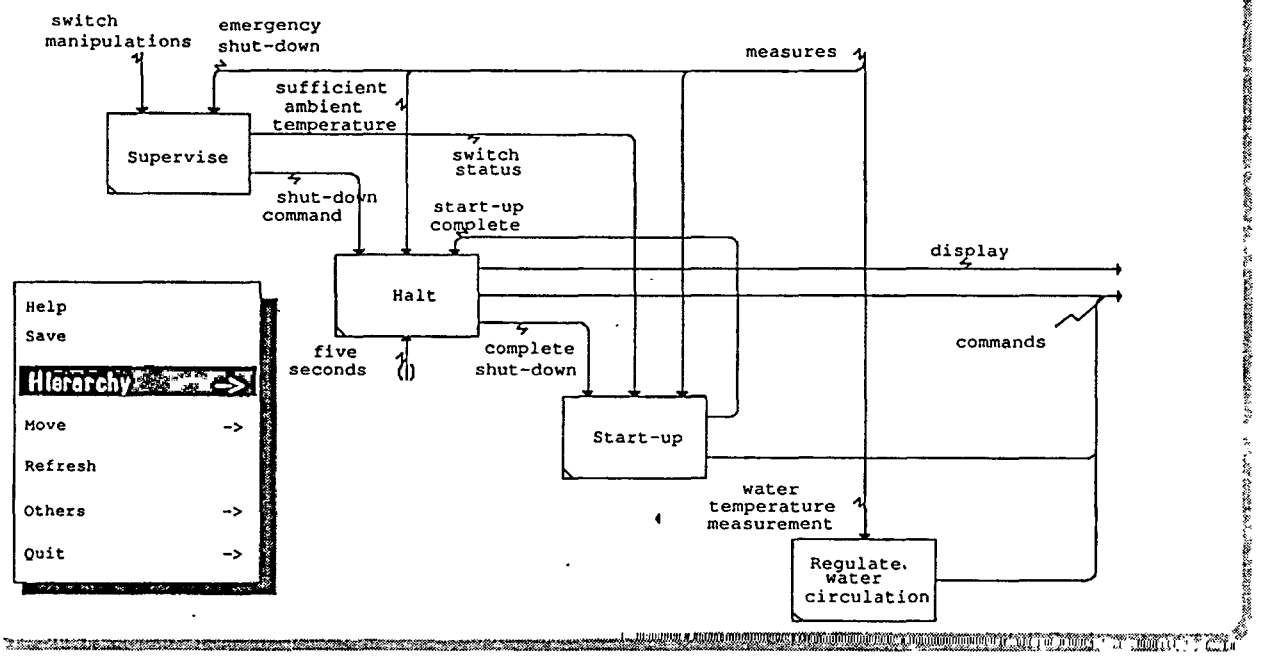
VERILOG — TEST POLICY – CONSEQUENCE



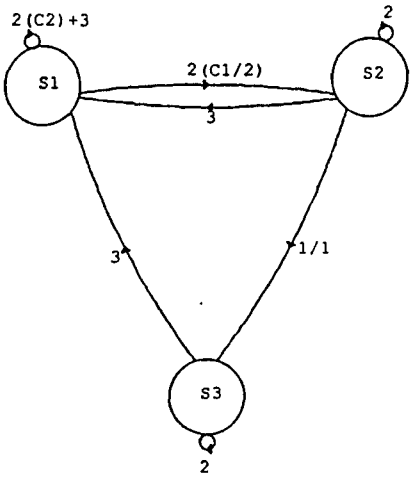
- ENFORCES FORMAL SPECIFICATION AND DESIGN
- REQUEST ADEQUATE TEST TECHNOLOGY

ASA

LANGUAGE AND EDITOR	for	Requirement Capture
INTERACTIVE SIMULATOR	for	Testing the Specification vs Needs
TEST GENERATOR	for	Validating Implementation vs Specification
ANNOTATION	for	Constraints Specification

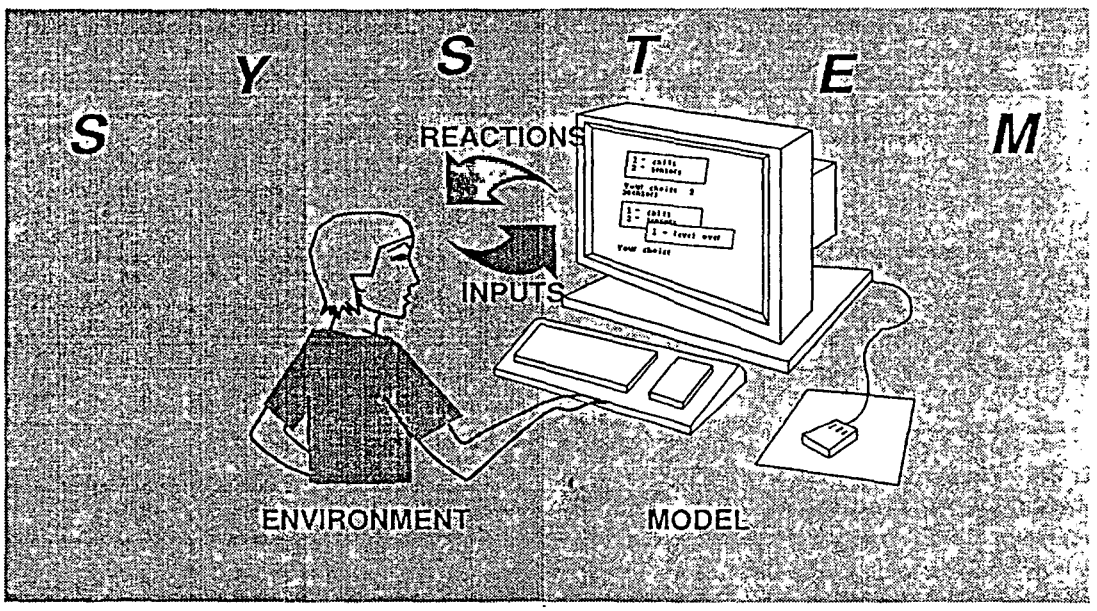


ASA state machine



message \ state	S1 off		S2 waiting	S3 on
Me1 adequ..speed			A1/S3	
Me2 inadequate ..temperature	C1	A2/S2	v/.	v/.
	C2	v/.		
Me3 syste..t off	v/.		v/S1	v/S1

ASA simulation



VERILOG — ASA test generation

```
LSATEST>    request

*****************************************
*       Current request commands        *
*****************************************
*       1  target -st M232 localised    *
*       2  fix -mo train_type - mi      *
*       3  forbid -tr M232 tr60         *
*****************************************
```

STRATEGY DEFINITION

SIMULATION

```
LSATEST>    simu

|   1  train features
|   2  train movement
|   3  train control
|   4  driver orders
|   5  ground environment messages
|   6  computer reset
|   7  computer cycle
|   8  cancellation data

Your choice>    6
>computer reset
.. m16 <computer reset
? /train_type ()

|   1  ok
|   2  nok

Your choice>
```


VERILOG — ASA test generation

```
LSATEST>    g -c

***                                          ***
***     G E N E R A T I O N   P R O C E S S I N G   ***
***                                          ***
+++     2  result scenarios     -   total 2      +++
+++     6  error scenarios      -   total 6      +++
+++     1  result scenarios     -   total 3      +++
+++     4  error scenarios      -   total 10     +++
+++     2  result scenarios     -   total 5      +++
+++     5  result scenarios     -   total 10     +++

***                                          ***
***     G E N E R A T I O N   R E S U L T S   ***
***                                          ***
***         10  result scenarios             ***
***                                          ***
***         10  error scenarios               ***
***                                          ***

        * CAUTION *

        Result scenarios are numbered from 1 to 10
        Error scenarios are numbered from 11 to 20

        Default : Analysis commands bear on correct scenarios

| Number of scenarios  :  20   |
| average length       :  9.8  |
| shortest             :  8    |
| longest              :  13   |
```

RESEARCH OF SCENARIOS

```
LSATEST>   scenario

 ____________________________
|                            |
|      RESULT SCENARIOS      |
|____________________________|

***     START OF SCENARIO 1   ***

    \invariant modification (disabled)
... M232 #  off

    \train_commands\ ... \car indicator (light off)
    \message_to_environment\compatibility required (true)


 >computer reset
... M211: \train_type(mi)

    M211: train_type verify(mi) = ok
    \message_to_environment\compatibility required(true)
    \train_commands\ ... \supplying state(light off)
    \message_to_environment\compatibility required(ture)

... M232 #  off
    \invariant_modif(disabled)
    M212: proms control = ok

    \train_commands\passive command
    \train_commands\ ... \car indicator (light off)
    \train_commands\ ... \feeding state (light off)
    \train_commands\ ... \initialization channel
```

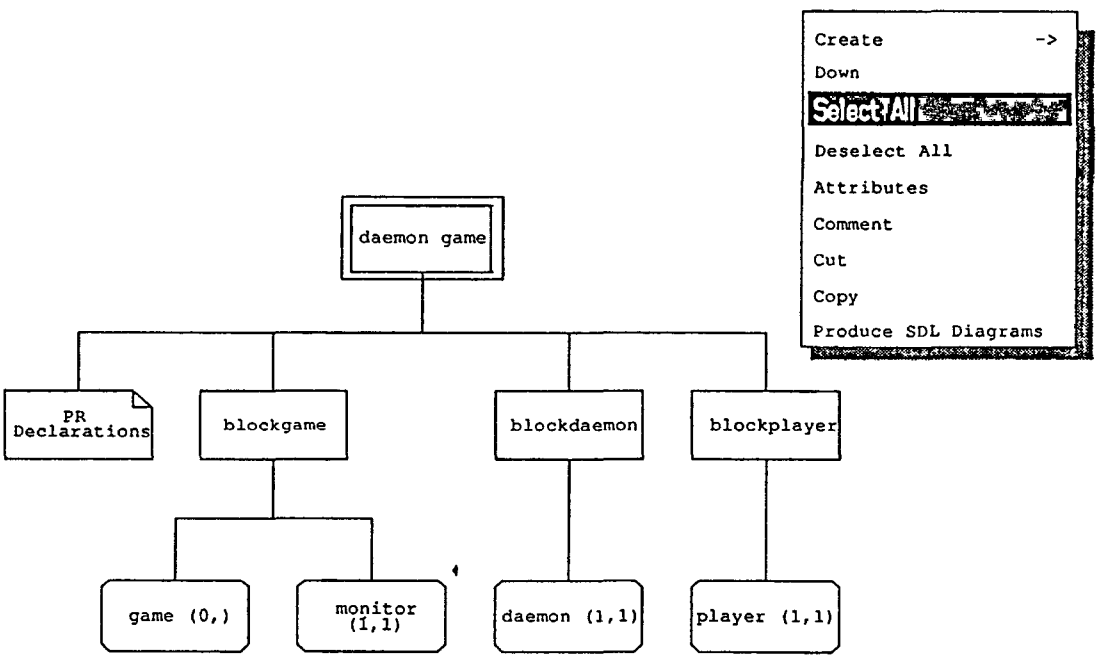
SCENARIOS PRODUCED


```
ANALYTSIS> coverage

 ____________________________
|                            |
|      MODULES COVERAGE      |
|____________________________|

              :
              :
              :

M2234: check locking
    State Coverage          :   76 %
    Transition Coverage     :   16 %
    Action Blocks Coverage  :   16 %
```

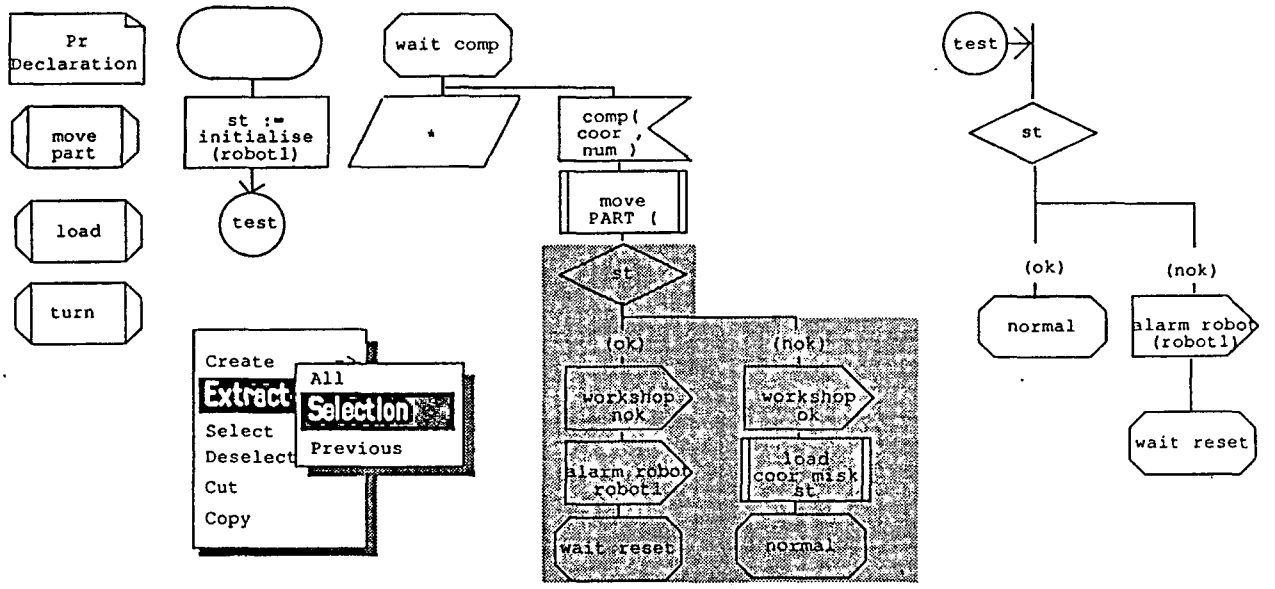
COVERAGE RATE OF THE SCENARIOS

GEODE

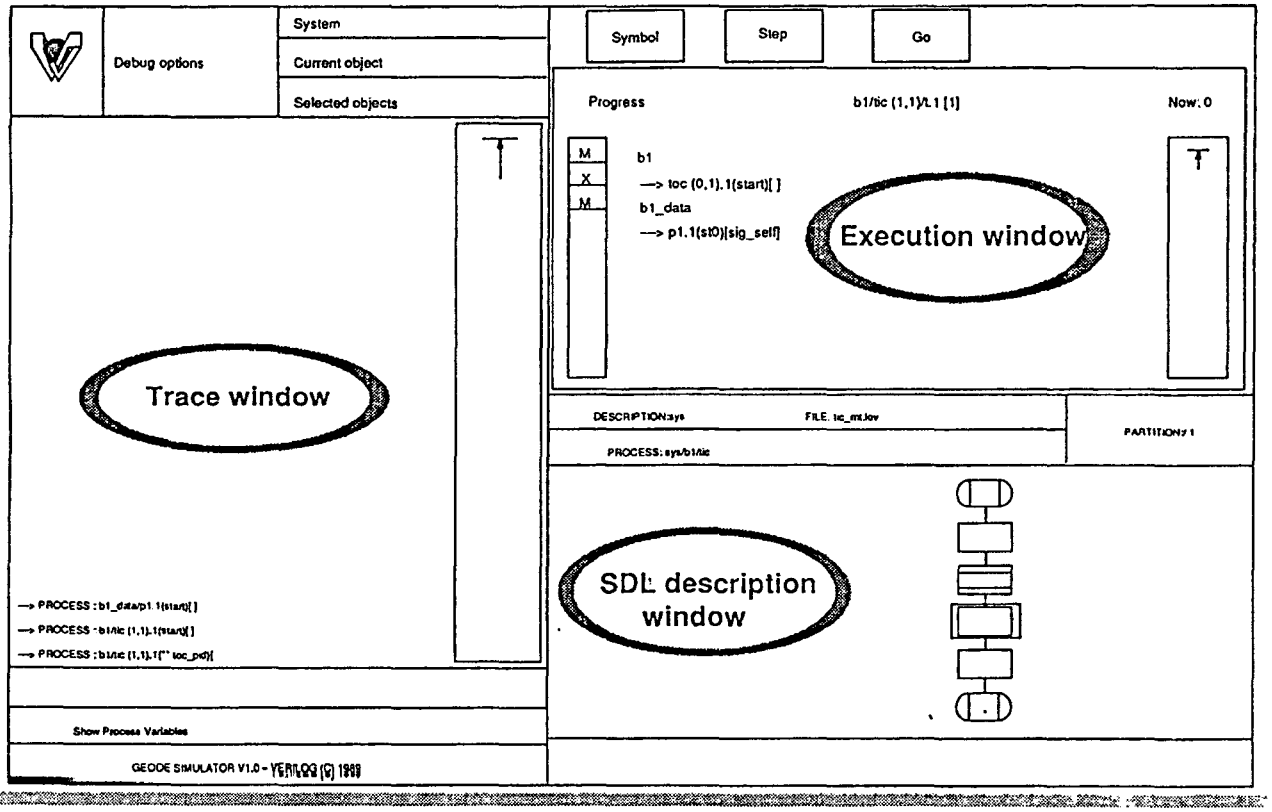
LANGUAGE AND EDITOR	for	Design Expression
INTERACTIVE SIMULATOR TRACEABILITY	for	Testing Design vs Specification
PETRI NETS	for	Refining and Checking the Constraints

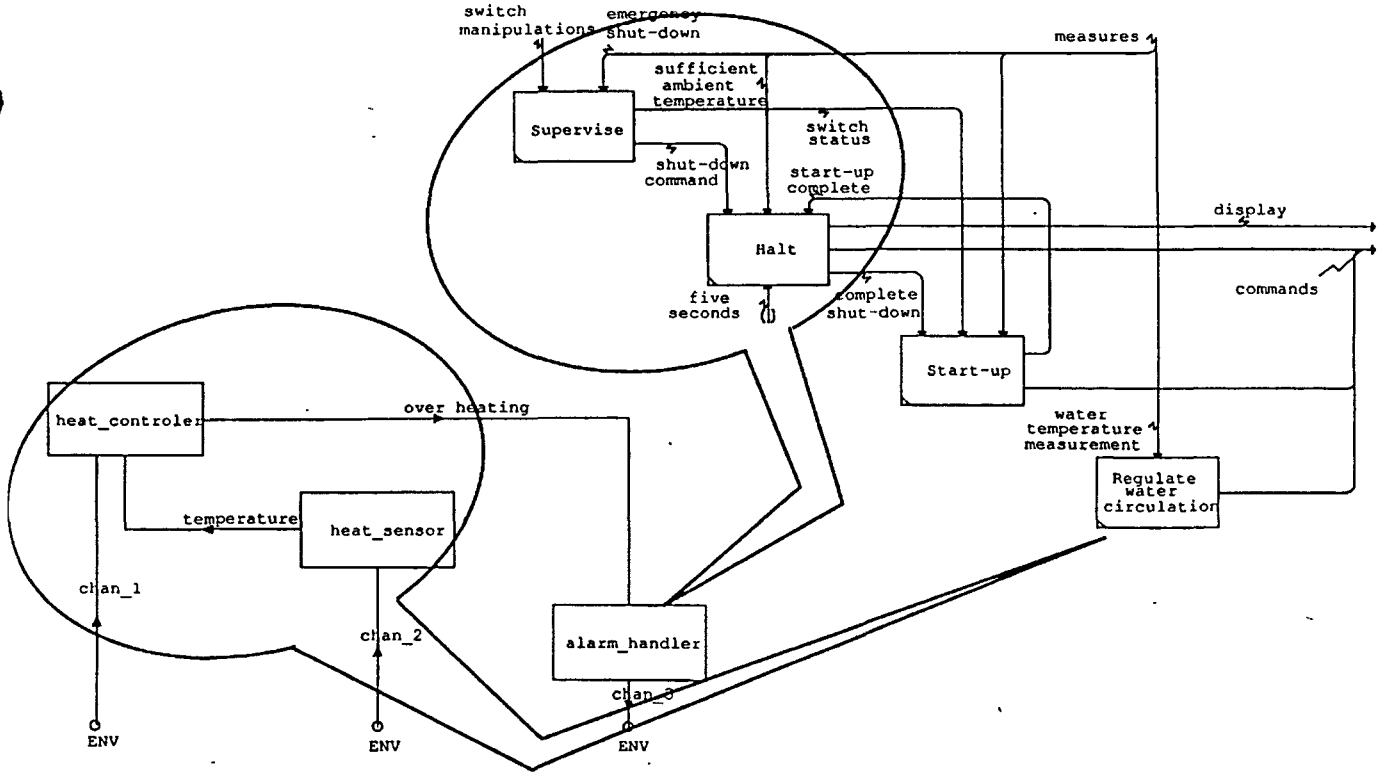


GEODE state machine



GEODE simulation



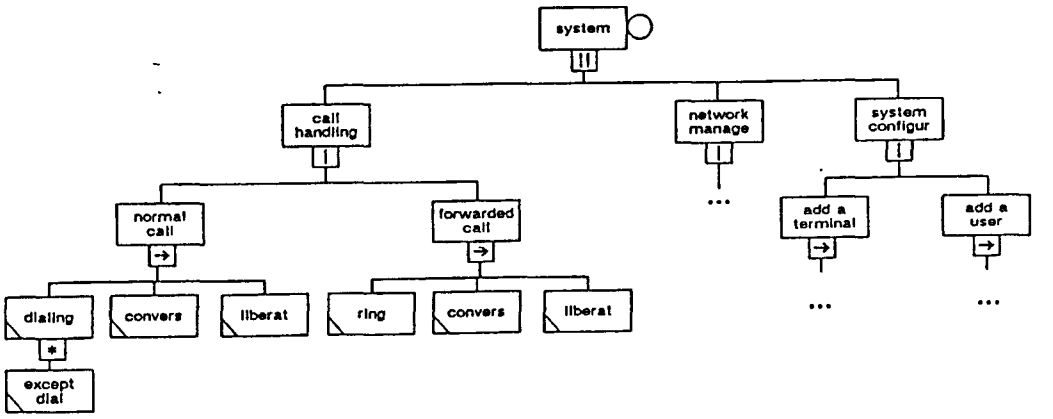


ORIGIN \ TARGET	M21	M22	M23	Compl
C1 : machining shop...	X		X	X
C2 : turn table.....				!
C3 : assembly shop....		X	X	X

M21 : machine A.....
M22 : assemble A' and B..
M23 : diagnosis.....

VERILOG

MSC : SCENARIOS COMPOSITION

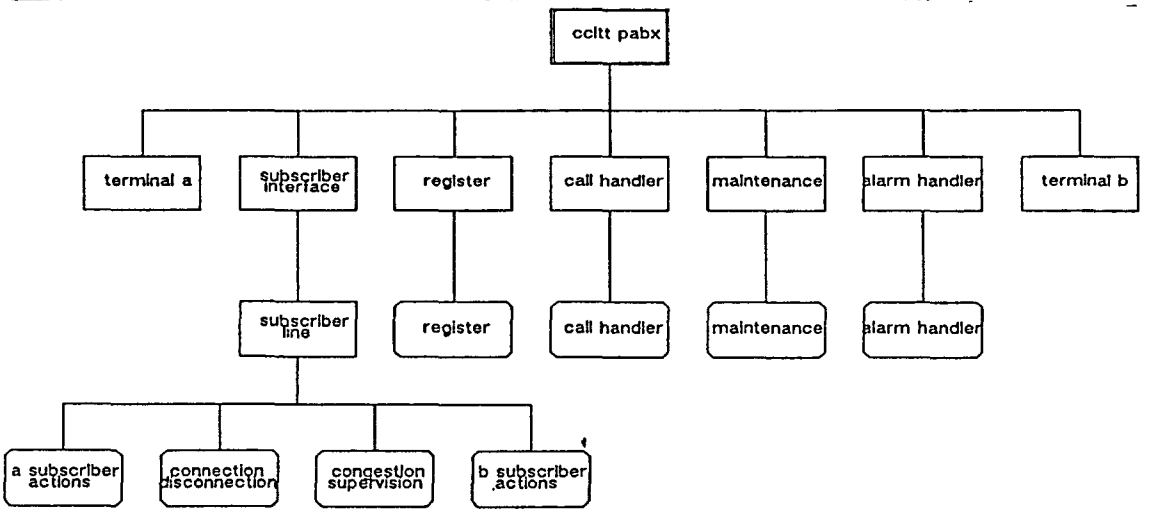


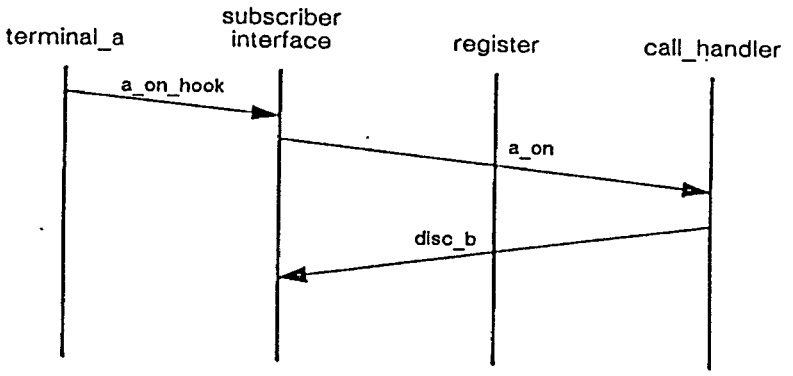
Legend:

- = is
- → and
- || parallel
- | or
- . exception
- repeat

VERILOG

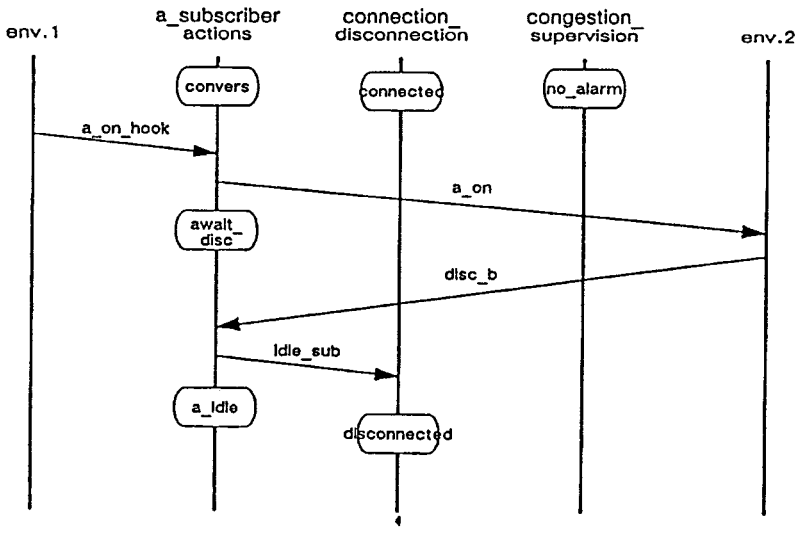
MSC : SYSTEM HIERARCHY



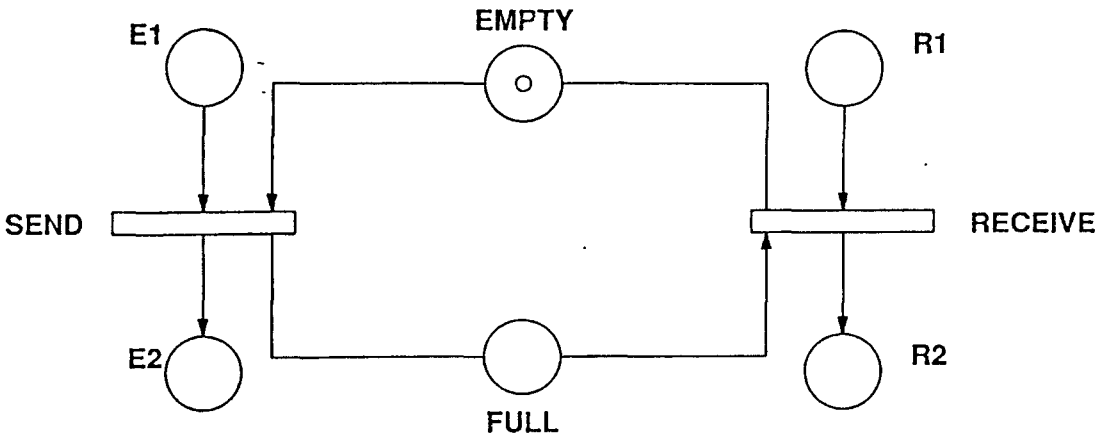


MSC belonging to "normal call / liberation" Scenario projected inside "ccitt_pabx" entity

VERILOG MESSAGE SEQUENCE CHARTS

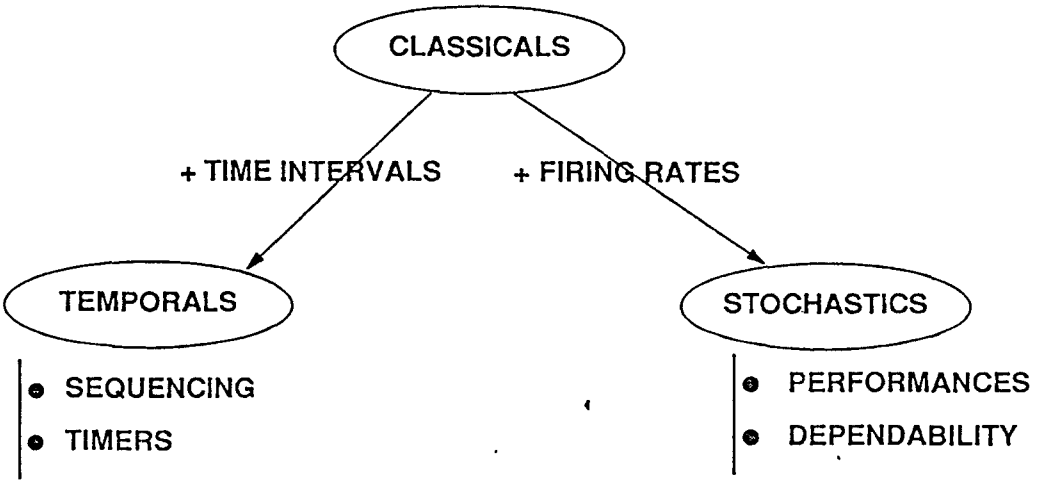


MSC belonging to "normal call / liberation" Scenario projected inside "subscriber_line" entity

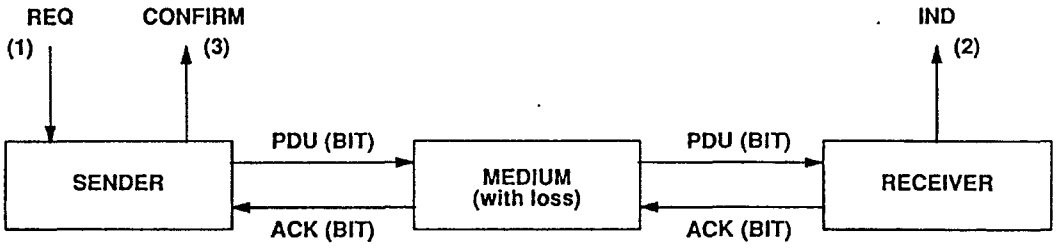


LOGICAL PROPERTIES

SYNCHRONISATION MECANISMS



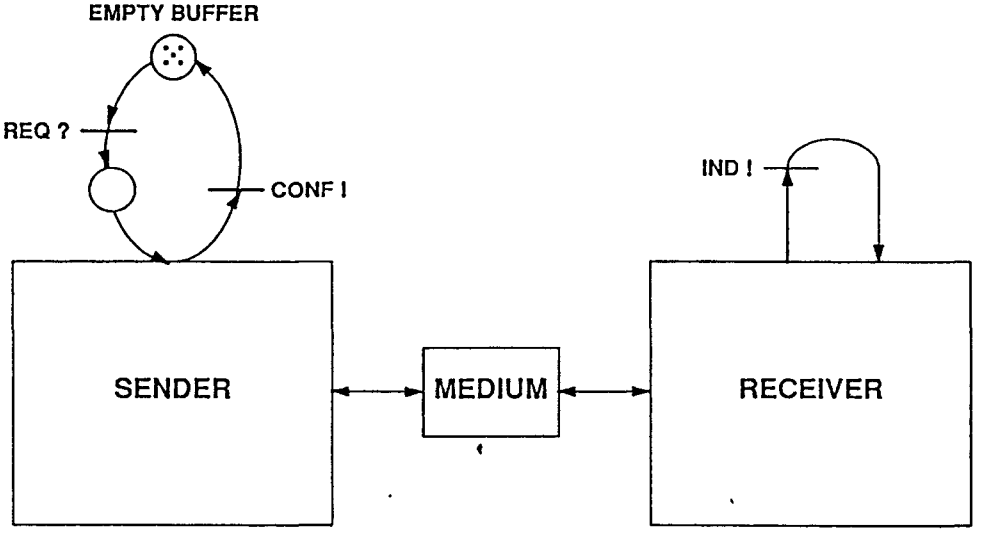
ALTERNATING BIT PROTOCOL



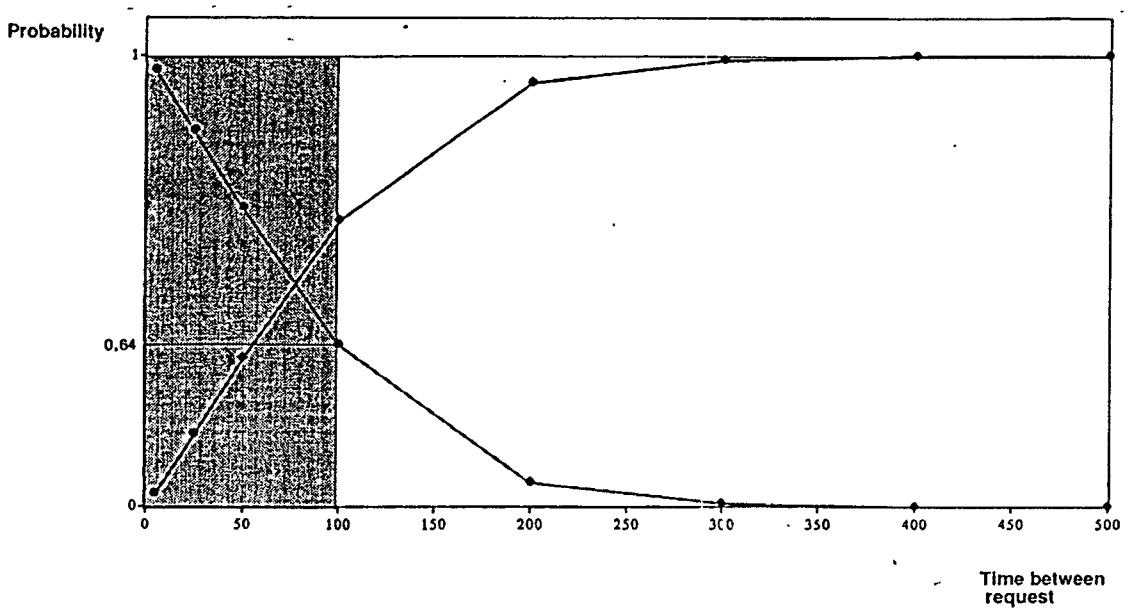
BIT = 0 ou 1

VERILOG STOCHASTIC PETRI NET

USER POINT OF VIEW



BUFFER LOAD



LOGISCOPE

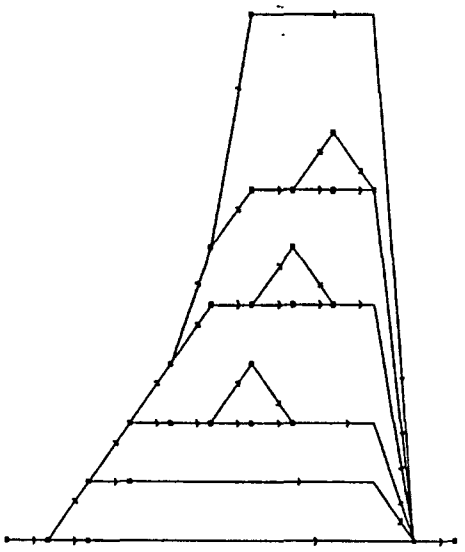
REAL TIME LANGUAGE ANALYSIS AND INSTRUMENTATION

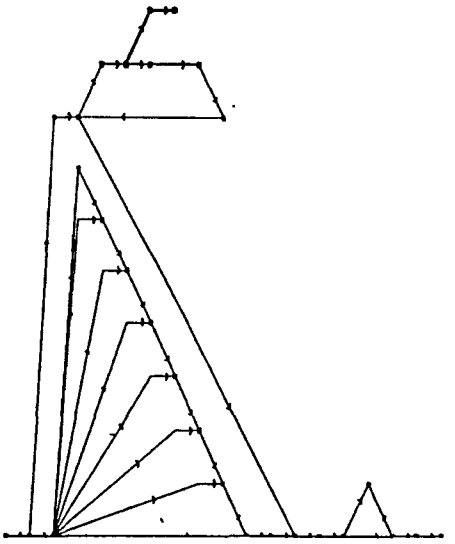
FOR

COMPLEXITY ANALYSIS

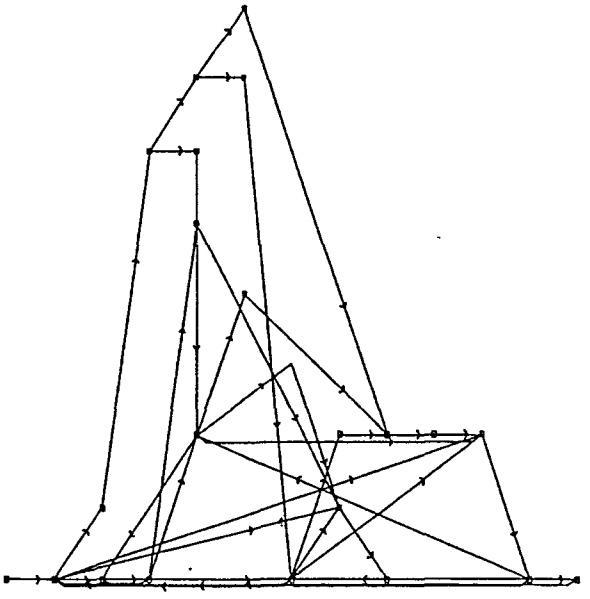
TEST COVERAGE

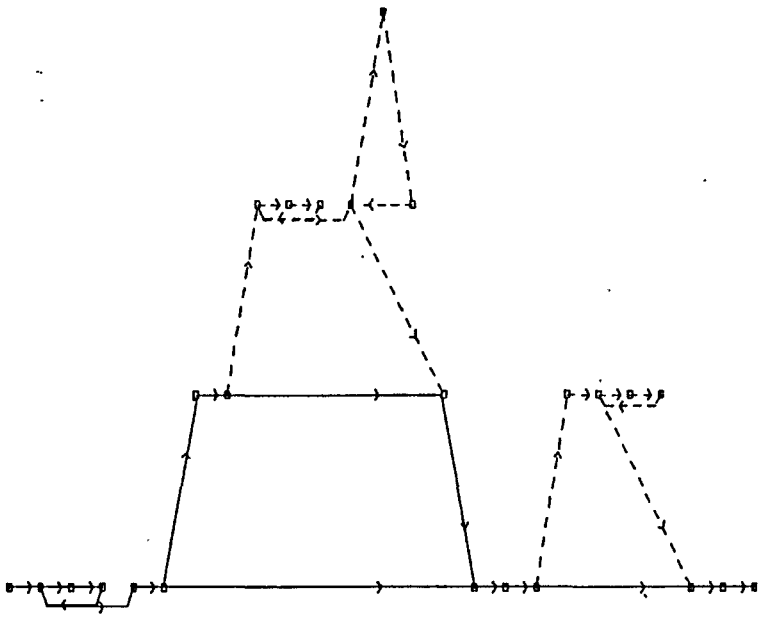
VERILOG LOGISCOPE control graphs

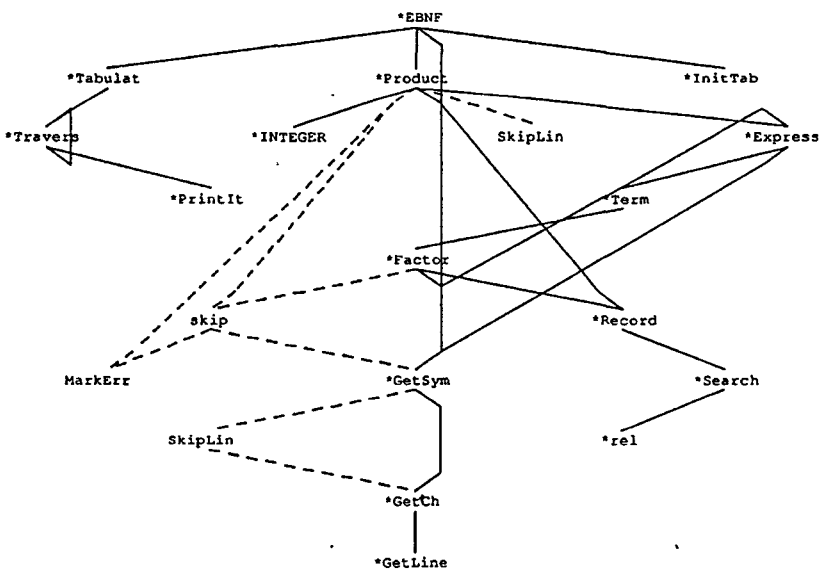




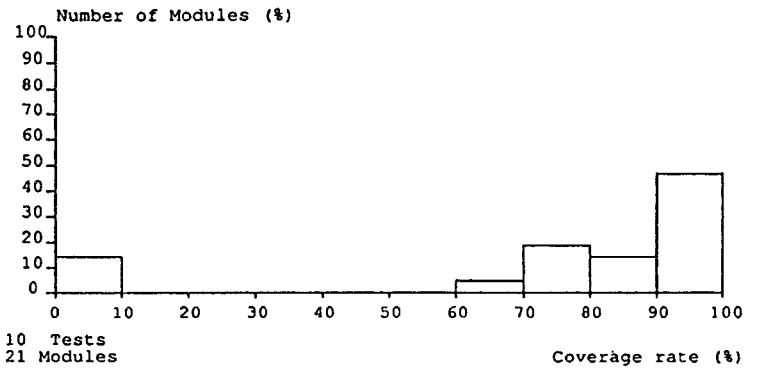
VERILOG LOGISCOPE diagnosis

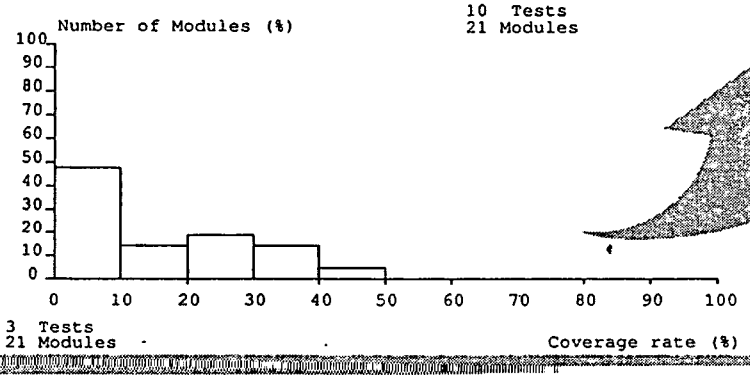





*EBNF
*Tabulat
*Product
*InitTab
*Travers
*INTEGER
SkipLin
*Express
*PrintIt
*Term
*Factor
skip
*Record
MarkErr
*GetSym
*Search
SkipLin
*rel
*GetCh
*GetLine

DDP	Line Number	Type	Condition
1	24	Start	
2	25	IF	sym = ident
3	28	ELSE	[sym = ident
4	29	ELSE-IF	[sym = ident & (sym = literal]
5	32	ELSE	[sym = ident] & sym = literal]
6	33	ELSE-IF	[sym = ident] & sym = literal] & (sym = lpar)
7	36	IF	sym = rpar
8	38	ELSE	[sym = rpar]
9	41	ELSE	[sym = ident] & sym = literal] & (sym = lpar)
10	42	ELSE-IF	[sym = ident] & sym = literal] & (sym = lpar) & (sym = lbk)
11	45	IF	sym = rbk
12	47	ELSE	[sym = rbk]
13	50	ELSE	[sym = ident] & sym = literal] & (sym = lpar) & [sym = lbk]
14	51	ELSE-IF	[sym = ident] & sym = literal] & (sym = lpar) & [sym = lbk] & (sym = lbr)
15	54	IF	sym = rbr
16	56	ELSE	[sym = rbr]
17	59	ELSE	[sym = ident] & sym = literal] & (sym = lpar) & [sym = lbk] & sym = lbr





AIRBUS A320 Equipment

Underground Train Safety

Certification of Military Application

Space Shuttle

Channel Tunnel

- EXISTENCE OF STRONG INDUSTRIAL PRACTICES
- CHANGE THE PRACTICES – IMPROVE PROFESSIONALISM
- FUTURE IS AUTOMATION AND REUTILISATION
- TESTING PRODUCT TOWARDS NEEDS WILL STILL REMAIN

Paper 3-T-3

CASE EVALUATION AND SELECTION PROCESSES

Mr. Leonard L. Tripp
Boeing Commercial Airplane

Mr. Leonard L. Tripp is in charge of the avionics software standards group at Boeing Commercial Airplane. He has been involved with software development standards, techniques and tools since 1970. He was the chairperson of the working group that produced IEEE Std 1002, Standard Taxonomy for Software Engineering Standards. He is currently a member of IEEE working group, P1209, Recommended Practice for the Evaluation and Selection of CASE Tools. He is the author of two books and over thirty technical papers. He obtained B.S. and M.S. degrees in mathematics from Brigham Young University.

CASE Evaluation and Selection Processes

Leonard L. Tripp

Boeing Commercial Airplane Group

Seattle, Washington

May 1990

Overview

- CASE Implementation Framework
- Evaluation Framework Basis
- Evaluation and Selection Process Framework
- Feature and Criteria Categories
- Feature Category Definitions
- Criteria Category Definitions
- Feature and Criteria Category Details
- Evaluation Process
- Selection Process
- Example
- Summary

Appendicies

Abstract

The use of CASE tools has focused on the overall goal of saving time and money and increasing productivity in the software development process. This approach uses tools to treat the symptoms of development problems, but not the cause. Typically, the root cause is the need to substantially improve the quality of the software. There is a demonstrated need for a methodology to select tools balancing quality and productivity issues. This briefing presents a comprehensive tool selection approach which involves both program and technical management of major software development efforts in the selection of a program specific CASE toolset. The approach includes the necessary procedures, tool categories, requirements and constraints which must be addressed in the selection process. The toolset is selected using an integrated set of quality and productivity criteria. The paper also illustrates the application of the selection methodology.

CASE Implementation Framework

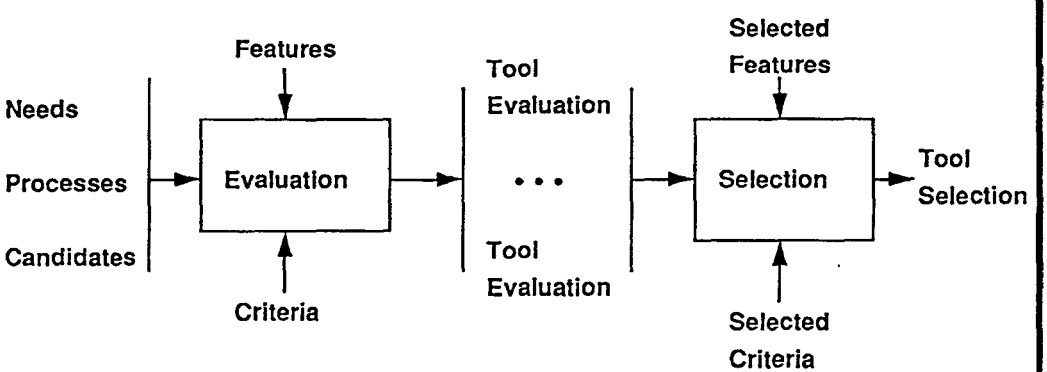
1. Obtain management commitment for change
2. Conduct organizational assessment
3. Develop CASE implementation strategy
4. Evaluate and Select CASE tools
5. Develop CASE implementation plan
6. Obtain implementation resources
7. Execute implementation plan
8. Collect metrics, monitor and evaluate

Evaluation Framework Basis

Software Tool Evaluation

- Feature - absolute characteristic of software
- Criteria - characteristics which are assessed in relative terms

Evaluation and Selection Process Framework



Feature and Criteria Categories

Feature Categories	Criteria Categories
analysis functions	correctness
applied standards	efficiency
configuration requirements	expandability
contractual matters	integrity
management functions	interoperability
performance requirements	maintainability
product identification	reliability
speciality requirements	reusabilty
transformation functions	survivability
	transportability
	usability
	vendor support
	verifiability

Feature Categories

- analysis functions
- applied standards
- configuration requirements
- contractual matters
- management functions
- performance requirements
- product identification
- speciality requirements
- transformation functions

Feature Category Definitions

analysis functions - A software function which provides an examination of a substantial whole to determine both qualitative and quantitative properties.

applied standards - Standards to which software or inputs or outputs conform.

configuration requirements - Those specific components of system hardware and/or software which are required in order for the software to function correctly.

contractual matters - Features determining the legal use of and support provided for software which may be specified in a contract with the vendor at the time of purchase.

Feature Category Definitions (C.)

management functions - Software functions which aid the management or control of software development.

performance requirements - A requirement that specifies a performance characteristic that a system or system component must possess, for example, speed, accuracy, size.

product identification - The specification of the manufacturer and version of the software which has the product capability.

speciality requirements - A requirement that specifies specialty characteristics the softwre must possess, for example, security, hardware control.

transformation functions - Software functions which describe how the subject is manipulated to accommodate the user's needs.

Criteria Categories

- correctness
- efficiency
- expandability
- integrity
- interoperability
- maintainability
- reliability
- reusabilty
- survivability
- transportability
- usability
- vendor support
- verifiability

Criteria Category Definitions

correctness - The extent to which the software design and its implementation complies with its specifications and standards.

efficiency - The extent to which software performs its intended functions with a minimum consumption of computing resources.

expandability - The degree to which architectural, data, or procedure design can be extended.

integrity - The extent to which unauthorized access to or modification of the software or data can be controlled.

interoperability - The extent to which two or more tools have the ability to exchange information and to mutually use the information that have been exchanged.

Criteria Category Definitions (C.)

maintainability - The extent to which a component facilitates updating to satisfy new requirements or to correct deficiencies.

reliability - The extent to which a component can be expected to perform its intended function in a satisfactory manner over a specified period of time.

reusability - The extent to which an existing software component can be used in other applications.

transportability - A measure of the effort required to transfer software from one hardware and/or software environment to another.

usability - The extent to which resources required to acquire, install, learn, operate, prepare input for, and interpret output of a component are minimized.

Criteria Category Definitions (C.)

vendor support - The extent to which a vendor is willing and able to provide the software user with assistance to ensure that the software performs desired functions and is willing and able to support the continuing maturation of the product.

verifiability - The extent to which a component facilitates the establishment of verification criteria and supports evaluation of its performance.

Feature Details

analysis functions

- consistency checking
- cross referencing
- data flow analysis
- mutation analysis
- regression testing
- requirements simulation
- statistical profiling
- traceability analysis

applied standards

- database standards
- graphics standards
- SQL standards
- tool-tool interface standards
- user interface standards

configuration requirements

- host hardware
- target hardware
- operating system
- minimum host primary memory
- minimum host disk space

Feature Details (Continued)

contractual matters

- number of users
- number of CPUs
- support availability
- basic software price
- installation costs
- other costs

management functions

- configuration management
- cost management
- object management
- performance monitoring
- program library management
- quality management
- resource management

performance requirements

- source code sizing
- storage
- timing requirements
- user profile

Feature Details (Continued)

product identification

- product name and version
- vendor name

speciality requirements

- associated tool requirements
- hardware control
- numerics
- options
- security issues

transformation functions

- activities transformation
- editing
- formatting
- incremental compilation
- linking
- object transformation
- program generation

Criteria Details

correctness

- completeness
- consistency
- traceability

efficiency

- communication effectness
- processing effectness
- storage effectness

expandability

- augmentability
- generality
- modularity
- self documentation
- simplicity

Criteria Details (Continued)

integrity

- security
- standard compatibility

interoperability

- communication commonality
- data commonality
- modularity
- rehostabilty
- retargetability

maintainability

- augmentability
- communicativeness
- consistency
- modularity
- self documentation
- simplicity
- structuredness
- test availability

21

Criteria Details (Continued)

reliability

- accuracy
- completeness
- consistency
- fault tolerance
- modularity
- simplicity

reusability

- application independence
- generality
- hardware independence
- modularity
- operating system independence
- self documentation

survivability

- autonomy
- distributedness
- fault tolerance
- modularity
- reconfigurability

22

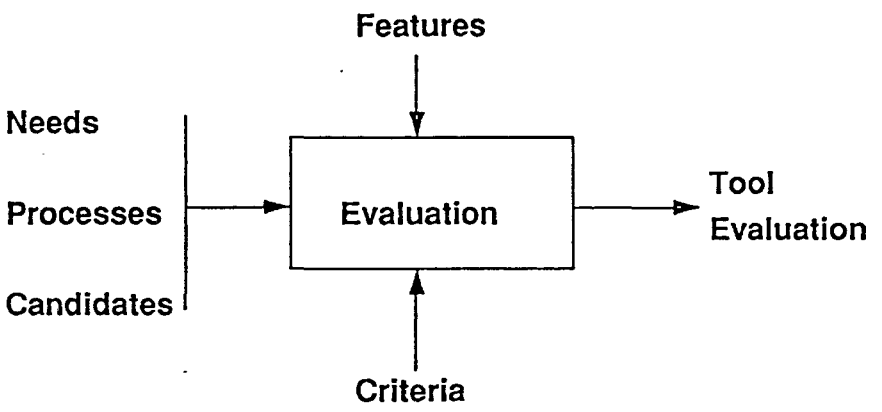
Criteria Details (Continued)

usability
- capacity
- ease of installation
- ease of use
- maturity
- on-line help
- power
- tailorability
- user documentation

vendor support
- corporate health
- pricing policies
- reputation
- support policies

verifiability
- communicativeness
- modularity
- self documentation
- simplicity
- standards compatibility
- structuredness
- test availability

Evaluation Process Overview



Evaluation Process

- Formulate candidate software search criteria
- Choose candidate software
- Choose evaluation features and criteria
- Perform evaluation
- Document evaluation results

Evaluation Process Details

- Formulate candidate search criteria - The criteria typically can be formulated from the CASE tool requirements or objectives.
- Choose candidate software - This may one or more products. A product can range from those that do a single function to those that perform a set of related functions.
- Choose evaluation features and criteria - Two cases: (1) evaluation and selection are independent and (2) evaluation and selection are coordinated. Evaluation is typically time-consuming and costly process. Hence, it is vital to choose only those features and criteria that are essential to the using organization. In some cases, if the cost of the tool and its implementation are reasonable, a detailed evaluation may not be required. In this case, a limited implementation may be the effective approach. In some cases, a parallel implementation of two candidates might be an alternative.

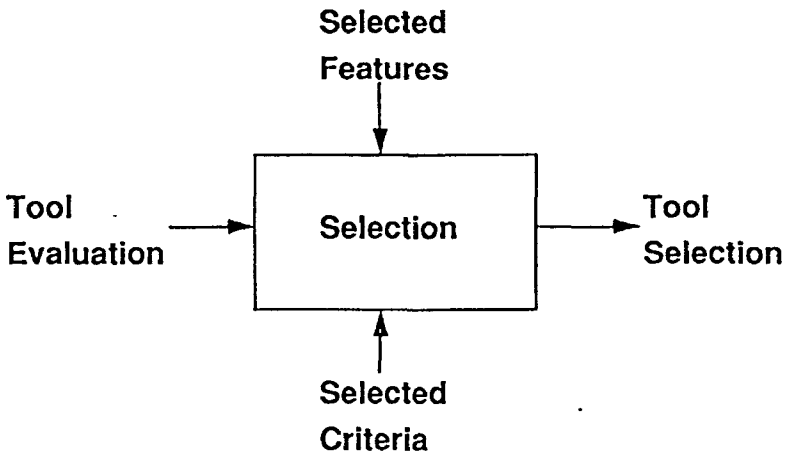
Evaluation Process Details (continued)

- Perform evaluation - The evaluation involves collecting data which are both subjective and objective in nature. The data is typically collected by a combination of analytical and demonstration methods. The balance between analytical and demonstration is influenced by how definitive the tool requirements are perceived to be understood. Typically, if the requirements are well-understood, then a checklist of detailed functionality and performance can be prepared and used to collect both subjective and objective data. On the other hand, if the requirements understanding is more general, then the focus is on gaining general impressions and perhaps, then confirm the impressions through further testing in the target organization. Examples of subjective data are the evaluation the "look and feel" of the software execution, assessment of how the software performs certain functions or evaluation of the quality of the system documentation. An area of concern with subjective assessment is understanding the amount of bias they contain. Objective data is characterized by discrete measurements reported in precise terms such as software perfomance. The quality and reliability of such data requires evaluation.

Evaluation Process Details (continued)

- Document evaluation results - The complexity of the evaluation results is related to the complexity of the evaluation process. In all cases, it is recommended that evaluation results should be documented. This will help in separating the evaluation and selection processes. It will also provide an audit trail. There are two difficulties in documenting subjective results. One is that there is needs to be a plan what features and criteria will be assessed and how to achieve consistency. The second difficulty is the reporting subjective results which arise because there is so much judgment invoved. The preparation of objective results typically involves the commitment of significant resources.

Selection Process Overview



Selection Process

- Determine features and criteria of interest
- Collect documented evaluation data
- Weight the chosen features and criteria
- Apply decision analysis
- Document software selection

Minimum Set of Features

- Product Identification
  - product name and identification
  - vendor name
- Configuration requirements
  - host hardware
  - minimum host memory
  - minimum host disk space
  - operating system
  - target hardware
- Contractual matters
  - basic software price
  - number of users
  - number of CPUs
  - installation costs
  - other costs
  - support availability
- Speciality Requirements
  - applied standards
  - associated tool requirements
  - security issues
  - user profile
- Functions supported

Minimum Set of Criteria

- application independence
- augmentability
- completeness
- consistency
- corporate health
- ease of use
- fault tolerance
- hardware independence
- modularity
- operating system independence
- processing effectiveness
- reputation
- simplicity
- standards compatibility
- storage effectiveness
- tailorability
- user documentation

Selection Process Details

- Determine features and criteria of interest - The features for evaluation and selection are not generally the same. In the selection process, it is often necessary to specify a level of acceptability for the chosen features. If the selector is not the evaluator, then the evaluation features and criteria should be chosen such that there is adequate coverage of important features and criteria. The framework can be used as a checklist in choosing the features and criteria.

- Collect reported evaluation data - The amount of collected data should be related to the size of the software investment and size of using community. Generally, the more opinions which are involved in evaluations, the more perspectives that are considered. If the decision group is large enough, then individual biases will be balanced out. The sets of data which are collected should be chosen because they come from reliable sources, they cover all of the software products of interest, and the important features and criteria are covered. It also important that the data be in a similar format.

Selection Process Details (continued)

- Weight the chosen features and criteria - Each feature and criterion must receive a weight which will establish it as more, less, or of equal importance as compared with other chosen features and criteria. The weighted features and criteria must be applied consistently to all products being considered as a group. The weighting scheme that is suggested assigns values ranging from 1 to 10 with one digit of precision.

- Apply decision analysis - This involves the use of the weighted features and criteria by the decision maker to select a software product. The application of the decision analysis may not produce one choice. The decision maker must have established a minimum level of acceptance, and if no product meets that level then none may be selected. The simplest decision analysis which can be applied is the use of a simple linear additive function. The composite rating for a feature or criteria is multiplied by its weight, then all weighted ratings are added together and the result is divided by the sum of the weights. It is important to remember that decision analysis calculations carrying only 1 significant digit.

Decision Analysis Example

Criterion	Weight (W)	Score (S)	W*S
Correctness	10	7	70
Efficiency	10	9	90
Expandability	8	7	56
Integrity	10	8	80
Interoperability	8	8	64
Maintainability	9	7	63
Reliability	9	8	72
Survivability	8	7	56
Usability	11	9	99
Vendor Support	10	8	80
Sum	93		730

Composite weighted score = 730/93 = 7.85 or 8

Example - Tool Objective by Life Cycle Phase

Life Cycle Phase	Objective
• Planning	• Provide cost and staffing estimates on task-by-task basis • Track dependencies between tasks • Provide schedule change analyses
• Requirements	• Determine completeness • Organize requirements
• Design	• Determine design performance • Verify design for integrity and completeness • Analyse design complexity

Tool Selection Example

Phase	Objective	Feature	Criteria	Candidate
Planning	Develop Task Estimates	Host H/W OS Ass. Tool Rqmts	ease of use user doc.	Primavera Project Timeline

Summary

- Toolset Considerations
  - life cycle view
  - tool dependencies
  - interrelationships of tools
- Iterative Aspects of Evaluation
  - survey market for tool availability
  - use demonstrations to clarify need
  - refine tool objectives
  - perform evaluation and selection again

Appendicies

- Bibliography
- Definition of Detailed Features
- Definition of Detailed Criteria

CASE Evaluation and Selection Bibliography

1. E.E. Anderson, "A Heuristic for Software Evaluation and Selection," Software- Practice and Experience, Vol. 19, No. 8, Aug. 1989, pp. 707-717.
2. J. Baker, "An Approach to Specifying Ideal Software Development Environments," IEEE 1987 (CH2503)
3. G. Baram and G. Steinberg, "Selection Criteria for Analysis and Design CASE Tools," ACM SIGSOFT Software Engineering Notes, Vol. 14, no. 6, Oct. 1989, pp. 73-80.
4. W.R. Beam, J.D. Palmer, and A.P. Sage, "Systems Engineering for Software Productivity," Trans. on Systems, Man, and Cybernetics, Vol. SMC-17, No. 2, Mar./Apr. 1987, pp. 163-185.
5. R. Bisiani, F. Lecouat, and V. Ambriola, "A Tool to Coordinate Tools," IEEE Software, Nov. 1988, pp. 17-25
6. P. Brereton (Ed.), Software Engineering Environments, Ellis Horwood Limited, Chichester, England, 1988.
7. T.A. Bruce, J. Fuller, and T. Moriarty, "So You Want a Repository," Database Programming & Design, May 1989, pp. 60-66.
8. D.L. Burkhard, "Implementing CASE Tools," Journal of Systems Management, Vol. 40, No. 5, May 1989, pp. 20-28.

Bibliography (Continued)

9. A.F. Case, Information Systems Development: principles of computer-aided software engineering, Prentice-Hall, 1986.

10. A.F. Case, "Evaluating and Selecting CASE Tools," CASE OUTLOOK, Vol. 1, No. 1, 1987.

11. R.N. Charette, Software Engineering Environments: concepts and technology, Intertext Publications, New York, 1986.

12. B.A. Christoph, "A Practical Approach to the Selection of Software Development Tools," Proceedings of the 18th Hawaii International Conference on Systems Science, 1985, pp 542-53.

13. M.R. Danziger and P.S. Haynes, "Managing the CASE Environment," Journal of Systems Management, Vol. 40, No. 5, May 1989, pp. 29-32.

14. S.A. Dart, R.J. Ellison, P.H. Feiler, and A. N. Habermann, "Software Development Environments," IEEE Computer, Nov. 1987, pp. 18-28.

15. Department of Defense, Evaluation and Validation (E&V) Guidebook, Version 2.0, 30 September 1989.

16. Department of Defense, Evaluation and Validation (E&V) Reference Manual, Version 2.0, 30 September 1989.

Bibliography (Continued)

17. R.D. Emrick, "Considering Computer Resource Consumption in the Selection of Appropriate Development Software," EDP Performance Review, Vol 13, No. 11, pp. 1-6.

18. R. Firth et al., "A Guide to the Classification and Assessment of Software Engineering Tools,: SEI Institute Report 87-TR-10, Carnegie-Mellon University, 1987.

19. A.S. Fisher, CASE: using software development tools, Wiley, 1988.

20. G. Forte, "In Search of the Integrated CASE Environment," CASE OUTLOOK, Vol. 3, No. 2, 1989, pp. 5-12.

21. G.D. Frewin, "Metrics in Procurement - a Discussion Paper," Measurement for Software Control and Assurance, Elsevier Appl. Sci., 1989, pp. 89-102.

22. M. Gibson, "A Guide to Selecting CASE Tools," Datamation, Jul. 1, 1988, pp. 65 and 66.

23. M. L. Gibson, C.A. Synder, and R.K. Rainer, Jr., "CASE: Clarifying Common Misconceptions," Journal of Systems Management, Vol. 40, No. 5, May 1989, pp. 12-19.

24. S. Glickman and M. Becker, A Methodology for Evaluating Software Tools," Conference on Software Tools, 1985, pp. 190-9.

Bibliography (Continued)

25. R.L. Glass, "Recommended: A Minimum Standard Toolset," ACM SIGSOFT Software Engineering Notes, Vol. 7, No. 4, Oct 1982.

26. P. Hass, "Criteria for Selecting Tools for Software Production," COMPA '84 Computer Applications, Software and Systems, 1984, pp. 305-320.

27. P.B. Henderson and D. Notkin, "Integrated Design and Programming Environment," IEEE Computer, Nov. 1987, pp. 12-16.

28. G.F. Hoffnagle and W.E. Beregi, "Automating the Software Devlopment Process," IBM Systems Journal, Vol. 24, No. 2, 1985, pp. 102-120.

29. R.C. Houghton, Jr. and D.R. Wallace, "Characteristics and Functions of Software Engineering Environments: An Overview," ACM SIGSOFT Software Engineering Notes, Vol. 12, No. 1, Jan. 1987, pp. 64-84.

30. C. Jones, "The Cost and Value of CASE," CASE OUTLOOK, 1987, Vol. 1, NO. 4, pp. 1-15.

Bibliography (Continued)

31. E.S. Kean and F.S. Lamonica, "A Taxonomy of Tool Features for A Life Cycle Software Engineering Environment," Rome Air Development Center, Griffis AFB, June 1985, DTIC Number AD B096 355.

32. R. Knight, "CASE Paybacks Perceived, If Not Exactly Measured," SOFTWARE NEWS, Feb. 1987, pp. 56-64.

33. P.K. Lawlis, "Supporting Selection Decisions Based on The Technical Evaluation of Ada Environments and Their Components," PhD. dissertation, Arizona State University, August 1989.

34. P.K. Lawlis, "A Supporting Framework for the Software Evaluation and Selection," Submitted to IEEE Computer for publication.

35. M.M. Lehman and W.M. Turski, "Essential properties of IPSEs," ACM SIGSOFT Software Engineering Notes, Vol. 12, No. 1, Jan. 1987, pp. 52-55.

36. C.F. Kemerer, "An Agenda for Research in the Managerial Evaluation of Computer-Aided Software Engineering (CASE) Tool Impacts," Proceedings of the 22nd Annual Hawaii International Conference on Systems Science, Jan. 1989.

Bibliography (Continued)

37. L. Lyon, "CASE and the Database," Database Programming & Design, May 1989, pp. 28-32.

38. R.E. Marmelstein, "Guidelines for Evaluation of Software Engineering Tools," Software Engineering and its Application to Avionics, 1988, pp. 17/1-6.

39. J. Martin, CASE and I-CASE: High Productivity Software, High Productivity Software, 1988.

40. J. Martin and C. McClure, "Buying Software off the Rack," Harvard Business Review, Vo. 61, No. 6, 1983, pp. 32-52.

41. R. Martin, "Evaluation of Current Software Costing Tools," ACM SIGSOFT Software Engineering Notes, Vol. 13, No. 3, Jul. 1988, pp. 49-51.

42. C. McClure, CASE is Software Automation, Prentice-Hall, 1989.

43. C.W. McKay, "A Proposed Framework for the Tools and Rules to Support the Life Cycle of the Space Station Program," Proceedings of the IEEE Compass '87 Conference, June 1987.

44. D. Mosley, "Breaking Down the Barriers to CASE," American Programmer, Vol. 2, No. 9, pp 11-17.

Bibliography (Continued)

45. C.R. Necco, N.W. Tsai, and K.W. Holgerson, "Current Usage of CASE Software," Journal of Systems Management, Vol. 40, No. 5, May 1989, pp. 6-11.

46. R.J. Norman et al., CASE Technology Transfer: A Case Study of Unsucessful Change," Journal of Systems Management, Vol. 40, No. 5, May 1989, pp. 33-41.

47. R.J. Norman and J.F. Nunamaker, Jr., "CASE Productivity Perceptions of Software Engineering Professional," Comm. of the ACM, Vol. 32, No. 9, Sept. 1989, pp. 1102-1108.

48. H. Oestreich, "Classification, Evaluation and Selection of Tools," COMPAS '84. Computer Applications, Software Systems, pp. 289-303.

49. L. Osterweil, "Software Environment Research: Directions for the Next Five Years," IEEE Computer, April 1981, pp. 35-43.

50. D.E. Perry and G.E. Kaiser, "Models of Software Development Environments," Proc. of 10th International Conference on Software Engineering, April 1988, pp. 60-68.

51. R.S. Pressman, "Selecting and Justifying CASE Tools," CASE OUTLOOK, Vol. 1, NO. 6, 1987.

Bibliography (Continued)

52. R.S. Pressman, Software Engineering, Second Edition, McGraw-Hill, 1987.

53 R.S. Pressman, Making Software Engineering Happen, Prentice-Hall, 1988.

54. W. Stinson, "Views of Software Development Environments: Automation of Engineering and Engineering of Automation," ACM SIGSOFT Software Engineering Notes, Vol. 14, No. 6, Oct. 1989, pp. 32-41.

55. L.E. Towner, CASE: Concepts and Implementation, Intertext Publications, New York, 1989.

56. D.A. Troy, "An Evaluation of CASE Tools," Proceedings of COMPSAC 87, pp. 124-30.

57. R. Vonk, "Analyst workbenches: een referentiekader," Informatie Jaargang, Vol. 30, No. 1, 1988, pp. 17-32.

58. P.T. Ward, "Embedded Behavior Pattern Languages: A Contribution to a Taxonomy of CASE Languages," The Journal of Systems and Software, Vol. 9, 1989, pp. 109-128.

Bibliography (Continued)

59. N.H. Weiderman, A.N. Habermann, M.W. Borger, and M.H. Klein, "A Methodology for Evaluating Environments," ACM SIGPLAN Notices, Vol. 22, No. 1, 1987, pp. 199-207.

60 D.N. Wilson, "CASE: guidelines for success," Information and Software Technology, Vol. 13, No. 7, Sept. 1989, pp. 346-350.

61. E. Winters, "Requirements Checklist for A System Development Workstation," ACM SIGSOFT Software Engineering Notes, Vol. 11, No. 5, Oct. 1986, pp. 57-62.

62. L. Zucconi, "Selecting a CASE Tool,", ACM SIGSOFT Software Engineering Notes, Vol. 14, no. 2, Apr. 1989, pp. 42-44.

63., CASE: The Potential and the Pitfalls, QED Information Sciences, Wellesley, Mass. 1989.

64., "CASE Tools Evaluation Criteria," CASE OUTLOOK, Vol. 1, No. 1, 1987.

Definitions of Detailed Features

activity transformation - A software function which performs a transformation on a product of one life cycle activity to produce a product to another activity.

associated tool requirements - Tools which must be available and compatible with the software.

basic software price - The costs involved in purchasing the rights to install software on a computer.

configuration management - The process of establishing baselines for configuration items, controlling the changes to these baselines, and controlling releases.

consistency checking - A software function which determines whether or not an entity is internally consistent in the sense that it is consistent with its specification.

cost management - A software function which supports the cost functions such as estimating, cost collection, etc.

cross referencing - A software function which relates entities to other entities by logical means.

data flow analysis - A software function which analyzes the formal requirements statements to determine interface consistency and data availability.

Definitions of Detailed Features (Continued)

graphics standard - A standard which governs how characters, symbols, or images are processed in a graphic display device.

hardware control - The ability of the software to control hardware directly, for example, interrupts, bit manipulations, file servers, task scheduling, preemption.

host disk space - The combined storage size (in megabytes) required of the on-line disk units of the host hardware to ensure that the software will run properly.

host hardware - The specification of the manufacturer and model of the computer hardware which will serve as the development platform for the software to be developed.

host memory - The size (in megabytes) required of the primary memory of the host hardware to ensure that the software will run properly.

installation costs - The costs required to make the software on a host computer ready for the first use by the end user.

linking - A software function which creates executable object code on the host machine from one or more independently compiled source code modules by resolving cross-references among the object modules, and possibly relocating elements.

Definition of Detailed Features (Continued)

mutation analysis - A software function which applies test data to a program and its "mutants" (programs that contain one or more likely errors) in order to determine test data adequacy.

number of CPUs - The number of computers which may legally serve as the residence for a particular software component.

number of users - The maximum number of users permitted simultaneous execution of a single purchased copy of the software.

numerics - Software features which determine the computational capabilities of the software.

object management - A software function which manages a collection of interrelated data (objects) stored together with controlled redundancy, serving one or more applications and independent of the programs using the data (objects).

object transformation - A software function which transforms a system object to another system object.

operating system - The specification of the name and version of the operating system under which the software will run.

Definition of Detailed Features (Continued)

options - Software features whose specified values (each of which causes the software to execute in a somewhat different, yet controlled manner) are set by the user.

performance monitoring - A software function which reveals the performance characteristics of the software component.

program generation - A software function which provides the translation or interpretation used to construct computer programs (such as language translator, generator, syntax analyzer, code generator, user interface generator, etc.).

program library management - A software function which performs the creation, manipulation, display, and deletion of the various components of a program library.

quality management - A software function which determines the achieved level of quality in deployed software systems.

regression testing - A software function which performs the rerunning of tests in order to detect errors spawned by changes or corrections made during software development and maintenance.

Definition of Detailed Features (Continued)

requirements simulation - A software function which expresses code-enhanced requirements statements to examine functional interfaces and performance.

resource management - A software function which manages the resources attributed to an entity.

security issues - Capabilities of software which protects its processing and data from unauthorized access or use.

source code sizing - The limits imposed on the size of selected components of the software.

SQL standards - A standard which defines the syntax, implementation or use of a data base query language.

statistical profiling - A software function which provides the analysis of a program to determine statement types, number of occurrences of each statement type, and the percentage of each statement type in relation to the complete program.

storage requirements - The amount of disk storage that is needed by the software to perform as a function of the problem size.

Definition of Detailed Features (Continued)

support available - The possibility of purchasing support for the software from the vendor on a continuing basis.

target hardware - The specification of the manufacturer and model of the computer hardware on which the software to be developed will be executed.

timing requirements - The limits imposed on the execution time of selected components of the software.

tool-tool interface standards - A standard which governs the exchange of information between tools from different vendors.

traceability analysis - A software function which can be used to relate a software component or design element back to its requirements.

user interface standards - A standard which governs how the interface to the user is built and used.

user profile - Characteristics required of the user in order to use the software productively.

vendor name - The name of the company that supplies the software and associated documentation to the users.

Definition of Detailed Criteria

accuracy - A quantitative measure of the magnitude of error expressed as a function of the relative error, with a high value corresponding to a small error.

application independence - The extent to which software is not dependent on the support required for a particular application.

augmentability - The extent to which software provides for expansion of capability for functions and data.

autonomy - The extent to which software is not dependent on interfaces and functions.

capacity - The extent of the upper and lower limits of the functions implemented by a tool.

communications commonality - The degree to which standard interfaces, protocols, and bandwidth are used.

communication effectiveness - The extent to which software performs its intended functions with a minimum consumption of communications resources.

Definition of Detailed Criteria (Continued)

communicativeness - The degree to which the software provides feedback while it is operating to keep the user informed of the functions being performed.

completeness - The extent to which a component provides the complete set of operations necessary to perform a function.

consistency - The extent to which uniform design and documentation techniques have been used throughout the software development process.

corporate health - The extent to which it is reasonable to assume that the vendor will remain in business with the ability to provide the committed level of customer support.

distributedness - The degree to which software functions are geographically or logically separated within the system.

ease of installation - The relative ease with which a software product may integrated into its operational environment and tested in this environment to ensure that it performs as required.

ease of use - The relative ease with which a novice user can become an effective user of the software.

Definition of Detailed Criteria (Continued)

fault tolerance - The extent to which the system has the built-in capability to provide correct execution in the presence of a limited number of hardware or software faults.

generality - The breadth of the potential application of program components.

hardware independence - The degree to which the software tool is decoupled from the hardware on which it operates.

host disk space - The combined storage size (in megabytes) required of the on-line disk units of the host hardware to ensure that the software will run properly.

maturity - The extent to which a component has been used in the development of deliverable software by typical users and to which the feedback from that use has been reflected in modifications to the component.

modularity - The extent to which software is composed of discrete components such that a change to one component has minimal impact on other components.

on-line help - The extent to which user documentation is readily available to the user from the program while it is operating.

Definition of Detailed Criteria (continued)

operating system independency - The degree to which the software is independent of operating system characteristics.

power - The extent to which a component has capabilities such as default and wild card operations, that contribute to the effectiveness of the user.

pricing policies - The degree to which the vendor's prices for product support and upgrades are reasonable and in accordance with accepted practice within the software industry.

processing effectiveness - The extent to which software performs its intended functions with a minimum consumption of processing resources.

reconfigurability - The extent to which software provides for continuity of system operation when one or more processor, storage units, or communication links fails.

rehostability - The extent to which a software component may be installed on a different host or different operating system with a minimum of reprogramming.

reputation - The degree of confidence expressed by software users in the vendor's willingness and ability to provide support for the software.

Definition of Detailed Criteria (continued)

retargetability - The extent to which a software component may accomplish its function with respect to another target with a minimum of modification.

security - The extent of protection of computer hardware and software from accidental or malicious access, use, modification, destruction, or disclosure.

self documentation - The degree to which the source code provides meaningful documentation.

simplicity - The extent to which the complexity of a system or system component (determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the type of data structures, and other system characteristics) is kept to minimum.

standards compatibility - The degree to which the software conforms to specific standards.

storage effectiveness - The extent to which software performs its intended functions with a minimum consumption of storage resources.

Definition of Detailed Criteria (continued)

storage requirements - The amount of disk storage that is needed by the software to perform as a function of the problem size.

structuredness - The degree to which a software component is constructed of basic set of control structures, each one having one entry point and exit.

support policies - The type and extent of support provided by the vendor the software.

tailorability - The extent to which the user interface of the software may be altered to conform to the preferences of the user.

test availability - The extent to which tests are available to verify that a program functions in accordance with its requirements.

traceability - The ability to relate a design representation or software component back to its requirements.

user documentation - The extent to which documentation conveys to the user of a system instructions for using the system to obtain the desired results.

Paper 3-T-4

THE 45 MOST IMPORTANT TESTING TOOL FEATURES

Mr. Dan Zimmerman
Director of Marketing
Software Research, Inc.

Mr. Dan Zimmerman is Director, Software Marketing at Software Research, Inc. (SR), San Francisco, California. He is a member of the Software Quality Technical Committee of the American Society for Quality Control and is a frequent speaker on quality assurance issues at Bay Area events. He is the author of a highly praised study, done for Frost & Sullivan, on the market for data entry systems. He holds a M.S. in Mathematical Sciences from Stanford University, and an M.B.A. from the University of California, Berkeley.

THE 45 MOST IMPORTANT TESTING TOOL FEATURES

"Testing tools, automated testing." The words are on everyone's lips. (Everyone in the software engineering and Software Quality Engineering world, anyway).

But what do they mean? Everyone it seems, has his own tool or feature of choice. Sometimes it seems like an election in a European or Middle Eastern country in times of turmoil: 110 political parties for every 100 citizens.

In this talk, I want to present my pick list of testing tool features from a variety of sources:

The *current tools on the market*
(particularly Software Research's),

Academic research tools, and

Customer comments.

Yes, there is some of my own discretion in this list. I have cut out large areas of testing that I felt were too impractical, such as Mutation Analysis and Symbolic Execution. The choices are too many. Even with this editing, I still have a large list and only have the time to sketch the features. Please see me after if you wish further explanation on any of the features I mentioned. So consider this one man's view of the testing tool world.

To explain a little further about this list, I started with the Software Research tools. I am the Director of Software Marketing for SR, and spend my days talking to people about the functionality of these and related tools. After 3 years, I do believe that this set is fine practical base and a leader in terms of breadth of testing tool functionality. The functionality of this group of tools can go a long way in building your complete function list.

In addition, I think it is good to mention the SR tools, since QW attendees often wish to know more about the tools, what they currently are and where they are going.

Capture & Playback systems:

(1) Capture of Keyboard and Mouse Activity.

Screens are captured for later comparison and review. **(SR: CAPBAK)**

Versions available for a variety of operating systems:

DOS,
MS-Windows,
UNIX RTE,
UNIX XWindows

General compatibility tools across many platforms.

(2) Editing of key/ mouse scripts with any editor.
(SR: CAPBAK)

(3) Interface with input generation programs. Input generation programs, such as AWK, SED, or Software Research's TDGen, can produce variations on an key/ mouse stroke file. The user can start with a key/ mouse stroke files, and produce versions of the key/ mouse stroke file with all the different permutations of various fields, or regular expressions.

Example: Take an accounting program. First, you use CAPBAK to capture a test session. You locate the portion of the keystroke file relating to a field, and insert a field value. TDGen can calculate variations.

(4) Time control.

Faithful time replay, or ability to speed up or slow down by a factor. Ability to put delays only against certain keystrokes.

Problem: variable loading during playback. Slower response during playback.
(SR: CAPBAK)

(5) Embedded programmable language.

(SR: CAPBAK/ UNIX, CAPBAK/ X)

"if (Systems or application call) then (Keystrokes) Else (other Keystrokes)"

While (system or application call) >... keystroke ....<.

The "While" construct implements the "Wait Until something happens" construct.

```
while  (notfound "string")
>
        ...
        sleep 6;
        ...
<
```

(notfound is a script that uses fgrep)

Other versions can wait until something happens in a window, etc.

MOTIVATION:

Many problems require repetitions of keystrokes, or a sensing of what has happened during the specific replay time. For instance, a login might be a part of your script, and CAPBAK might need to sense if you need to wait for the login.

(6) Embedded previously recorded key/ mouse stroke sequences in scripts.

```
Keystroke file:
>this is a keystroke file ....<
#include login
>....more keystrokes ... <
```

Problem: how to change complicated test key/ mouse stroke sequences with new releases of software.

General solution: Use a modularized keystroke design, so that specific key/ mouse stroke files contain changeable sequences.

Example: login sequences contained in a specific file, and called from files that exercise more of the application. Login can be easily changed.

(SR: CAPBAK)

(7) "Organized" or "grouped" key/ mouse strokes

Motivation: Need to make modifications of scripts more easily to fit new versions of the software.

Implementation: CAPBAK's Marker mode: User adds labels to the script, which demark the different parts. Display key/ mouse strokes in groups labeled by screen, goal. Users put comments that cover a section of key/ mouse strokes. Ability to search and sort on these labels

```
{Marker text: screen 1} qw 3 {CurDn}3{CurDn} {Esq}
{Market text: Screen 2}  654.33{CurUp}2{CarRtn}
```

Example: Automatically make search & change on all keystrokes that relate to one screen.

(SR: CAPBAK)

(8) Ability to simulate more than one test session at the same. (SR: NPlabak)
NPlabak is a utility that enables CAPBAK/ UNIX to replay more than one session at the same time.

Command Invocation: nplabak -f portmapfile

Where portmapfile is the following:

```
#    mtest map file  for sr0 demo
yrkeystrokefile1.ksv / dev/ tty7d 9600 yrscreenimagefile1.rsp
yrkeystrokefile2.ksv / dev/ tty7b 9600 yrscreenimagefile2.rsp
```

(9) On-the-fly editing of key/ mouse stroke files during replay.

One way of modifying key/mouse scripts is to edit after the test session is completed. Sometimes it is more accurate or easier to make modifications during the replay session. The user would press a configurable keystroke during replay and the last few key/mouse strokes would appear, ready for editing.

(SR: Planned: April 1991)

(10) Editing of key/ mouse strokes interactively during playback.

Menu assist in adding "IF.. THEN...", "WHILE" logic or a prerecorded collection of keystrokes.

A menu would appear to prompt the user in determining the correct information for logic statements.

(SR: CAPBAK/ X 1st quarter, 1991)

(11) Single step through script. **(SR: CAPBAK/ X 1st quarter, 1991)**

(12) Elapsed time tracking of transaction. The ability to time how long a system takes to accept and respond to a request. User presses a key to start timing, and then presses the same key to stop the time measurement. Useful in performance monitoring.

(SR: CAPBAK: 2nd Quarter 1991)

(13) Differencing: masking within a range.

Example: Total Amount: 1,000 Total amount should be accepted if it is between 900 and 1,200.

(14) Definition of screen variables: Identification of variables on the screen with screen positions or associated text. The variable could be used in the above program logic.

Example: On screen, it says: " Total Amount: 1234" CAPBAK would record the text "Total Amount: " and the position of the numbers, and know where to look. During replay, an "if " or "while " statement could compare the "total Amount " field to other values.

Benefit: With the ability to identify a field, user can
Compare against other values,
Put a value into a disk file,
Get a value from a disk file.

Additional adaptation for graphics:
How do you get the characters from
a graphic screen to evaluate and compare?
Need to build pattern matching services available

to recognize characters.
(SR: Expected August 1991)

(15) Graphic Differencing: Comparing 2 windows

Problems:

Comparing 2 windows that are shifted by a few pixels.

You want to automatically compare 2 windows that are the same except for different fonts.

Solution: Smarter Differencing utilities

(1) Passing 2 screen images if n% of the pixels are the same, or if they are the same ignoring a color or bit plane.

(2) Program that "searches" for an edge and then matches the edge by shifting and then comparing. User can specify the direction of search for an edge, and what the edge looks like.

(3) As above, pattern recognition programs to reduce graphics to characters.

(CAPBAK/ X: December 1990)

(16) Ability accept graphic screens of different formats. Microsoft graphic formats, screen dumps, post script, etc. Added on as user encounter them. **(CAPBAK, February 1991)**

(17) Network simulation: TCP/ IP, Ethernet, etc. CAPCOM. Coordination with key/ mouse stroke capture.

Example use: simulating a LAN connection to work station. Testing processing of a particular set of transactions sent over a LAN. **(CAPCOM: September 1991)**

TEST MANAGEMENT

(18) A manager, or higher level script description tool to specify test cases. (SR: SMARTS) . The higher level script would automate the execution and evaluation of interactive or batch programs.

An example from a SMARTS test script.

```
define case  yourtestcasename
{
Source
        "  comments here. Anything you want ";
Activation
        "plabak",
        /* plabak is a CAPBAK command */
        "cp a  b",
        "script  a c";
Evaluation
        "a.bsl" vs. "a.out";
```

Importance: Time savings, representation of test plans, standardization of scripts to make switching from one engineer's test script to another easy and easier to encourage test scripting. Reporting is easier with a large system, if each test case represents a particular functionality. Easy to see what is and is not working. Management by exception.

Comment: With this type of functionality, a capture playback tool becomes a testing tool, instead of a keyboard enhancer.

(19) Flexible evaluation criteria. Typically one file or screen compared against an expected file or screen, but could be system resource or elapsed time.

SMARTS allows you to write any function and use it as a evaluation criteria.

Syntax:

```
evaluation with function
        "yourfunction  arg1 arg2";
```

(20) Reset ability when the test case hangs. Ability to sense if a test case has taken too long, and kill the processes of the case. Then, the procedures specified in the Terminate clause are activated. The user writes the activation.

Example: User has 1,000 test cases for an application. The test cases run over the 4 days. The application is not completely stable, and it is certain that at least one of the test case will hang.

(21) A way to show the relationship between actual test case under execution and the high level test script and/ or the requirements document.

With Windowing systems its possible to run a CAPBAK test session in the main screen, and have a side window to show the SMARTS test tree, or hierarchy of test cases. As a test case is finished it is noted on the test tree window. Another window could show portions of the requirement's document as they execute.

Importance: Make sure that the tester understands exactly what needs to be tested, and can demonstrate it to his boss or outsider investigators.

DIFFERENCING, COMPARING

(22) Ability to compare windows and screens, and parts of screens.
(SR: CAPBAK/ MS-DOS, CAPBAK/ UNIX, CAPBAK/ X, EXDIFF)

(23) General masking ability: Ability to ignore irrelevant information, such as time stamps, operator names, system output variations, etc.

Masking of rectangles by coordinates. (Coordinates vary according to the graphics system used).

Creation of masks on screens interactively. **SR: Mask**

(24) Test reviewing features: Ability to review results of a test run, after the fact.

Toggle between two screens.
Highlight differences.
(SR: Mask)

(25) Masking features for text files:

Masking of X number of characters after a pattern match.
Line number mask.
Byte number mask.
Coordinate mask.

(SR: EXDIFF)

COVERAGE ANALYSIS:

26, 27, 28) Certification of the completeness of a test suite from the standpoint of the source code.

Logical branches
Call Pair
Paths through the code

Aid to the development of test cases that exercise missed logic

Guidance in the development of new test cases. By using Hot Hit report, the user can determine the best new cases to add to the test suite.

SOURCE CODE LEVEL APPROACH: **(SR: TCAT, S-TCAT, TCAT-PATH)** Instrument or preprocess the source code, then compile, link and execute the modified program.

Pros:
Higher level coverage
Logical branches
Call pair

Paths through the code
Data definition, use
More concise reports
Can be used on almost any embedded system

Cons:
15% - 20% execution time overhead for writing to the trace files.

(29) Coverage Analysis using the executable file Samples bus. Less overhead, but harder to read, and less helpful in finding errors. Typically has only statement coverage C0.

(30) Coverage Analysis of Definition - Use pairs. DU. This coverage tool will measure the percentage of paths between a data definition and the first data use. Like all coverage tools, the tools will tell what has been exercised and what still needs to be exercised.

Comment: Promises to be the next step after the C1 or branch testing of TCAT, and not as difficult to attain as the full path test of TCAT-PATH.

(31) UU, Use to Use.

TEST PLANNING

(32) Generation of test plans from Source Code: (SR : TCAT-PATH) Listing of all logical paths A covering set of paths is chosen to be tested. (Many can be chosen) A test set is developed to cover these paths assures logical branch coverage (C1) and a good Path coverage (Ct).

(33) Generation of test plans from pseudocode: User specifies pseudocode As with TCAT-PATH, a covering set of logic is created.

(**MetaTest: January 1991**)

(34) Visualization of Program logic. (SR : TCAT-PATH) Logical branches become lines and a diagram of the program is given. Character version now.

Next Step: Visualization of program Logic with advance GUI. Logical branches become lines and diagram of the program is given. Use X Window features, enabling intuitive representation.

(SR: TSCOPE Planned January 1991)

(35) Visualization of Program logic with coverage data. "Pulsating path, trembling trees" **(SR: TScope)**

REQUIREMENTS ANALYZERS:

(36) Test script generators from Requirements documents .

Works off of any specification document in electronic form. The user outline functions in an electronic form of the requirements text. Keeps track of the functions, automatically creates test scripts, such as SMARTS test scripts. **(SR: SpecTest planned in December 1990).**

(37) Hypertext analyzers: (build your own. Several Hypertext tools exist, for instance, Guide from Owl Software)

User makes hypertext notes on a requirements document. The requirements document could be a variety of other tools, such as Upstream CASE tools, or word processors.

(38) Traceability. The ability to match a test to a section of requirement and visa versa.

(39) Upstream CASE Specification Analyzers: (SR: TestCASE: planned in February 1991).

Generates SMARTS test scripts from front end CASE system specification. For instance, from Cadre, IDE, Index Technology specifications.

(40) Measurement of test completeness from point of view of requirements.

With requirements document, generate a percentage of characters included in a test, divided by the total number of characters.

With a upstream CASE specification, generate the percentage of processes with test cases divided by the total number of processes.

TEST CASE GENERATION TOOLS:

(41) Generation of variants of a template file. Summary: start with a file that represents most test cases. For example, a file that looks like most input transactions. Assume that all input transactions can be described by varying fields in the file, name, amount, trans type, etc. A second file has lists of field values. A variants generator would generate all permutations of the input file, or an input file with random selection of values.

(SR: TDGen)

(42) Generation of test input based on source code, a la Korel. Start with source code and executable to a program. A series of paths are calculated, as mentioned

above. A set of paths is chosen, and input is generated. Works on one path at a time, therefore can handle small number of paths. Tool will work like a debugger.

The general case, to calculate all possible input is the Holy Grail of testing, and virtually impossible. A proposed solution to produce input for any given path is presented in several papers by Bogdan Korel, of Wayne State University.

Summary of Korel's Approach: dynamic approach, starting with existing input, which is then modified to cover more and more branches of the desired path.

(43) Generation of test input from data dictionaries, description input situations, output situations.

Summary: User describes data elements ranges, input files, critical input and output situations.

Cause and effect graphing, boundary condition checking heuristics can be applied to calculate output.

EXPERT SYSTEMS:

(44) A checklist that can be applied to current situation with the expert system technology.

Collection of rules, "if... then..." statements to apply someone experience and expectations. Tester states a situation, and the the system mentions all pointers that apply at that time.

This is the wisdom gained through working on a particular system formalized in an expert system. A log of "gotchas" for type of program, therefore specific, and not generalizable.

Problems exist on how to describe the system to the expert system database. Many prototypes exist. See QW89 proceedings for reports on example systems.

(45) Reliability models: Input data: Coverage analysis, number of error discovered, complexity metrics, amount of time worked on.

Output: curve of cumulative errors found over time. Projection of this curve into the future.

Paper 3-M-1

HOW TO MANAGE REGRESSION TESTS

Mr. Bob Stahl
The Interface Design Group

Mr. Robert Stahl is the President of the Interface Design Group located in Petaluma, Ca. He is a former Director of MIS and has twenty five years of experience designing and testing software. He holds a B.S. in Physics and Mathematics from the University of California at Berkeley and M.S. in Physics from the University of Maryland.

How To Manage Regression Tests

(Excerpted from the FAST Methodology)

The Interface Design Group
P.O. Box 991
Bodega Bay, CA 94923
(707) 664-6531

The Big Questions

1. Did the change work?

2. Did it break anything else?

3. Is the RTS still OK?

Regression Testing Problems

Problem 1: Adequacy unknown

Problem 2: Can't capture some actions

Problem 3: RTS too large to run after each change

Problem 4: No correlation of individual tests to functional / structural parts of the system

Problem 5: Changes to system cause too many tests to " fail "

Problem 6: RTS gets out of step with the system

Regression Test Set Design

1. **Keep the tests simple.**
   - Each test should test one thing.
2. **Keep the tests independent.**
   - Each test should start and end at a common point.
   - Independent tests make it easy to add, delete, and reorder tests.
   - They also help to prevent a failed test from affecting following tests.
3. **Group linked tests into a Test Case.**
   - Test Cases are run, deleted, and moved as a unit.
4. **Perform (vs. embed) long, common sequences.**
   - This both shortens the RTS and smoothes handling the inevitable changes.
5. **Lock critical tests.**
6. **Package the State Of The World with the RTS.**
   - Include databases, profiles, environment.

Storing and Tagging Regression Tests

Test Id	Inputs	Outputs	Description
AT.18.321	2.18, A, Y	$918, Accept	Model Applicant
AT.18.330	3.62, F, N	$ 0 , Review	Applicant w/ priors

- File
- Database
- Data Dictionary
- Spreadsheet

EC VECTORS

An EC Vector is a list of the Equivalence Classes (EC's) for each input.

The list is in the same order as the input variables.

EXAMPLE:

The input variables are:

AMOUNT	PREAPPROVED	YTD

Their values are:

$300.00	Y	$6,450.00

The Equivalence Classes are:

AMOUNT: I (0 - 100), II (101 - 500), III (> 500)

PREAPPROVED: I (Y), II (N)

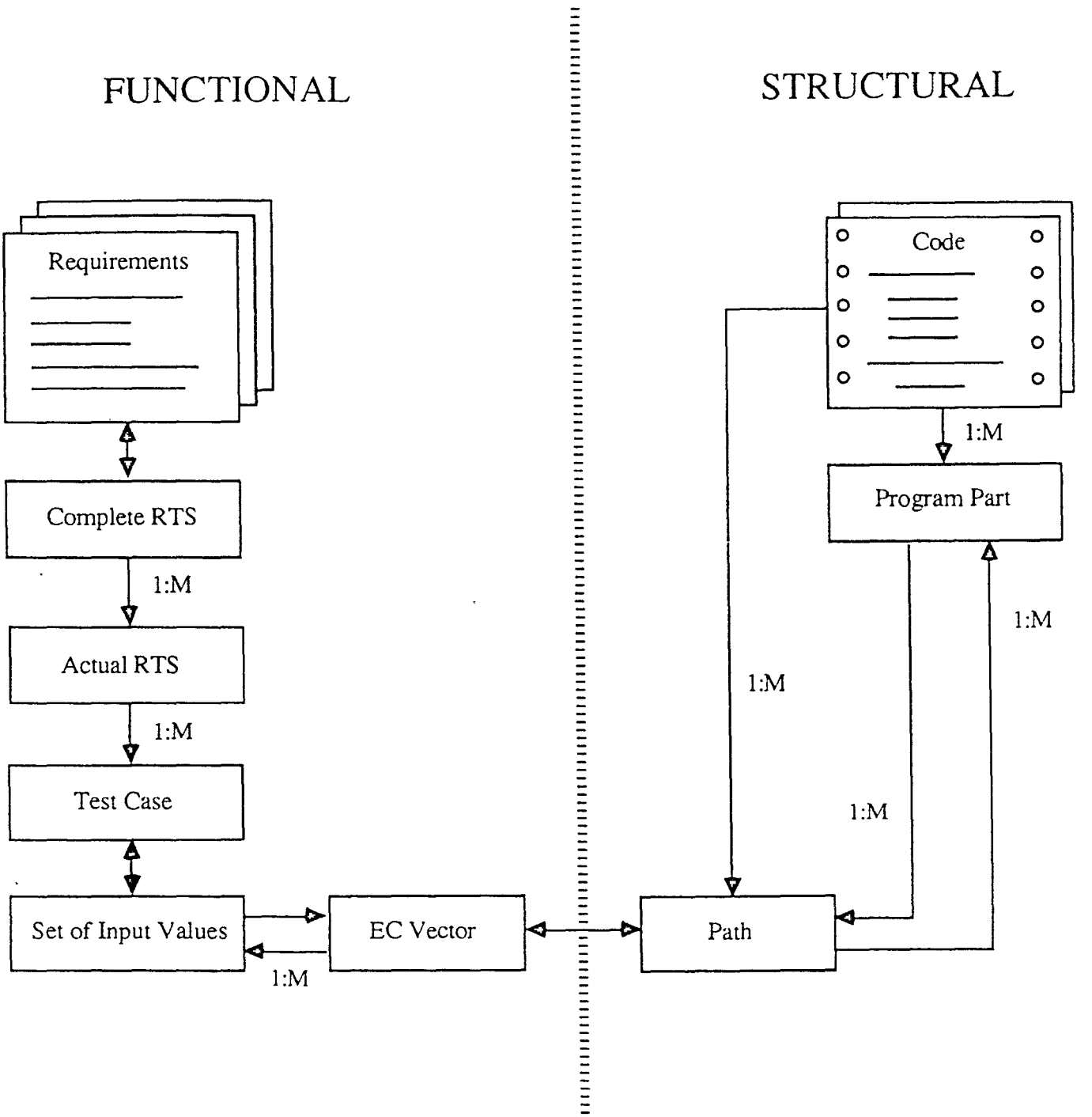
YTD: I (0 - 1,000), II (1,001 - 3,000), III (> 3,000)

The EC Vector for these inputs is: II, I, III

THE Questions

1. If I modify this part of the system, which regression tests need to be rerun?
2. What parts of the system does this test hit?
3. Which path does this test execute?
4. As a result of this modification:
   - Which tests should mismatch?
   - What should the results be?
5. Which tests are now:
   - Obsolete?
   - Redundant?
   - No longer test the same thing?

THE Picture


FUNCTIONAL
STRUCTURAL
Requirements
Complete RTS
1:M
Actual RTS
1:M
Test Case
Set of Input Values
1:M
EC Vector
Path
Code
1:M
Program Part
1:M
1:M
1:M

Types of Changes
Requirements Viewpoint

1. No effect at requirements level

2. Affects answers, but not decisions

3. Changes decision criteria

4. Adds or deletes decisions

Predicted Effect of a Change
RTS Viewpoint

1. Nothing

2. Maskable Items Only

3. Presentation Format Only

4. Answers

   - Predicted mismatches

   - Should have mismatched but didn't

5. Tests which still work but don't test the same thing now.

PRESENTATION FORMAT

Display Format:

- Location in window / screen
- Window size & location on screen
- Colors, borders, icon design . . .
- Device independence

Variable Characteristics:

- Right padding of alpha fields,
  left padding of numeric
- Type (fixed, floating, decimal places)
- Precision

PRESENTATION FORMAT SOLUTION

Object Oriented Tests

- Macros
- For Windows, OS/2 Presentation Manager:

  Switch Dynamic Linklibs (DLL's)

Are These Two Tests The Same?

INPUTS	OUTPUTS
5.0, Y, 8.2, BB	ACCEPT, $352.00
9.3, Y, 2.3, AX	ACCEPT, $1.91

RTS COMPRESSION

- Eliminates redundant tests
- Usually is the solution for:
  - RTS too large to run frequently
  - Which tests to rerun?
  - Too many mismatches
  - Adequacy of RTS unknown
- Can generate an RTS from production data!

Decision Maker Variables

A Decision Maker variable is not an original input.

It is formed in the program by compounding original input variables, and it is used to make decisons.

Decision Makers appear in the specs, and they have their own Equivalence Classes.

They must be included in the variables used in the EC Vector for compression.

Note that Boolean combinations (AND, OR, NOT) do *not* need to be considered as separate Decision Makers.

Examples:

IF ((A+B) < 100) THEN ADD 1 TO TOTALS.

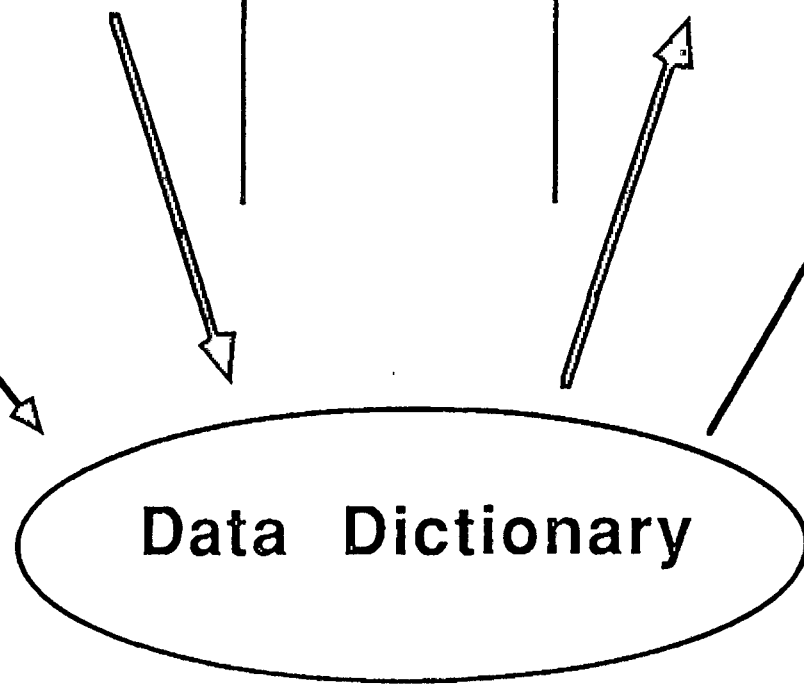
(A+B) is a Decision Maker

IF (A < 100 AND B < 100) THEN ADD 1 TO TOTALS.

No Decision Makers

RTS With EC Vectors

Test Id	Flags	Inputs	Outputs	EC Vector		Description
AT.18.321	L, S	2.18, A, Y	$918, Accept	II, IV, I	II, V	Model Applicant
AT.18.330	U, S	3.62, F, N	$ 0 , Review	I, IV, II	II, III	Applicant w/ priors



15

HOW TO COMPRESS:

- Sort on EC Vectors
- Eliminate duplicates

Paper 3-M-2

MANAGEMENT DECISION INFORMATION FROM SOFTWARE TEST PLANS

Ms. Jacqueline Jones
Senior Quality Assurance Consulant

Ms. Jacqueline Jones is a Senior Quality Assurance consultant and trainer at Computer Power-Applied Information, Oak Brook, IL. As a testing consultant and trainer, Ms. Jones has helped hundreds of executives, managers, developers, testers and users to understand software test planning. Ms. Jones has over 25 years experience in data processing and quality assurance. Her experience includes, programming, analysis, auditing, training and consulting in a wide-variety of business environments. Ms. Jones is a graduate of Cornell University with a Bachelor of Science degree in Industrial Engineering and Operations Research and a Master of Business Administration degree in Accounting.

MANAGEMENT DECISION INFORMATION FROM SOFTWARE TEST PLANS

Quality Week '90
Presentation
by

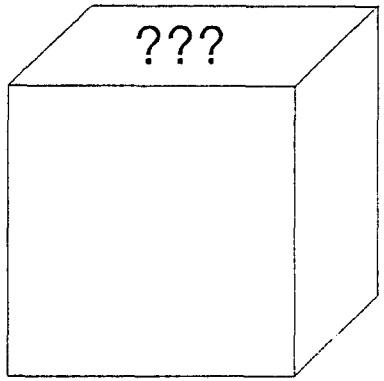
Jacqueline Jones, CQA
Computer Power Group - Applied Information

KEY POINTS

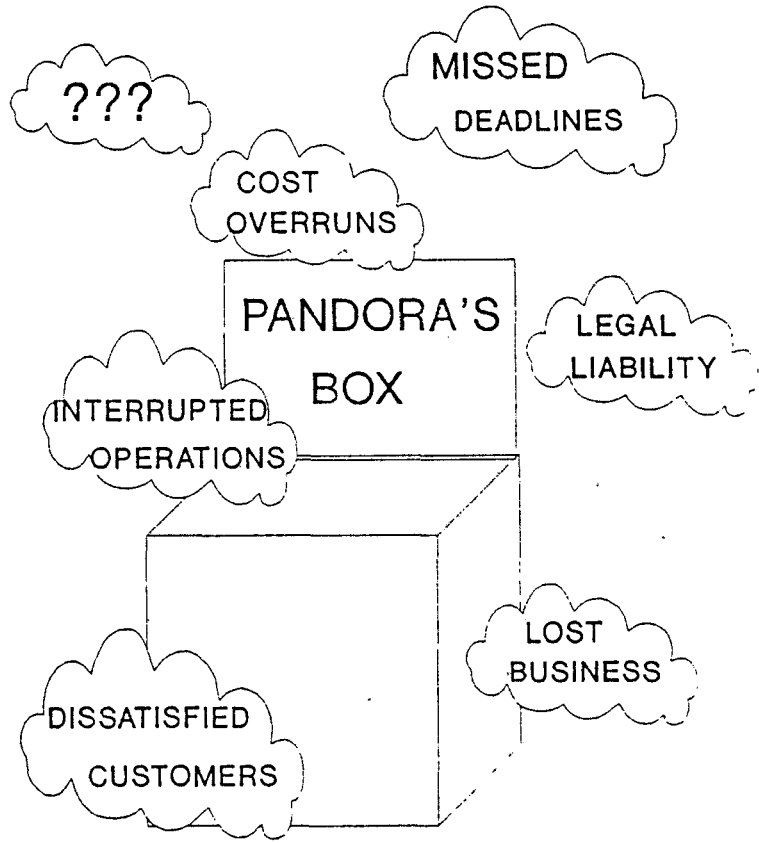
- Management may not be getting vital decision information from the test planning process.
- As organizations become more dependent on information systems, test planning is a tool for identifying system-related risks.
- Managers must understand testing and make testing decisions.
- Test planning deliverables are a rich source of information for risk management decisions.

WHAT YOU
DON'T KNOW

ABOUT A SYSTEM ...



CAN
HURT YOU!!



PROJECT INFORMATION
ABOUT

o STATUS

o RISKS

o TRUE COSTS

IS NECESSARY FOR MANAGING

SUCCESSFUL BUDGETS

TO BE
MEANINGFUL

THIS INFORMATION

MUST BE:

o Quantitative

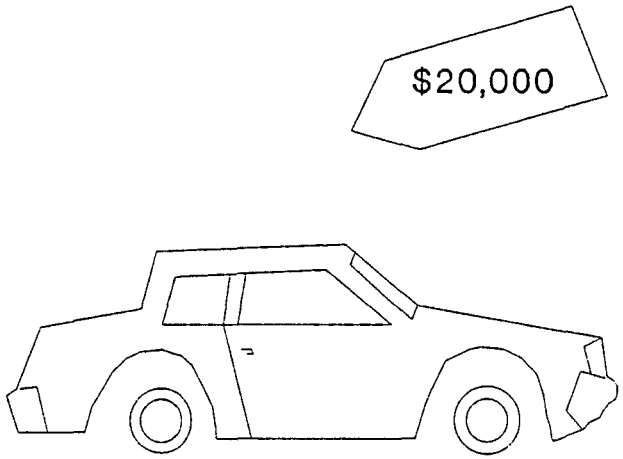
o Objective

o Timely

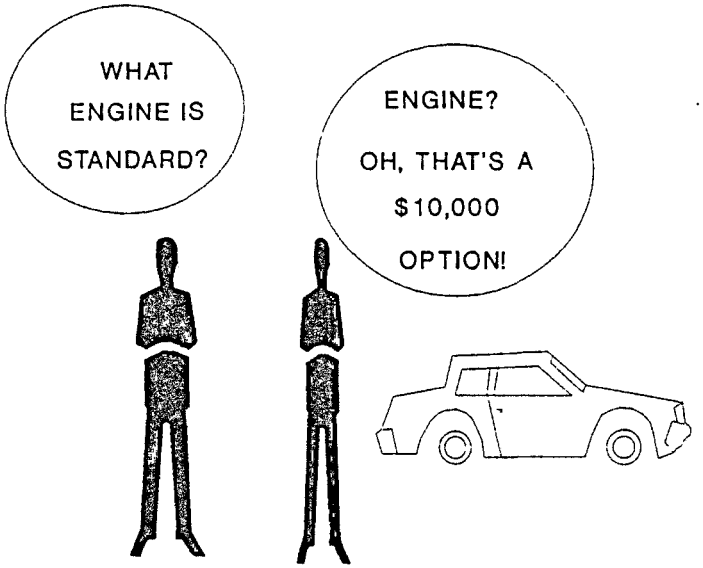
o Explicit

EXPLICIT

...



EXPLICIT ABOUT



- What IS or IS NOT Included

- Assumptions MUST BE well documented

WHY

is management

NOT GETTING

this vital

INFORMATION

?

We sometimes hear ...

They don't know what they need?

They don't know how to ask for what they need?

"People" don't want to hear bad news?

"We can always catch up -"

We can shorten schedule by "shrinking" testing time.

We often find that ...

TESTING

is MISTAKENLY viewed

as a

TECHNICAL

ISSUE

ONLY

!!!

BUSINESS ISSUES
IN TESTING

o SCOPE

what is or is not being tested

o OBJECTIVES

what is or is not included in testing objectives

o STRATEGIC RISKS

investment in testing versus potential cost of NOT testing

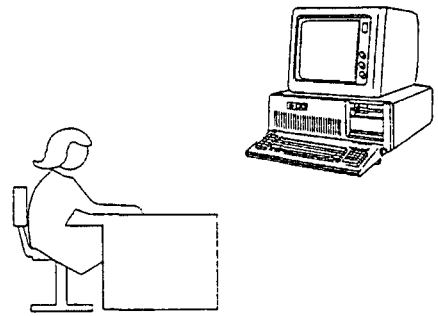
HOW and WHY
are these BUSINESS issues?

1. AUTOMATION

2. INTEGRATION

3. ACCOUNTABILITY

1. AUTOMATION



Trend toward automation

of ALL

KEY BUSINESS FUNCTIONS

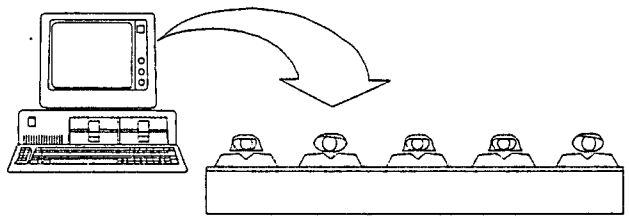
AUTOMATION
Characterized by:

- Increasing number of operational areas being automated
- Elimination of manual backup systems
- Loss of personnel with system-independent skills

INTEGRATION
Characterized by:

- A problem in one system causes problems in all
- Systems ARE the business

3. ACCOUNTABILITY



Enhanced systems facilities

enabling

ACCOUNTABILITY TRACING

ACCOUNTABILITY

Characterized by:

- Systems problems are more visible, in general
- System problems are no longer isolated events
- "The computer did it!" no longer implies NO FAULT
- Security and audit trails pinpoint problem sources

GOOD
TEST PLANNING

o ADDRESSES ALL OF THESE ISSUES

o IS A FUNDAMENTAL TOOL FOR IDENTIFYING SYSTEM-RELATED RISKS

BUT ...
Management MUST

MAKE TESTING DECISIONS

IN ORDER TO

MANAGE RISKS

AND TO MAKE
THESE DECISIONS

MANAGEMENT

MUST UNDERSTAND

THE TESTING PROCESS

in order to make

INFORMED DECISIONS

related to testing

WHY ISN'T
THIS HAPPENING?

1. INCOMPLETE
   UNDERSTANDING

2. INSUFFICIENT
   INFORMATION

3. INAPPROPRIATE
   PRESENTATION

INCOMPLETE UNDERSTANDING of TESTING

- Current scope AND limitations
- Process
- Roles and responsibilities
- Impact/importance

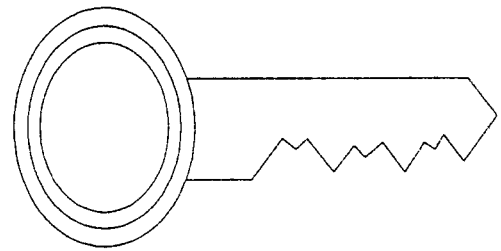
INSUFFICIENT INFORMATION to make INFORMED DECISIONS

- Assumption that management KNOWS underlying detail
- Cost/benefit analysis only IN TERMS OF MIS BUDGET
- "CRISES" prevent thorough research time

INAPPROPRIATE PRESENTATION of available information

- Use of TECHNICAL, instead of BUSINESS, terminology
- Lack of cost/benefit analysis
- Needs and benefits not related to business functions

The KEY is ...



TO VIEW

TEST PLANNING INFORMATION

AS

MANAGEMENT INFORMATION

HOW?

- Use BUSINESS, not technical, terminology
- Relate alternatives to business functions and risks
- Present organizational, not just MIS, costs and benefits
- Explicitly relate testing decisions to functional responsibilities
- Organize to present information in a timely manner
- Tie testing to corporate, strategic goals

TEST PLAN DELIVERABLES are a RICH SOURCE

of the

SPECIFIC information needed

to make INFORMED

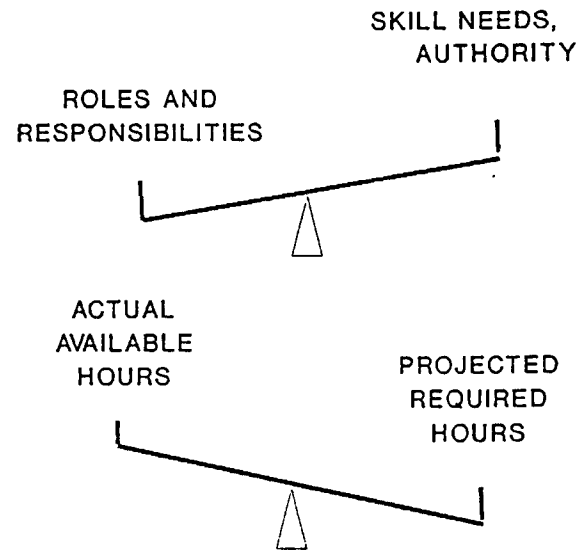
risk MANAGEMENT decisions

KEY
TEST PLAN DELIVERABLES

- STRATEGY PLAN
  - Testing organization
  - Requirements lists
  - Build strategy
  - User acceptance criteria
  - Assumptions
  - Coverage strategy
  - Tools/Training
  - Work plan
  - Procedures and standards
- TEST CASE DESIGNS
- STATUS REPORTS
- QUALITY ASSURANCE REVIEWS

TESTING ORGANIZATION

DO WE HAVE THE PEOPLE



TO DO THE JOB?

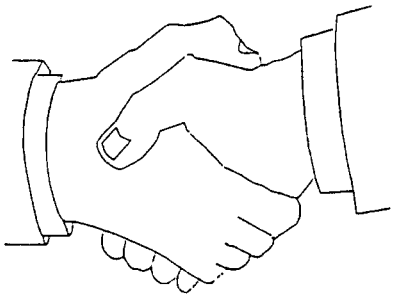
HIGH LEVEL REQUIREMENTS

1. CONTROL SYSTEM ACCESS
2. PROVIDE WIDGET DATABASE
3. MAINTAIN WIDGET INVENTORY
4. PROCESS WIDGET ORDERS
5. PRODUCE INVOICES
6. REPORT WIDGET SALES
7. BACKUP/RECOVER WIDGET DB
8. INTERFACE WITH G/L SYSTEM

Do Users, Developers, Testers,
and Management
ALL AGREE
this list SUCCINCTLY PRESENTS the
Major Business Functions, Procedures,
Security and Backup Functions,
Interfaces to Other Systems,
of the system?

User Acceptance

Have Users documented
SPECIFIC CRITERIA



for MEETING

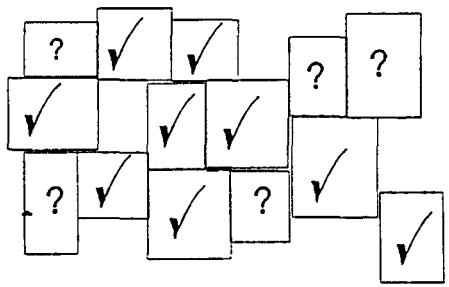
USER REQUIREMENTS

and

EXPECTATIONS?

Coverage Strategy

EXACTLY WHAT IS THE



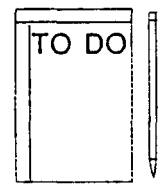
PLANNED SCOPE OF TESTING?

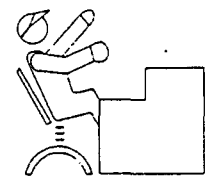
METHOD OF VERIFYING SCOPE?

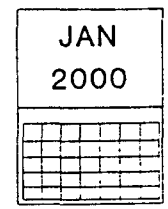
RATIONALE FOR EXCLUDING REQUIREMENTS OR CATEGORIES OF TESTING?

Work Plan

WHAT	WHO	WHEN
Test Productivity Reporting System	Fred Freeloader	Start: 6/15/90 Finish: 12/31/99



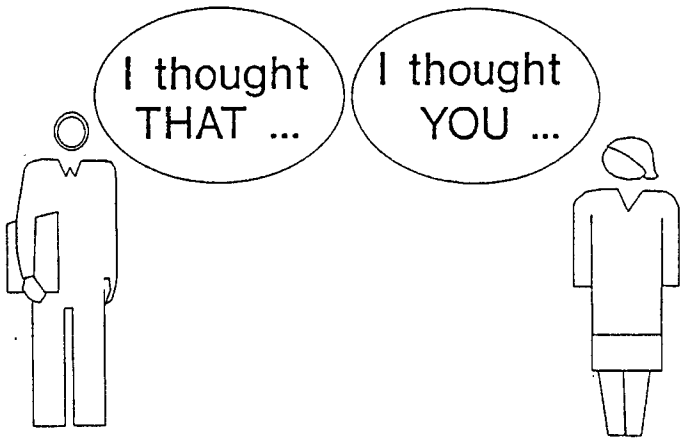




REALITY or FANTASY?

Are tasks, assignments, estimates and schedule REALISTIC and CONSISTENT?

EXPLICIT ASSUMPTIONS
to clarify expectations



Is there ANY POSSIBILITY
of even SLIGHTLY DIFFERENT
INTERPRETATIONS
of any requirement, task
or deliverable?

INTERMEDIATE LEVEL REQUIREMENTS

3. MAINTAIN WIDGET INVENTORY

3.1. Load Startup Inventory
3.2. Add Widget to Inventory
3.3. Change Widget in Inventory
3.4. Delete Widget from Inventory
3.5. Adjust Item Quantity
3.6. Report Inventory Activity

Do Users, Developers, Testers,
and Management
ALL AGREE
this list SUCCINCTLY PRESENTS
Job Tasks, Transactions, Screens,
Reports, Procedures and
and Performance Requirements,
of the system?

DETAIL LEVEL REQUIREMENTS

3.4. Delete Widget From Inventory

3.4.1. Validate widget code
3.4.2. Retrieve quantity on hand
3.4.3. Delete widget record
3.4.4. Pass code, qty to Act Rpt pgm
3.4.5. Display "Not On File" msg

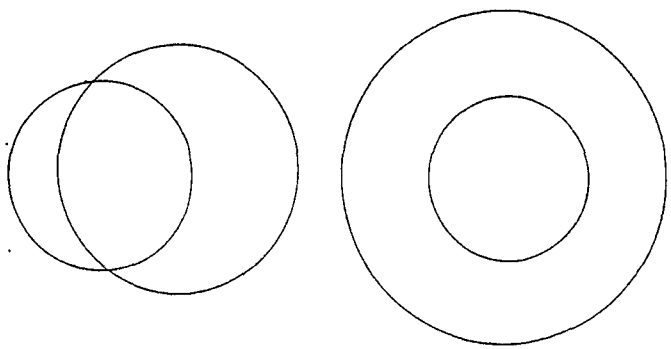
Do Users, Developers, Testers,
and Management
ALL AGREE
this list SUCCINCTLY PRESENTS the
Data Elements, Field Validations,
Cross Validations, Computations,
Editting, Formatting, and Data Displays
of the system?

PROCEDURES and STANDARDS

- Change Management
- Question/Discrepancy Reporting
- Problem Reporting
- LIbrary Control
- Quality Assurance Reviews
- Status Reporting

Change
Management

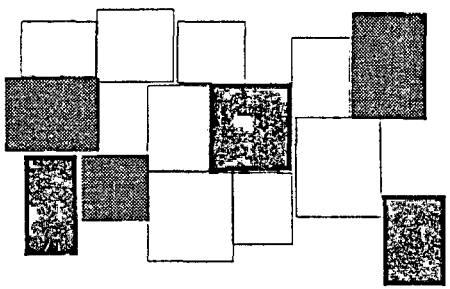
CONTROL SCOPE



CREEP and GROWTH

Question/Discrepancy
Reporting

IDENTIFY and CLARIFY

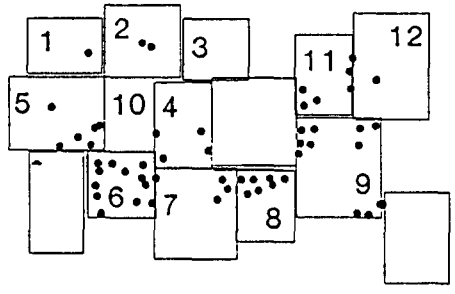


FUZZY OR MISSING

REQUIREMENTS

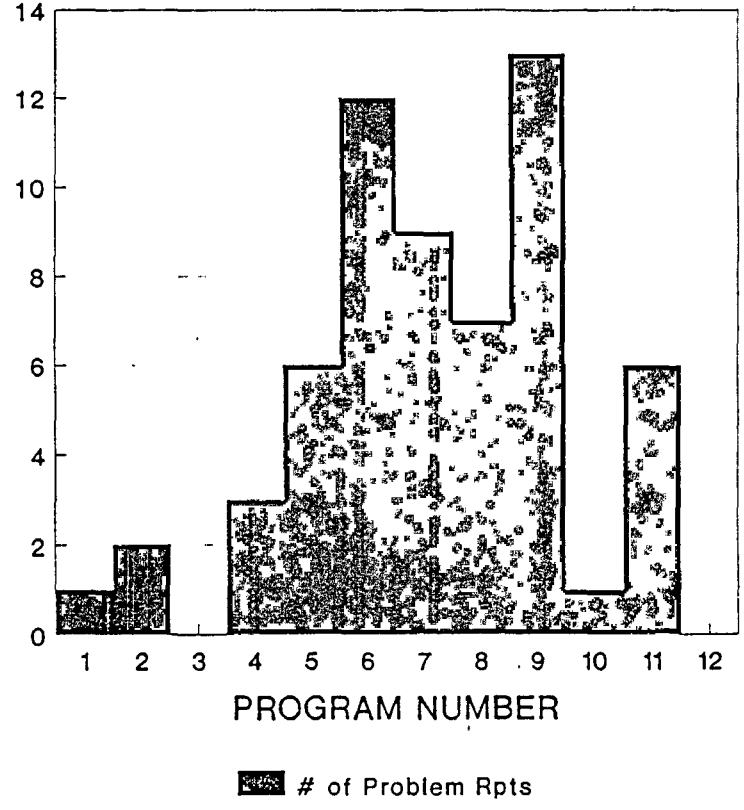
Problem Reporting

NOTE DEFECT CLUSTERING
TO FOCUS ON



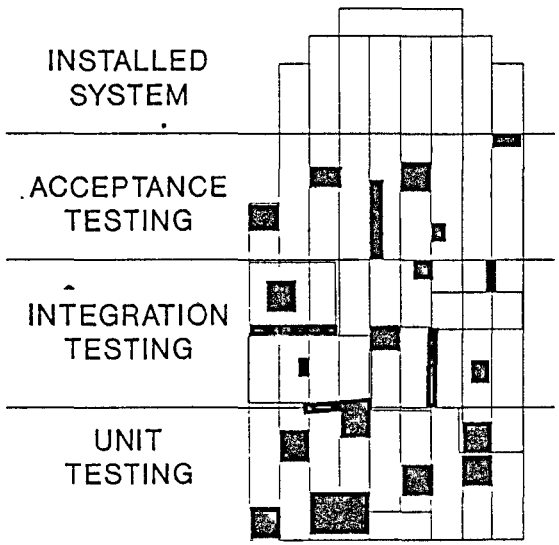
POORLY DEFINED REQUIREMENTS
OR
POORLY DESIGNED MODULES

Discrepancies/Problems



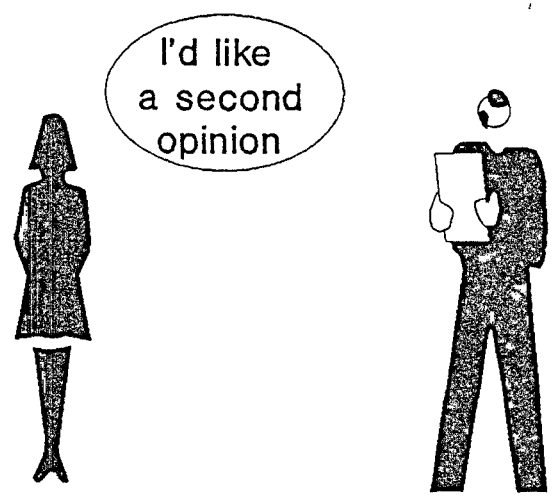
Library Control

Are system components
THOROUGHLY TESTED



at each lower level
BEFORE PROMOTION
to minimize risk of a
INSTALLED SYSTEM CRASH?

Quality Assurance Reviews



OBJECTIVE

RISK ASSESSMENT

MANAGEMENT DECISION INFORMATION
FROM SOFTWARE TEST PLANS

TEST ESTIMATING ASSUMPTIONS and ALGEBRA

As the project progresses and each estimate can be replaced with actual numbers, the remaining projections must be revised accordingly.

1. 10 High-level user requirements (HLURs)

2. 10 Intermediate-level user requirements (ILURs) / HLUR ==> 100 ILURs

3. 10 Detail-level user requirements (DLURs) / ILUR ==> 1000 DLURs

4. 4 Valid testable conditions / DLUR ==> 4000 valid testable conditions

5. 6 Invalid testable conditions / DLUR ==> 6000 invalid testable conditions

6. 5 testable conditions can be tested / test case ==> 2,000 test cases

7. 20 test cases / test script (or run) ==> 125 basic test scripts (or runs)

8. 1/2 hour design time / test case
   ==> 1,000 hours for test case design = 25 person-weeks
   ==> 2,000 hours for test execution = 50 person-weeks
   ==> 334 hours for test strategy plan = 8.5 person-weeks

9. 4 weeks for Requirements phase ==> 2 full-time Testers during Requirements phase to develop Test Strategy Plan

   12 weeks for Design phase ==> 2 full-time Testers during Design phase to design test cases

   24 weeks for Development phase ==> 2 full-time Testers during Development phase to execute tests

10. 3 weeks for Acceptance phase
   ==> 4 Users (with 30% of time devoted to Acceptance test case design, 12 person-weeks) during Design phase
   ==> 8 Users full-time during Acceptance phase to execute acceptance test cases (24 person-weeks)

Paper 3-M-3

CASE AND TOTAL QUALITY MANAGEMENT

Mr. Gene Forte
CASE Consulting Group, Inc.

Mr. Eugene L. Forte Principal Consultant and Executive Editor, CASE OUTLOOK is the President of CASE Consulting Group, publisher of CASE OUTLOOK, the CASE market's most influential journal and the CASE BUYER'S GUIDE, a comprehensive sourcebook on CASE products and services. As a researcher and analyst, Forte specializes in the latest tools and techniques for systems planning, development and management, and is a frequent speaker at industry conferences and workshops worldwide. Technical professionals, managers, vendors, and industry analysts look to CASE OUTLOOK for comprehensive reporting, interpretation and insight into the most significant trends in the CASE industry. Through its publications and consulting activities CCG is one of the most highly respected sources of information on the practical uses of systems design automation in both commerical and engineering applications. Consulting clients include IBM, Hewlett Packard, Cadre Technologies, CitiCorp, the Yankee Group, Mentor Graphics, The Institute for Information Industry, R.O.C and British Columbia Telephone. From its inception, CCG has promoted CASE as a comprehensive approach to systems development comprising all phases of the engineering life cycle, and all aspects of project management and control. An electronics engineer by training, Forte received his B.S.E.E. from California State University and also holds an Executive MBA from the University of Oregon.

CASE and TOTAL QUALITY MANAGEMENT

Gene Forte
President
CASE Consulting Group, Inc.

publisher of
C/A/S/E outlook®
Journal of Software Design Automation

Software Crisis

Computing, communications, display and actuator power increasing exponentially • • •

➪ The problems we are attacking are much bigger and more complex and contain a larger proportion of software.

➪ To enable viable new systems, the cost per function of software must decline and the reliability must increase in proportion with other disciplines.

So far, this hasn't happened!

Software Crisis or Chronic Affliction?

- Software development has been getting continuously worse for 20 years.
- Now the problem is more apparent.
- Software is MUCH more expensive.
- Software is strategically important.
- Growing opportunity cost.

What's Wrong with Software?

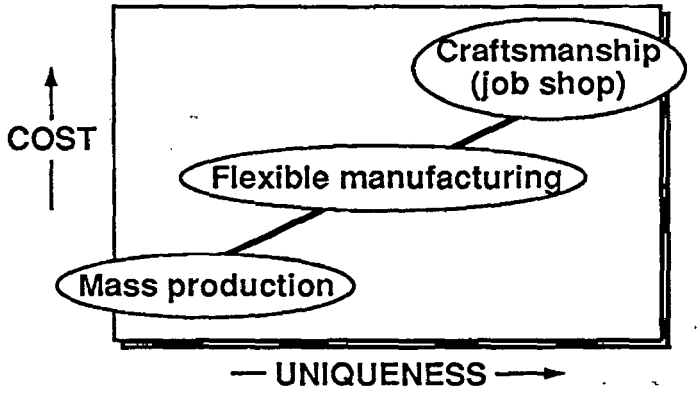
- Increasing size, complexity and cost.
- Methods and processes based on simpler systems and older technology.
- Processing power regarded as a precious resource.
- Management view of software as overhead cost.
- Lack of standardization: methods, tools, processes.
- Unsophisticated tools; slow tool evolution.
- Lack of understanding of the software engineering process; no metrics.
- Little environmental support for team engineering; inefficient support architectures.

How To Raise Productivity

1. Mechanization
2. Specialization
3. Management techniques

Source: TQC Wisdom of Japan, Hajime Karatsu, Productivity Press, Cambridge, 1988.

Software Development As A Manufacturing Process



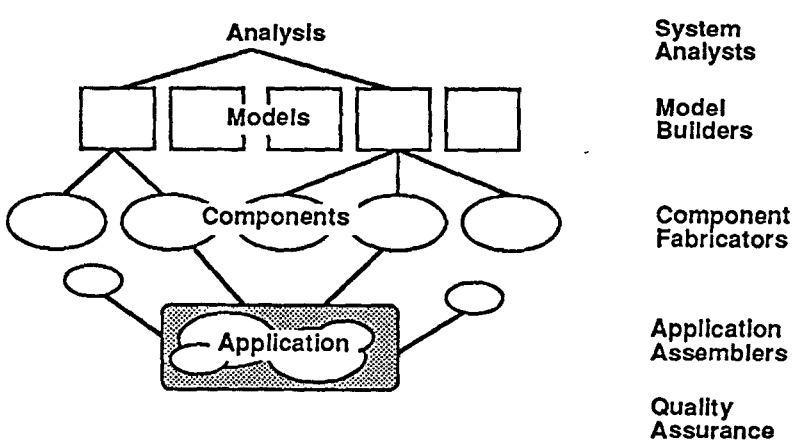
The Software Factory Paradigm

"A factory produces high quality goods efficiently."

- The machines must be precise and efficient.
- Workpieces must be moved efficiently and safely between machines.
- The production process must be well planned and managed.

Source: CASE: The Next Steps. Ovum Ltd., London.

Software Factory Model



Total Quality Management

- Process understanding and control
- Quality = meeting requirements
- Do it right the first time (zero defects)
- Continuous process improvement
- Emphasis on metrics

→

- Higher productivity
- Reduced cycle time

Japanese Manufacturing Philosophy

- Quality is free.
- Software engineers are experts. Managers serve the experts.
- Mistakes are treasures, the study of which leads to process improvement.
- Automation is valued because it facilitates consistent quality.
- Cost reduction comes by reducing development time.

Source: Production/Operations Management, Roger Schmenner, SRA, 1987.

Japanese Manufacturing Philosophy 2

- Any labor that does not directly add value to the product is waste.
- Labor is a fixed cost.
- Expediting and "work around" are sins.
- The cleaner and more organized the "factory floor," the better the S.E. will be able to identify problems in the process.
- Patience is more than its own reward.

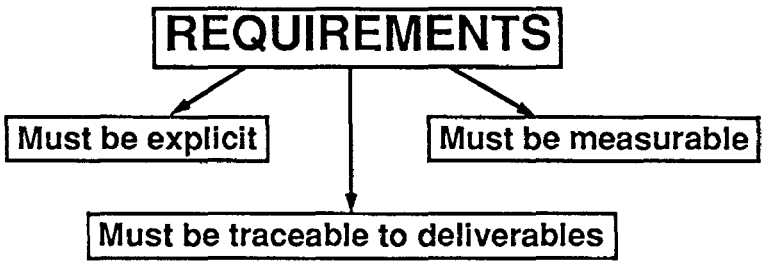
Source: Production/Operations Management, Roger Schmenner, SRA, 1987.

The Absolutes of Quality Management

- Quality means conformance, not elegance.
- There is no such thing as a quality problem.
- There is no such thing as the economics of quality; it is always cheaper to do the job right the first time.
- The only performance measurement is the cost of quality.
- The only performance standard is zero defects.

Source: Quality Is Free, Phil Crosby, McGraw-Hill, 1979.

Quality is Conformance to Requirements



Cost of Quality

"There is no such thing as the economics of quality."

"The cost of quality is the cost of non-conformance."

Source: Quality Is Free, Phil Crosby, McGraw-Hill, 1979.

Quality is Free

"Quality is free. It's not a gift, but it is free. What costs money is all the unquality things–all the actions that involve not doing the jobs right the first time."

"Quality is not only free, it is an honest-to-everything profit maker...If you concentrate on making quality certain, you can probably increase your profit by an amount equal to 5 to 10 percent of your sales."

Source: Quality Is Free, Phil Crosby, McGraw-Hill, 1979.

Quality is Free 2

"...anyone who has studied quality control knows that whenever defects are eliminated through quality control, costs go down."

"When I first visit a company that has not yet adopted QC, I'm invariably told that it cannot do so because the industry is unique in some way....As far as I'm concerned, there is no company or type of industry that cannot engage in quality control and benefit from it."

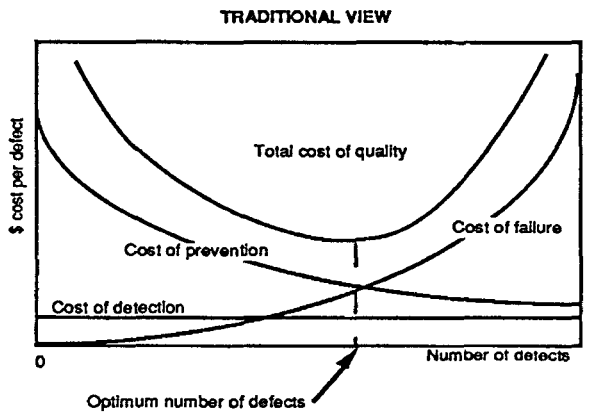
Source: TQC Wisdom of Japan, Hajime Karatsu, Productivity Press, Cambridge, 1988.

How TQC Reduces Cost

- Output that otherwise goes to waste is marketable.
- Production can be increased using the same resources.

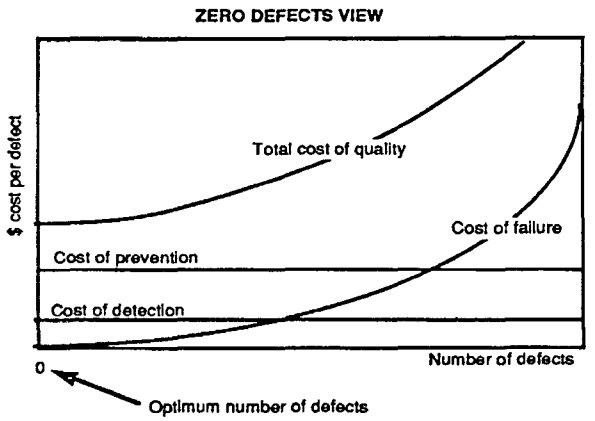
Source: TQC Wisdom of Japan, Hajime Karatsu, Productivity Press, Cambridge, 1988.

Why Quality is Free



Source: Production/Operations Management, Roger Schmenner, SRA, 1987.

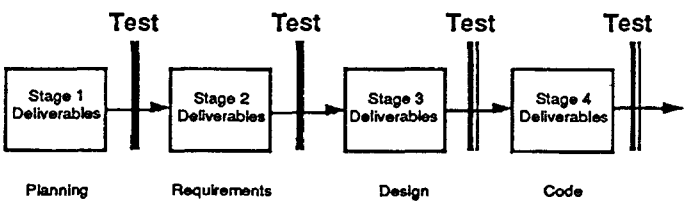
Why Quality is Free 2



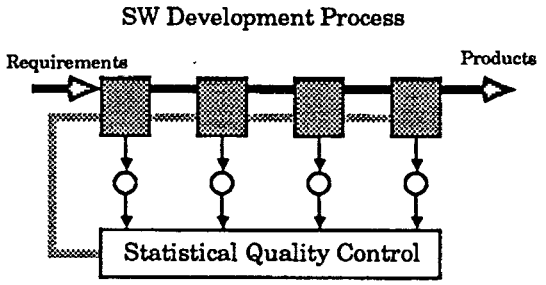
Source: Production/Operations Management, Roger Schmenner, SRA, 1987.

Zero Defects

Early defect detection and removal



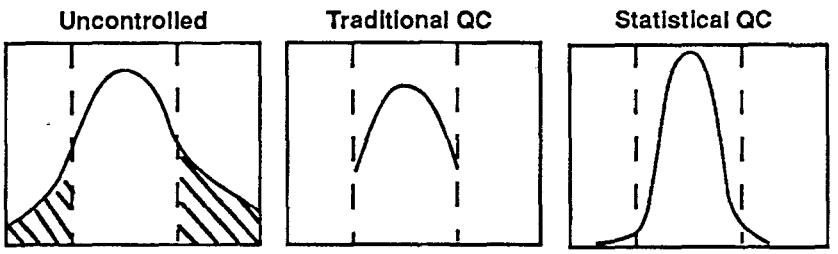
Continuous Process Improvement



Shewhart's Statistical Quality Control

- Detect the causes of dispersion (process analysis).
- Determine the degree of contribution of each factor.
- Stabilize the process (standardization).
- Prevent recurrence by constant monitoring (process control).

The Impact of SQC



Metrics

- Number of defects
- Defect severity
- Defect type
- Defect density
- Rate of defect discovery
- Outstanding problem reports
- Change requests
- User satisfaction
- Mean time to failure
- Mean time to repair
- System availability
- McCabe's complexity metrics & extensions
- Coding standards
- Function points

Total Quality Management and Software Development

- Explicit requirements & design
- Requirements traceability
- Zero Defects philosophy
- Process instrumentation
- Statistical quality control
- Continuous process improvement
- Objective use of process data

CASE and Software Quality 2

Design capture	• Standard Representations
	• Advanced Human Factors
Design validation	• Modeling and Simulation
	• Rapid Prototyping
	• Early Defect Removal
	• SW Instrumentation
Construction	• Automatic Transformation
	• Reusable components
Coordination	• Defined Methodology
	• Information Management

CASE and Software Quality

- Rigorous methodology
  - –Improved communications
  - –Reduced ego investment
  - –Proven design strategies
  - –Localization of defects
  - –Basis for early testing
  - –Basis for automatic transformation
- Automatic checking
  - –Tools find analysis & design errors
  - –Validation through modeling, simulation and prototyping
  - –Zero defects at each stage

CASE and Software Quality 3

- Change impact analysis
  - –Control of ripple effect
  - –Developers more willing to make changes
  - –Fewer undetected side effects
  - –Understand cost of change
- Automatic transformation
  - –Elimination of routine tasks
  - –Eliminates coding errors
  - –Reduces cost of construction
  - –Supports rapid prototyping

CASE and Software Quality

- Rigorous testing
  - –Test paths
  - –Test data generation
  - –Test coverage
  - –Test priority
  - – Regression testing
- Information management
  - –CASE repository
  - –Zero information loss
  - –Information available where it is needed
  - –Publication bottlenecks reduced
  - –Supports reusability

CASE and Software Quality

- Mechanization of process management
  - –Better process control
  - –Task coordination
  - –All tasks completed in proper sequence
  - –Deliverables completed and under CM
  - –ECR management
  - –Automatic process instrumentation
  - –Historical database
  - –Statistical analysis
  - –Decision support tools
  - –Management communication tools

Stages of Quality Maturity

1. Uncertainty
2. Awakening
3. Enlightenment
4. Wisdom
5. Certainty

Quality is Free, Phil Crosby, McGraw-Hill, 1979.

SEI Process Maturity Model

LEVEL	CHARACTERISTICS	KEY PROBLEMS	RESULT
Optimizing	Improvement fed back into process	Automation	Productivity Quality
Managed	Measured process (quantitative)	Technology change Problem analysis Problem prevention	
Defined	Process defined and institutionalized (qualitative)	Process meas. Process analysis Quantitative plans	
Repeatable	Process dependent on individuals (intuitive)	Training Technical practices Process focus	
Initial	Out of control (ad hoc/chaotic)	Project management Project planning Configuration mgmt. Software QA	Risk

Stages of Quality Maturity

1. Uncertainty
2. Awakening
3. Enlightenment
4. Wisdom
5. Certainty

Quality is Free, Phil Crosby, McGraw-Hill, 1979.

SEI Process Maturity Model

LEVEL	CHARACTERISTICS	KEY PROBLEMS	RESULT
Optimizing	Improvement fed back into process	Automation	Productivity Quality
Managed	Measured process (quantitative)	Technology change Problem analysis Problem prevention	
Defined	Process defined and institutionalized (qualitative)	Process meas. Process analysis Quantitative plans	
Repeatable	Process dependent on individuals (intuitive)	Training Technical practices Process focus	
Initial	Out of control (ad hoc/chaotic)	Project management Project planning Configuration mgmt. Software QA	Risk

Getting Your Process Under Control

Initial → Repeatable

Process Goals	CASE Support
• Management oversight	• Project planning
• Product assurance	• Cost estimation
• Change control	• Project control
	• Version management
	• Configuration mgmt.
	• Regression testing
	• Documentation

Getting Your Process Under Control

Repeatable → Defined

Process Goals	CASE Support
• Establish a process group	• Strategic planning
• Establish a software development process architecture	• Analysis tools
• Introduce a family of software engineering methods and technologies	• Design tools
	• Construction tools
	• Testing tools
	• Methodology guidance
	• On-line documentation

Getting Your Process Under Control

Defined → Managed

Process Goals

- Provide automatic support for gathering process data
- Turn the management focus from the product to the process

CASE Support

- Design metrics (including code)
- Project metrics
- Test results analysis
- Metrics database
- Statistical analysis
- Repository

Getting Your Process Under Control

Managed → Optimized

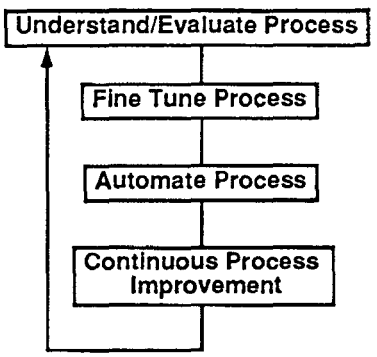
Process Goals

- Paradigm shift: focus on process optimization instead of product optimization
- Process tuning; continuous improvement

CASE Support

- Information resource management
- Software development architecture
- Reuse strategy & tools

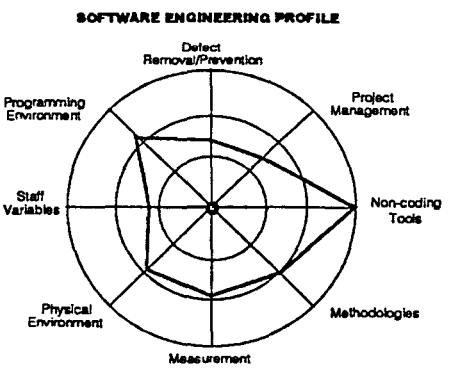
An Approach to TQM



- Define overall goals/metrics.
- Assess current process.
- Use metrics and SQC to identify problems and causes, simplify and control processes and identify best practices.
- Transfer best practices throughout organization.
- Build and use tools to automate processes & methodologies.
- Continue use of metrics and process assessment.

Source: Managing the Software Development Process, Hewlett Packard Co.

Software Process Assessment Profile



Source: Managing the Software Development Process, Hewlett Packard Co.

Four Pillars of a Quality Program

- Management participation and attitude
- Professional quality management
- Original programs
- Recognition

Managing for Total Quality

- *Develop the vision*
  - –Set policies that describe the plan
  - –Establish goals and objectives that require continuous measurable improvement
  - –Provide directives that prioritize goals and objectives
- *Clarify the vision*
  - –Harness the collective intelligence
  - –Focus the organization

Source: George Washington University School of Engineering

Managing for Total Quality 2

- *Establish the system*
  - –Set performance standards
  - –Prevent organizational collapse
- *Share the system with employees*
  - –Educate with philosophy
  - –Train with methods and tools

Source: George Washington University School of Engineering

The 14 Steps to TQM

1. Management commitment
2. Quality Improvement Team
3. Quality Measurement
4. Cost of Quality Evaluation
5. Quality Awareness
6. Corrective Action
7. Ad Hoc Committee for Zero Defects

Quality Is Free, Phil Crosby, McGraw-Hill, 1979.

The 14 Steps to TQM 2

8. Supervisor training
9. Zero Defects Day
10. Goal setting
11. Error cause removal
12. Recognition
13. Quality Councils
14. Do it over again

Quality Education

- Orientation to the concepts and procedures of quality
- Direct skills improvement
- Quality ideas communication program

Deming's Do's and Don't of Quality

- DO plot data chronologically to spot trends.
- DO gives lots of education and training on SQC to workers and managers at all levels.
- DO change the project manager's role from being quantity based to being quality based.
- DO stimulate teamwork across functions.
- DO drive out fear in the workplace resulting from fear of layoffs.
- DO inspect material (work products) 100%.

Source: Production/Operations Management, Roger Schmenner, SRA, 1987.

Deming's Do's and Don't of Quality 2

- DON'T expect slogans to work.
- DON'T pay workers according to incentive schemes based on quantity.
- DON'T award business to the lowest bidding vendor.
- DON'T have multiple inspectors.
- DON'T study defects from a process that is in statistical control; the variations are random.
- DON'T buy any equipment (CASE tools) until you have the process under SQC.

Source: Production/Operations Management, Roger Schmenner, SRA, 1987.

Barriers to Achieving Quality

1. Lack of proof that a quality program works.
2. Lack of positive support by senior management.
3. Pressure to meet short-term objectives.
4. Resistance to quality programs by middle management.
5. Lack of sufficient resources to implement quality program.

Source: Bill Perry, Executive Director, Quality Assurance Institute

Quality Can Be Threatening

- Use data to improve the *system*, not to blame *individuals*
- Focus on processes not human error
- Decisions about quality must be completely objective, based on data not opinion
- Keep data confidential

Software Quality Circles

- Cooperative effort between software development staff and QA staff
- Use data to determine the cause of defects
- Generate suggestions for improvement
- Assign responsibility for implementation
- Measure impact/improvement
- Make accomplishments and rewards visible

In Conclusion

- The solution to the software crisis is a combination of:
  - –Mechanization
  - –Specialization
  - –Management techniques
- The key management technique is Total Quality Control
- CASE supports and amplifies all three aspects.
- CASE is not a solution by itself.
- Methods, processes, skills and tools are equally important.

Paper 3-M-4

FILE MANAGEMENT AND REPORTING FOR SOFTWARE CONFIGURATION CONTROL "LESSONS LEARNED"

Ms. Regina Palmer
Martin Marietta

Ms.Regina Palmer is currently a Senior Software Quality Assurance engineer in the Research and Technology department at Martin Marietta Space Systems Company. She was the SQA lead on the RADC software program during the last of its development with oversight duties for system acceptance test.

FILE MANAGEMENT AND REPORTING FOR SOFTWARE CONFIGURATION CONTROL

Regina Palmer and Modenna LaBaugh
Martin Marietta Space Systems Company
P. O. Box 179, M/S H4330
Denver, CO 80201

This paper addresses the discrepancy reporting system used to identify nonconformances/changes to software and the automated Configuration Management (CM) system utilized on a Research and Technology contract at Martin Marietta Astronautics Group, Space Systems Company. The contract was sponsored by Rome Air Development Center (RADC), Command and Control Directorate, Software Engineering Branch, Griffiss Air Force Base, New York. The original schedule involved 3 incremental software deliveries, 2 hardware deliveries and acceptance test at RADC. The delivered system contained 199K lines of code (LOC) including Commercial Off-the-Shelf (COTS), legacy, and developed software. 38% of the code was COTS or unmodified legacy software. 21% was a conversion of graphics software implemented in VAX Fortran for display on Evans & Sutherland graphics terminals to Fortran 77 on an Apollo DN570 workstation. The remaining 81K was developed under the contract. The manpower on the program fluctuated from 4 to 8 programmers. The software was written in "C", Lisp and Prolog. This paper only addresses data collected for the 70K "C" code.

There were a total of 913 discrepancies found during the life of the program. There were 626 discrepancies found prior to the start of system test and 287 discrepancies were found after the start of system test. 69% of the total discrepancies were found prior to system test and 31% after the start of system test. These discrepancies were tracked with our internal software reporting system which tracked nonconformances and changes to software.

This paper is divided into two parts, Discrepancy Tracking and Change Control and Software Configuration Management. In the Discrepancy Tracking and Change Control section, the specific tool names and programmer names have been given alphanumeric codes to hide the identity of the programmers. The Software Configuration Management section identifies the specific tool names and correlates the tool names to the naming conventions.

DISCREPANCY TRACKING AND CHANGE CONTROL

The process of tracking discrepancies in software provides information to help improve productivity and efficiency. Through analysis, it was determined that engineers that had a higher percentage of errors per 1000 LOC, tended to be less efficient in the correction of errors. Often the corrections to the code generated additional errors. Two databases were used to track errors/error correction and change control.

Discrepancy Database

The discrepancy database was used to track errors and correction of errors. In addition, the database tracked the length of time it took an engineer to correct and test a discrepancy and the total lines of code that were changed. This database was also used to identify to management the workload of the engineers. The structure of the discrepancy reporting system database consisted of:

1) SARSNUM - The number of the discrepancy report was tracked to status the number of discrepancies open, the number of discrepancies written against each tool, the number of high priority discrepancies and the size of each open discrepancy. The discrepancy report we used was called the Software Automatic Reporting System (SARS).

2) OPENSRS - This field of the database tracked the date the discrepancy was opened which allowed the program to be statused on the open discrepancies. A discrepancy was considered open when Quality signed the discrepancy report.

3) DESCR - A description of the discrepancy was maintained to identify the discrepancy against a specific tool. A report could be generated to identify similar discrepancy descriptions so that management could see a trend of certain types of discrepancies.

4) DISP - A description of the correction of the discrepancy was maintained to track what was accomplished to correct the problem. This field of the database was also used to document the test case that was run to test the change. If an error was discovered in a tool that was a similar error in another tool, a report could be generated for the engineer which could identify the corrections to discrepancies instead of looking through all the discrepancy reports.

5) PRIORITY - There were five priority levels for discrepancies established that described the severity of the discrepancy. These five priority levels ranged from "A" to "E":

   a) An "A" discrepancy was an error in the code in which the software did not meet the requirements or design, an error which was a documentation error which caused the code to not meet the requirements or a code error which crashed the system making the system nonoperational until the error was fixed.

   b) A "B" discrepancy was an error which crashed the system but there was a work around and the system could be used.

   c) A "C" discrepancy was an error found in the code which did not interfere with the operation of the system.

   d) A "D" discrepancy was a minor error in the code such as a typographical error in a help message.

   e) An "E" discrepancy was an enhancement to the current system, directed by the contractor or the customer.

6) LOCATION - This field of the database tracked discrepancies found by the customer and discrepancies found internally. This field had two entries: "C" for customer and "I" for internal.

7) SIZE - This field was used to identify the approximate time to fix a discrepancy ranging from "small" to "large":

   a) A "small" rating meant that the discrepancy could be fixed in less than four hours.

   b) A "medium" rating meant that the discrepancy could take between four and eight hours to fix.

   c) A "large" rating meant that the discrepancy could take more than eight hours to fix.

8) CLOSEDSRS - This field of the database tracked the date the discrepancy was closed. A discrepancy was closed when Quality had performed a build of the system and the changes to the tool had been successfully tested. Quality would then sign signifying that the discrepancy was closed.

9) CAUSE - This field identified the cause of the discrepancy such as a requirements error, design error or coding error.

10) ICA - This field identified the correction of the discrepancy such as requirements revised, design revised or code revised.

11) ENG-HOURS - The number of hours to fix a discrepancy was maintained to identify the length of time it took to correct a discrepancy/change.

12) CPC_NO - The tool number was tracked to identify the number of open discrepancies against each tool, the size of discrepancies against each tool and the priority of discrepancies against each tool.

Discrepancy Reporting

The reports generated from the discrepancy database consisted of:

1) An Open Status Report was generated on a monthly basis which identified the discrepancies that were open and which tool had discrepancies open against it. During system test, this report was generated daily. It was used by program management as a daily synopsis of the test effort. An example of the report is shown in Figure 1.

2/15/88 **SARS OPEN STATUS REPORT** PAGE: 1

SARS #	OPEN DATE	DESCRIPTION OF DISCREPANCY	CPC #	PRIORITY	SIZE OF CHANGE
D29454	02/03/88	Description value containing a negative is improperly formatted.	B2	C	S
D29437	02/11/88	Password switch accepts <RETURN> as a good password.	A3	A	S
D29457	02/09/88	IO_Subsystem predicate is wrong.	C2	D	S
D29459	02/12/88	Output commands are incomplete and input commands are non-existent.	C4	B	L
D29402	02/12/88	A1 does not work with last fault handler change.	A3	B	S
D29452	02/02/88	PLT table does not reserve enough XX variable numbers.	C2	D	S
D29460	02/12/88	Stimulus Selections should not appear on the function menu in C3.	C3	D	M

Figure 1 Open Status Report

2) A Closed Status Report was generated on a monthly basis which identified the discrepancies that were closed each day. This report was generated daily during system test and was used to identify where retest could be started. It was also used to status against our page and line schedule for actual closure versus scheduled/promised closure. Figure 2 shows an example of the report.

3/1/88 **CLOSED STATUS REPORT** PAGE: 1

SARS #	OPEN DATE	PRIORITY	CPC #	DATE CLOSED
D44558	09/01/87	E	C1	09/01/87
D44560	09/01/87	D	C2	09/01/87
D44561	09/01/87	D	C2	09/01/87
D44563	09/01/87	D	C2	09/01/87
D44637	09/04/87	C	A3	09/04/87
D44553	08/31/87	D	C1	09/07/87
D44554	08/31/87	D	C1	09/07/87
D44556	09/01/87	C	C1	09/07/87
D44557	09/01/87	D	C1	09/07/87
D44604	09/01/87	D	C1	09/07/87
D44609	09/03/87	C	C3	09/07/87
D44620	09/05/87	C	B2	09/07/87
D44621	09/05/87	C	B2	09/07/87
D44622	09/05/87	C	B2	09/07/87
D44653	09/06/87	B	B2	09/07/87
D44660	09/11/87	A	A4	09/15/87
D44633	09/09/87	D	A3	09/16/87
D44658	09/11/87	C	C4	09/17/87
D44612	09/17/87	D	C4	09/17/87
D44613	09/17/87	D	C4	09/18/87
D44632	09/09/87	D	A3	09/18/87
D44659	09/11/87	B	A3	09/22/87
D44636	09/24/87	D	A3	09/25/87
D44616	09/04/87	C	A3	09/29/87
D44682	09/29/87	D	A3	09/29/87

Figure 2 Closed Status Report

3) A Monthly Status Report was generated which described the discrepancies that were written that month and the discrepancies that were closed. This report helped visualize progress as we approached the end of the program. When charted over the life of the program, it showed a convergence to a state where the error rate started decreasing to a level that gave us confidence that the product, when delivered, would have few undetected errors. An example of the report is shown in Figure 3.

3/1/88 **SARS OPEN/CLOSED STATUS REPORT** PAGE: 1

SARS #	OPEN DATE	PRIORITY	CPC #	ASSIGNED TO:	DATE CLOSED
D29452	02/02/88	D	C2	P3	02/09/88
D29454	02/03/88	C	B2	P3	02/17/88
D29420	02/05/88	B	A3	P2	02/05/88
D29457	02/09/88	D	C2	P3	02/10/88
D29434	02/15/88	D	A2	P4	02/15/88
D29465	02/15/88	D	B2	P3	02/15/88
D29479	02/18/88	C	A4	P5	02/20/88
D29491	02/19/88	D	C2	P5	02/19/88
A16596	02/24/88	B	C2	P3	02/24/88
A16604	02/25/88	A	C4	P5	02/26/88
A16611	02/26/88	D	A4	P5	
A16612	02/27/88	D	A4	P5	
A16614	02/27/88	B	A4	P5	02/29/88
A16617	02/29/88	B	A1	P2	02/29/88
A16618	02/29/88	D	A3	P4	02/29/88
A17305	02/29/88	C	C1	P5	

Figure 3 Monthly Status Report

4) A Priority Status Report was generated weekly during system test that identified the tools that had discrepancies that were open with a priority of "A," "B," "C," "D" or "E". Prior to system test, this report was generated on an as-needed basis. A discrepancy that had a priority of "A," "B," or "C" was fixed first. "A" had top priority then "B" then "C". The example report shows that we had serious discrepancies outstanding prior to our system test. These "A" discrepancies were not failures of the code to execute, but failure of the developers to have completed help files. Their "A" rating was related to non compliance with requirements. Figure 4 shows an example of the report.

SARS PRIORITY STATUS REPORT

WEEK ENDING **3/1/88**

SARS #	OPEN DATE	CPC #	PRIORITY
D29320	01/04/88	C2	A
D29322	01/04/88	C1	A
D29447	02/03/88	B2	A
D29482	02/18/88	A4	B
A17302	02/29/88	A4	C
A17305	02/29/88	C1	C
A17306	02/29/88	C2	C
A17309	03/01/88	C2	C
A17310	03/01/88	C2	C
A17311	03/01/88	C4	C
A17312	03/01/88	C4	C
A17314	03/01/88	C2	C
A17315	03/01/88	A3	C
A16611	02/26/88	A4	D
A16612	02/27/88	A4	D
A17308	03/01/88	C4	D
A17317	03/01/88	C3	D
A17318	03/01/88	C3	D
D29431	02/09/88	A4	E

Figure 4 Priority Status Report

5) A Workload Status Report was generated on a weekly basis during system test which identified the discrepancies that were open against each tool. Prior to system test, the report was generated on an as-needed basis. This report also identified the size of the fix: "small," "medium" or "large." Each tool was assigned to a specific engineer. With this report, management could view the workload of each engineer and re-assign priorities if necessary. For an example of the report, see Figure 5.

2/25/88 **WORKLOAD STATUS REPORT** PAGE: 1

SARS #	OPEN DATE	ASSIGNED TO:	CPC #	PRIORITY	SIZE OF CHANGE
D29401	02/03/88	P2	A2	C	L
D29420	02/05/88	P2	A3	B	S
D29452	02/02/88	P3	C2	D	S
D29457	02/09/88	P3	C2	D	S
D29475	02/17/88	P3	B2	D	S
D29496	02/20/88	P3	C2	B	S
D29460	02/12/88	P4	C3	D	S
D29418	02/03/88	P4	A3	C	M
D29434	02/15/88	P4	A2	D	S
A16597	02/24/88	P4	A3	A	M
D29427	02/05/88	P5	A4	C	S
D29433	02/08/88	P5	A4	C	S
D29431	02/09/88	P5	A4	E	L
D29479	02/18/88	P5	A4	C	S

Figure 5 Workload Status Report

6) A Monthly Metrics Report was generated to analyze the metrics scores of the programmers and re-assign workloads in accordance with these scores. It was found that tool fixes that generated no change in these scores introduced fewer errors than changes which caused fluctuation in the score. The tool name is identified by AX with X being the number of the tool. The programmer is identified by PX with X being a particular programmer. Tool size was measured in 1000 LOC. Lines of code were source code only and did not include comment lines or the mandatory commentary at the head of the programs. Figure 6 gives an example of the report.

TOOL/ PRGR	TOOL SIZE (K)	ERRORS/ TOOL/ 1K LOC	MOD. AVE LOC	AVG. BRANCH SCORE	Δ BRANCH	AVG. BOOLEAN SCORE	Δ BOOLEAN	Δ B&B	SYSTEM SCORE	Δ SCORE	HAL-STEAD	ΔH
A4/(P5)	19	6.8	42	.70	-.06	.91	-.02	-.08	4.21	-.10	.67	-.04
A3/(P2)	17	8.1	46	.79	-.01	.97	-.01	-.02	4.37	+.02	.70	+.02
A1/(P1)	8	9.1	49	.78	0.00	.96	0.00	0.00	4.33	+.02	.69	+.01
B2/(P3)	13	6.4	29	.82	-.01	.91	-.02	-.03	4.49	+.01	.70	+.01
B1/(P4)	8	5.7	33	.80	0.00	.95	0.00	0.00	4.35	-.03	.69	+.01
C4/(P5)	15	3.2	100	.81	+.02	.94	-.04	-.02	3.93	0.00	.57	-.05
C2/(P3)	12	5.3	34	.78	0.00	.93	0.00	0.00	4.38	+.15	.69	+.02
C3/(P4)	11	2.7	45	.78	0.00	.90	0.00	0.00	4.24	0.00	.67	0.00
C1/(P3)	9	6.5	33	.79	0.00	.93	0.00	0.00	4.28	+.02	.63	0.00

Figure 6 Monthly Metrics Report

Change Control Database

The change control database tracked the files changed on each discrepancy report and the number of lines of code changed. The structure of the change control database consisted of:

1) CPC_NO - The tool number was tracked to identify the files in the tool that were changed, the discrepancy report that changed the file, the revision of the files and the number of lines of code changed for that tool.

2) FILE_NAME - The name of the file was tracked to document the configuration of the file before the change and after the change.

3) LOC_ADDED - The number of lines of code added was tracked to identify the number of new lines of code that had to be added to correct the discrepancy.

4) LOC_CHG - The number of lines of code changed was tracked separately from the lines of code added.

5) REVISED_BY - The name of the person that corrected the discrepancy was statused.

6) OLD_REV - The version of the file that the fix was incorporated into was identified to track the configuration.

7) NEW_REV - The new version of the file after the build was performed was identified to document the new configuration.

8) SARSNUM - The number of the discrepancy report that the discrepancy was written against was tracked to identify the discrepancy reports that were in the configuration.

Change Control Reporting

1) An Engineering Change Status Report was generated which identified the revision level of the file for a particular tool. This report also identified the number of times a file was changed per discrepancy. This report was used in the analysis of measuring the programmers that were able to fix a discrepancy the first time. An example of the report is shown in Figure 7.

MONTH ENDING 2/28/88 **ENGINEERING CHANGE STATUS REPORT** PAGE: 1

CPC #	FILE NAME	LOC ADDED	LOC CHANGED	OLD REV	NEW REV	SARS #	CHANGED BY:
A4	A4_ed_screen	4	4	2	3	A14213	P5
A4	A4_ed_get_stuff.c	0	4	2	3	A14213	P5
A4	A4_ed_monitor.c	2	1	19	20	A16681	P5
A4	A4_ed_monitor.c	6	1	20	21	A16681	P5
A1	A1.c	-1	0	2	3	A14221	P1
A1	Position.c	21	9	2	3	A14221	P1
C1	C1output.c	0	3	14	15	A16608	P3
A1	scale_rotate.c	-14	55	9	10	A16645	P2
A1	A1_switches.c	3	1	12	13	A16644	P2
B2	B2.c	0	1	18	19	A16648	P3
A3	A3_database.c	31	1	17	18	A17315	P2
C3	C3_scenarios.c	1	0	4	5	A17317	P4
A2	A2_graphics_exec.c	1	2	2	3	D29312	P2
A2	A2_graphics_exec.c	0	3	3	4	D29312	P2
A3	A3_funcs.c	-24	1	5	6	AA2495	P2
A3	A3_funcs.c	1	1	6	7	AA2495	P2
A3	A3_funcs.c	-10	70	7	8	AA2495	P2

Figure 7 Engineering Change Status Report

Status Accounting For Discrepancy Reporting

The status accounting system encompassed the majority of the cost of quality data necessary for management reporting. Data that should have been collected in addition to the basic data was time to retest fixes and time for quality to verify builds and test fixes.

Numerous times during the change control process, a discrepancy was not fixed with the first attempt. The engineer had turned in the discrepancy report to Quality signifying the problem had been corrected and tested. Quality would perform the build of the tool with the revised files. Sometimes problems would be detected during the build process and the discrepancy would be sent back to the engineer. Sometimes the problem would not be detected until the test of the fix. The time that the engineer took to re-correct the problem and retest the fix was not tracked. The time that it took Quality to verify, build and test a change should have been tracked. In addition, the time it took Quality to rebuild and retest a change should have been tracked. To get an accurate picture of the time for correcting a discrepancy, this data should be tracked.

During the change control process, Quality performed numerous tasks. These tasks included checking files out to engineers for correcting discrepancies, verifying that the only change made to the software was approved, performing the build of the software with the automated CM system, and testing the tool to ensure that the discrepancy was indeed corrected and did not generate additional errors in the tool. The time for performing these tasks were not tracked in the status accounting system. To get an accurate picture of the total cost of quality, this data must be tracked.

Documentation Reviews

Many times when Quality reviews a document, numerous reviews are performed before the document is approved. To get an accurate picture for reviews of documents, the time for reviewing rewrites must be taken into consideration. When a software document is deliverable to a

customer, it goes through reviews. An engineer generates the document, the project reviews it for technical accuracy and Quality reviews it for format, technical accuracy and completeness. The document then goes to CM and data management prior to being delivered. The majority of the time Quality finds problems through their reviews and writes action items and the document gets updated. The document then goes back through project and Quality review again. Quality reviews the document for incorporation of open action items and that additional problems have not been incorporated into the document. This process may occur several times before a document is approved and sent to the customer. The customer may also have comments against the document and the document will be sent back through the review process again. To date, this data has not been collected. This data could be used when working with proposals. If this data was available for a particular document with a specific number of pages, the review time for documents could be more accurate than it is currently.

SOFTWARE CONFIGURATION MANAGEMENT

The software source existed in two areas as it evolved. The developers had a library system used to hold code during integration testing and Quality controlled a library for released code that was used for all formal testing and building deliverables. During implementation of the code, the individual programmers used their own directories to debug units. After their initial testing, the code was transferred to the integration libraries and tested by the software leads. Prior to formal test the leads released the code to the Quality library.

Both these libraries were similar in structure, though the integration library did not use a source code management system. The structure was such that simply renaming the logical path to the root directory allowed builds to be made using either area. Builds for formal test and delivery always used the Quality library. Software development builds used a combination of already released items in the Quality library, as well as unreleased code. Path names in the code were strictly controlled to prevent non library areas from being hard coded into the software.

The source code management system used in the Quality library was DSEE™, Domain Software Engineering Environment, a product of Apollo Computer Inc. This tool also gave an advantage for retesting changes before re-incorporation in the test baseline. Its ability to track software modules reserved for update allowed us to perform a temporary build before returning the updated code to the library. This in turn gave us the opportunity to retest the fix, evaluate the test results and then release the update to the library if it was acceptable and not have to rebuild the test configuration. If it was unacceptable the test configuration could be returned to its previous configuration by removing the updates from the test work directory and using the build command. Instead of recompilations, DSEE would use the last compilation of released code. The number of compilations saved were selectable. We chose to keep the previous five during our system acceptance testing. We kept all versions of source code.

As with any source code control system, this library structure did take up more disk space than the integration library which was not a DSEE library and only kept the most current version of source code. The DSEE library occupied 15% of the available disk storage, but this also included the executable test areas. This was equivalent to the integration library plus the developers' user areas.

Library Structure

The application was described in the functional description document as a loosely coupled tool suite of ten Computer Program Component (CPCs). Seven of these CPCs executed on Apollo™ workstations, two on a Symbolics™ workstation and one on a VAX ™. The library structure for the Apollo software followed this allocation by establishing seven CPC libraries under a root library given the name "RPS" which stood for Rapid Prototyping System. The CPC libraries were named to correspond with their designation in the program documentation. They were called CPC_1.1, CPC_1.2, CPC_2.1, CPC_2.2, CPC_3.1, CPC_4.2 and CPC_4.3. There were also two libraries created for common reused code. They were called "ins" for included files and "lib" for library

utility programs. Files in the "lib" library were "C" routines called by more than one other program. Files in the "ins" library, though, were data structure definitions, not programs. These were included as part of the compilation of the file using them, whereas, "lib" files were called by programs as part of their execution. The other three CPCs were controlled on their respective machines using status accounting techniques of version numbers, directory dates and comparisons against archived source.

The software configuration management system on the Symbolics had similar features to that on the Apollo, but the Symbolics lacked the storage resources to allow us to use them fully. The VAX portion of the software was previously developed software which had insignificant change traffic anticipated. Because of the established nature of the VAX application we did not take advantage of the power of the VAX configuration management system, CMS™ which would have given us similar tools as DSEE. Before the end of the program, Apollo had released additional functionality for DSEE that would have allowed us to control builds on the two remote machines, but due to time constraints, the instability of our ethernet interface and conservatism we did not pursue its use for such change control. The library structure is shown in Figure 8.

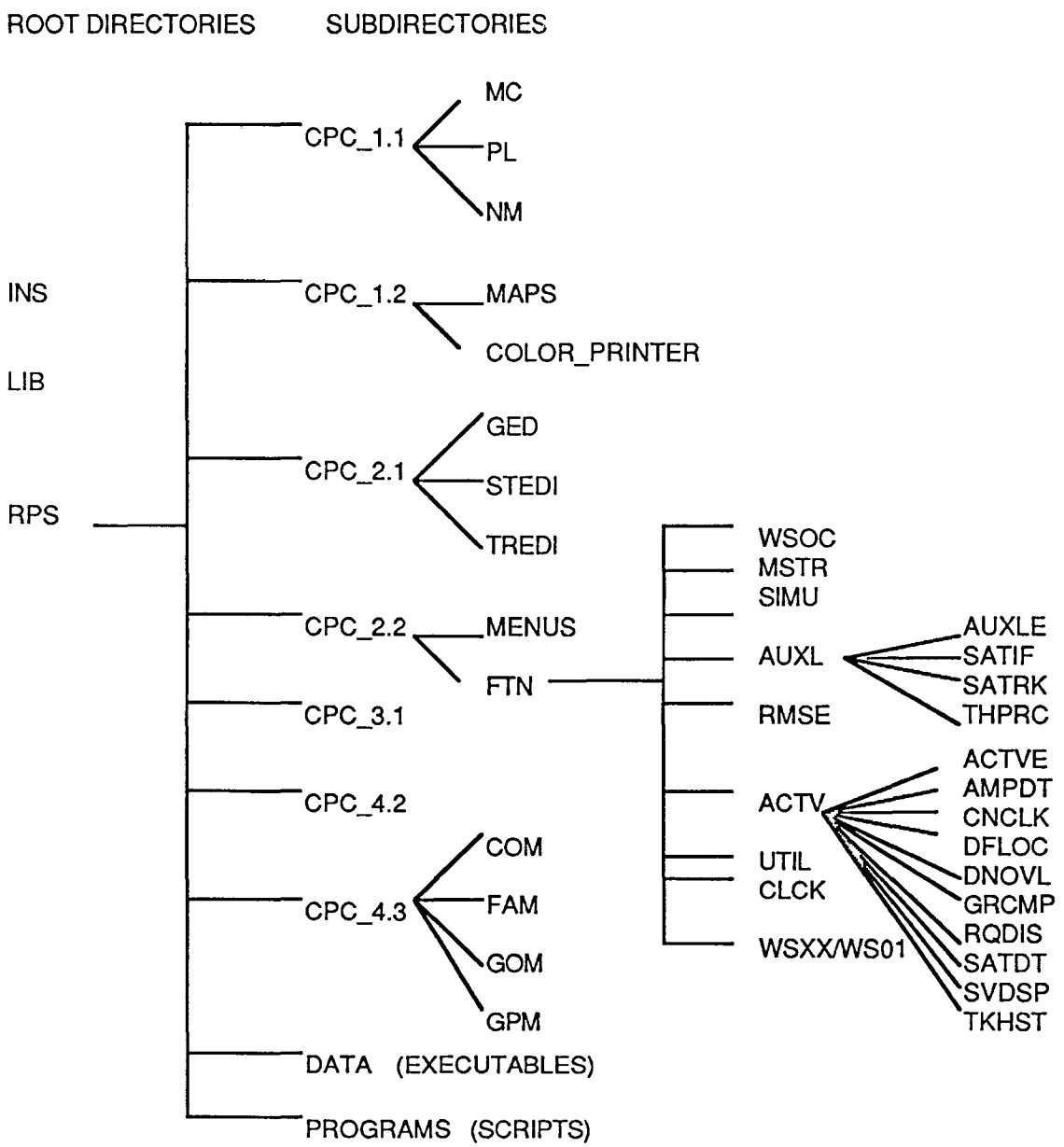


Figure 8 RPS Library Structure

A few of the Apollo CPCs had more than one major capability identified in the requirements documentation. It was decided during the design phase to have sublibraries reflecting this. DSEE supports the creation of nested libraries with protection of all files and libraries defined by the library administrator. Files in the libraries were protected against writing by anyone other than the librarian but allowed program members to read the file and any previous versions.

Our library also contained the structure for the executables under a directory called "data". Each of the tools used a large number of graphic displays for obtaining user inputs and displaying help or other messages. As part of the locating strategy it had been decided to place these in a reserved area for access by the executables. This established a naming convention to be used by the programmers that prevented their hard coding path names to their own areas. This also became the area that the DSEE build tool maintained. The subdirectories under data contained all the menus/displays for each tool, while the data directory contained the executables for the tools.

This structure also supported tape production for delivery of only executables for our prototypes. We made deliveries of executables to multiple sites, but only our customer, RADC, received the DSEE source libraries also. Our system does not require source code to be resident with the application, but it duplicates all data files used for execution of the system.

Conventions

The development conventions followed included:

1. Naming convention for files

   xxx_yyyyyy where xxx is a mnemonic for the CPC and yyyyyy is a meaningful name signifying the functions addressed in the file.

   These mnemonics were:

   mc - CPC 1.1 Master Control Executive
   pl - CPC 1.1 Process Linker
   nm - CPC 1.1 Notification Manager
   map - CPC 1.2 Map Generation
   ged - CPC 2.1 Graphic Editor
   st - CPC 2.1 Switch Table Editor (STEDI)
   tr - CPC 2.1 Tray Editor (TREDI)
   cgb - CPC 3.1 Connected Graph Builder
   sg - CPC 4.2 Scenario Generator
   com - CPC 4.3 CM Translator
   fam - CPC 4.3 FM Translator
   gom - CPC 4.3 OM Translator
   gpm - CPC 4.3 PM Translator

   This was generally followed in the CPCs, though some files of a generic nature produced before the start of this contract did not follow this convention.

2. All code references to location for finding data files or executables refer to a '//rps_node/rps' path. Which can be set at the system level to represent the location of either the development integration library or the controlled DSEE library. The path to the DSEE library was established by setting "//rps_node" to "//dsee_node/dsee".

3. Shell scripts that called executables were located in '//rps_node/rps/programs'.

4. The executables were located by function in '//rps_node/rps/data'. That is, 'start_stedi' the shell script for starting the STEDI tool was located in programs and the executable referenced was in '//rps_node/rps/data/stedi'.

5. The Apollo applications software source was distributed amongst CPC and common libraries. The CPC libraries were in the path '//rps_node/rps/cpc_x.x', where x.x is one of 1.1, 1.2, 2.1, 2.2, 3.1, 4.2, 4.3.

6. The common libraries were '//rps_node/ins' and '//rps_node/lib'.

7. Changes to files were divided into the four steps listed below:

   A. The person responsible for configuration management of the RPS software reserved the needed file(s).

   B. The maintenance programmer updated and tested the changes in a nonproduction area.

C. The person responsible for configuration management replaced the file(s).

D. The person responsible for configuration management used the DSEE build files to rebuild into the production area.

Delivery Area

It was necessary to establish the Quality library system very early in the program because of the early delivery of prototypes to our customer. The first build attempt for testing our first prototype demonstrated that building from the development area was a catastrophe. Even though great care had been taken to gather all files into the integration area by the software leads and direction had been given to avoid hard coded pathnames, the build was a failure. We discovered this when the build failed to execute once it was physically isolated from the user areas.

We had at first thought that a software development controlled library would suffice up to the point of acceptance testing for the final delivery. But after the experience of the first prototype build we decided a Quality controlled library would be an advantage. A controlled library supplied two advantages, one by providing a librarian responsible for the builds instead of diverting a developer and the other by establishing release procedures that the programmers could get used to before the usual panic at the end of the program.

For our release mechanism we wanted ease of entry into the library for new functionality to support block releases of new requirements while still controlling change to already tested and released functionality. DSEE made it easy to accept large numbers of files at one time for batch submittals for library creation and updates, since it responds to command files as input as well as interactive commands. The new libraries created matched the software development structure for the initial release and used a feature of the tool to tie a block release to a name. The initial release was performed for an Initial Operating Capability (IOC) delivery so it was named IOC-1. When we were preparing for IOC-2, all new files were accepted as initial release but files already existing had was/nows generated so that changes could be evaluated against the proposed updates described on our SARS form. It was standard procedure for both releases that the code was reviewed for adherence to programming standards and implementation in accordance with its program design language.

Change Control

Consideration was given to allowing check out by the programmers to update files, but it was decided that this was more a function of a development library. When problems were identified in the released software, a SARS form was written to chronicle the problem, the test environment and priority of the fix. Our change board was used to come to agreement on what tool contained the error, the needed fix and who should do it. The person responsible for the fix received the SARS after change board review, attempted to duplicate the problem from the description, did analysis of what modules were effected, estimated the time to fix the problem and requested files from the librarian. Both the estimate to fix and the files needed were annotated on the SARS form. The librarian would then reserve the current version of the released source code into a designated work area, mark the SARS with the version number of the code and return it to the responsible programmer.

As part of the reserve mechanism, DSEE requests annotation of need for the file. The librarian would enter the SARS number, thereby, establishing a tracking from a specific version change to a specific problem. This was useful in identifying change traffic in our system activity reports.

In our library system we established two directories for change traffic. One was called dsee_retrieve, the other dsee_update. Dsee_retrieve was used by the librarian as a repository for reserved code that needed fixing. Dsee_update was used by the programmer to return updated code. When a programmer put code into dsee_update, the SARS related to the code

```
DEFAULT FOR ?*.PAS =
      TRANSLATE
          /COM/PAS %SOURCE -opt -b % result
          cpf %result.bin %source({?*}.pas, %leaf).bin -r
      %DONE;
END OF ?*.PAS;

DEFAULT FOR ?*.C =
      TRANSLATE
          /COM/CC %SOURCE -opt -b % result
          cpf %result.bin %source({?*}.c, %leaf).bin -r
      %DONE;
END OF ?*.C;

DEFAULT FOR ?*.FTN =
      TRANSLATE
          /COM/FTN %SOURCE -opt -b % result
          cpf %result.bin %source({?*}.ftn, %leaf).bin -r
      %DONE;
END OF ?*.FTN;

DEFAULT FOR ?* =
      TRANSLATE
          catf %source > //rps_node/rps/data/%source(%leaf)
      %DONE;
END OF ?*;
```

This ensured that the default action for Pascal, C and Fortran source was compilation, followed by replication of the binary file into the working directory. By copying the binary, we made visible to the developers the most current version of all software that they might bind into their own executables during informal testing. DSEE kept its own copy of the compilation unit in a binary pool identified in the build file.

The RPS tools were divided into DSEE aggregates that corresponded to major functionality of the RPS tools. The following is a list of these DSEE aggregates:

1. CGB - the Connected Graph Builder.

2. CS - Color Printer Routines.

3. FTP - File Transfer Protocol shell scripts.

4. GED - the Graphic Editor.

5. Master - a set of executive functions which is bound with all demos produced in the RPS.

6. INSERT_FILES - contains include files that are needed to provide functionality to user defined demonstrations.

7. MAPS - a tool which accesses World DataBase I to draw maps of the world. The RPS applications that perform extraction, GED formatting, merging of features, metafile creation, projections, scaling, and translation included in the Support Tools.

8. MAPPER - Contains the control functions for the user interface to the Maps tool.

9. MCDATA_HELP - Contains the Notification Manager help files.

10. NEW_LIB - the common library of RPS routines which is bound into a 'userlib.private' executable that is loaded at boot time by the Apollo operating system. Its source resides in the "lib" library.

11. NMREFLIST - This produces the Notification Manager Reference List (NMRL) demonstration used in the CGB for adding, changing and deleting NMRL entries.

12. Preproc - the Preprocessor that prepares user switch table files into a "C" file that is compiled and bound with Master for the user demo.

13. PRINT_PROC - A program used by the Notification Manager to list the current processes of a CGB user.

14. RPS_HELP - Contains the help files for the higher level Master Control functions.

15. RPS_MC_DEMO - The main executive program, which is a RPS defined demo.

16. RUN_SCRIPT - A program used by the Notification Manager to start processes.

17. SG.GO - the Scenario Generator which is a RPS defined demo.

18. SG_HELP - contains the help files for the Scenario Generator.

19. STEDI - the Switch Table Editor.

20. STEDI_MENUS - the screens used by STEDI.

21. TRANSLATOR - the Apollo user interface for obtaining performance model data. It is a RPS defined demo.

22. TREDI - the Tray Editor.

Many of the RPS tools were RPS demonstrations and had menus and help files associated with them. These were distributed into the DSEE aggregates:

COM_MENUS	COM_HELP
FAM_MENUS	FAM_HELP
OM_MENUS	GOM_HELP
GPM_MENUS	GPM_HELP
STEDI_MENUS	TRANSLATOR_HELP

An example syntax for one of these aggregates follows:

```
AGGREGATE translator =

    DEFAULT FOR ?*.C =
        TRANSLATE
            /COM/CC %SOURCE -opt -b %RESULT
            cpf %result.bin %source({?*}.c,%leaf).bin -r
        %DONE;
    END OF ?*.C

    DEFAULT FOR ?* =
        TRANSLATE
            catf %source > //rps_node/rps/data/translator/%source(%leaf)
```

```
        %DONE;
    END OF ?*;

    TRANSLATE
    /sys/d3m/ind3m
    /com/bin - <<!
    %result_of (?*.c).bin
    //rps_node/rps/data/mc_data/master
    -b //rps_node/rps/data/translator/tran
    -end
    !
    %DONE;

    DEPENDS_RESULT
        MASTER;
        ELEMENT translator.c @cpc_4.3 =
            DEPENDS_SOURCE
                tran.ins.c @INS;
                com.ins.c @INS;
                fam.ins.c @INS;
                gom.ins.c @ins;
                gpm.ins.c @INS;
        END OF translator.c;
        ELEMENT run_model.c @cpc_4.3 =
            tran.ins.c @INS;
            objects.ins.c @INS;
        END OF run_model.c;
        .
        .
        .
END OF translator;
```

This block if called with the DSEE build command "build translator" would compile all the C source ELEMENTS listed under the DEPENDS_RESULT that had changed since the last build, bind them with all other binaries from previous builds along with an executable called "MASTER" into an executable called "tran" and place it into the "//rps_node/rps/data/translator" directory. Within each ELEMENT block are listed source files referenced by the program. These are identified when using the make model utility of DSEE. If they are not listed in the build file but referenced in the program, a warning will be issued at time of compilation but the build tool knows to use them. Because warnings and errors are reported to the screen or a designated build file, the operator can status the build and correct the build file with any new references. If an error occurs in any of the actions taken within an aggregate, the build tool notifies the operator that the build was not completed. It will complete as many individual actions as it can but not the final bind to create the executable. After problems are corrected, the next build request will only recompile those elements that failed previously. A complete rebuild can be forced, in the case of our system a complete build ran 12 hours with no other users on the system. We preferred the advantage of only rebuilding changed files.

During test, the test conductor with Quality concurrence decided if an operational work around for a problem could allow continuation of the test. If yes, the test continued and software support would be given the SARS to tackle using the development library and test area. Fixes would be gathered to support retesting of related changes and the DSEE tool used to rebuild the system tagging the updated code with the SARS form numbers and the build with a version number. By tagging these builds it was possible to recreate any test version of the system by referencing the version number. A listing of this build could also be produced that would show all the versions of source code and system files used as well as the SARS fixes included in the test baseline.

Tagging the builds was also helpful in the few cases where a change induced unwanted consequences in the tool that necessitated roll backs to previous versions.

The build tool also aided us when we decided to upgrade our operating system. This occurred twice during the two year life of the program. After preparing a new system disk, we used the build file to report on all the recompilations needed. This gave us an immediate idea of the impact from changing the system and pointed out dependencies we would not have caught from the release notes on the update. The build file was also a convenient place from which to start to identify files that would need rework due to the changes described in the release notes.

We also found the build tool helpful when bugs were found in the vendor software. On two occasions Apollo sent us interim releases of system libraries which we were able to locate in our data directory then call from our build file for use in specific tools that needed these fixes without intermingling them with the system libraries. This gave us a documented reminder that we had some interim release software that another update of the system libraries might have wiped out if we had put it in the system library.

CONCLUSION

The discrepancy tracking and change control implementation on the program supported two important quality assurance jobs:

> Verification that all deficiencies in, and changes to released software and associated documents are documented and all corrective action is completed;
>
> Reporting that reflects the results of trend analysis.

Software configuration management supported quality assurance by providing software library controls and current documented baseline configuration, as well as the ability to generate previous configurations.

On our next programs the tools have changed but these basic tasks still remain. The programming language changes from "C" to Ada™, the contract requirements move from research and development minimum controls to NASA life cycle requirements and the quality role expands accordingly. Whether there are 12 or 60 items tracked in the discrepancy system, one or many Computer Program Configuration Items (CPCIs), the need still exists for discrepancy reporting and software configuration management. Automated tools for database management and source code control should be sought out to perform quality assurance tasks as the best means of supplying accurate data, especially since human resources are always limited.

Paper 3-A-1

USING SOFTWARE METRICS AND PROCESS MATURITY TO ASSURE SOFTWARE QUALITY

Dr. Shari Lawrence Pfleeger
CONTEL Technology Center

Dr. Shari Lawrence Pfleeger is head of the Software Metrics Project at Contel Technology Center's Software Engineering Laboratory. She consults nationwide on software engineering, computer security, and other aspects of software systems development. Dr. Pfleeger is the author of numerous research papers in mathematics and computer science and of two university textbooks. She is a member of the ACM, IEEE Computer Society, and Computer Professionals for Social Responsibility.

Using Software Metrics and Process Maturity to Assure Software Quality

Shari Lawrence Pfleeger

Contel Technology Center

Chantilly, Virginia

Software Engineering Laboratory

Process and Metrics Project

Presentation to Quality Week 1990

Overview

- Contel's Process and Metrics Project
- Goals of Metrics and Process Maturity
- Process Maturity Levels and Associated Metrics
- Steps to Take in Using Metrics
- Expected Benefits

Purpose of Process and Metrics Project

To aid software development by helping to

- improve the productivity of the development teams
- improve the quality of the software produced
- provide greater visibility for the process
- provide greater control over the process
- manage risk

Objectives of Process and Metrics Project

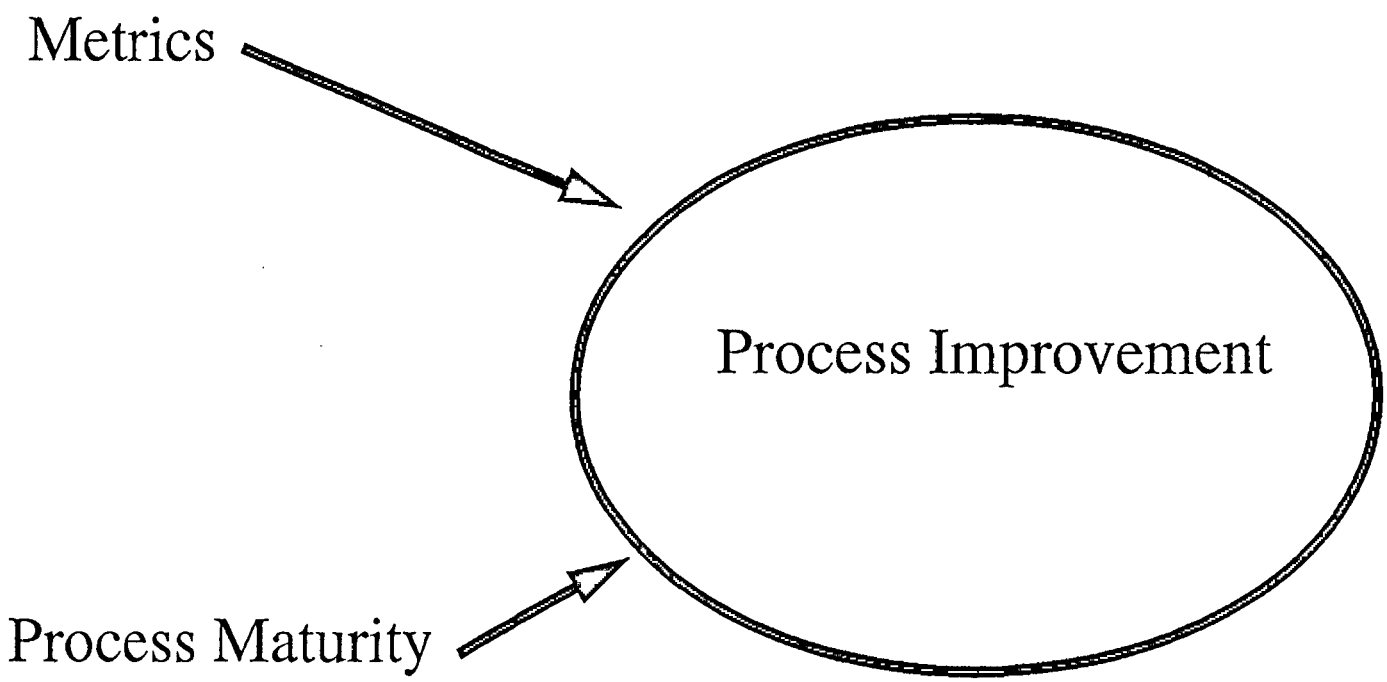
- To determine the amount and kind of software development being done at Contel (what are we doing?).
- To characterize the approaches to software development and the environments in which development is being done (how are we doing it?).
- To keep abreast of methods, tools and techniques that can improve productivity and quality.
- To transfer that knowledge to the business units, where appropriate.

Overall goal: Quality improvement through process improvement

Steps to improvement:

1. Provide framework and incentive for metrics collection and analysis
2. Recommend initial metrics set, begin pilot projects
3. Training about setting up and using metrics
4. Provide "metrics toolkit"
5. Build corporate historical database
6. Monitor status and set improvement goals

Common Goal



Metrics Recommendations: Five Stages

- Business goals, organization and commitments
- Initial set of recommendations: August 1989
- Pilot projects
  - Are metrics definitions realistic?
  - Are metrics definitions applicable to all projects?
  - Are data collection and analysis an unreasonable burden for the projects?
- Analysis of results
- Final recommended metrics set: Mid-1990

Process Maturity and Metrics

Level	Characteristics	Measurement
5. Optimizing	Improvement fed back to process	Process + feedback for changing process
4. Managed	Measured process (quantitative)	Process + feedback for control
3. Defined	Process defined, institutionalized	Product
2. Repeatable	Process dependent on individuals	Project
1. Initial	Ad hoc/chaotic	Preliminary project (baseline)

Initial Metrics Set

Tied to process

- What is a process maturity level?
- How mature is my process?
- How can I improve (and possibly move up in the hierarchy)?

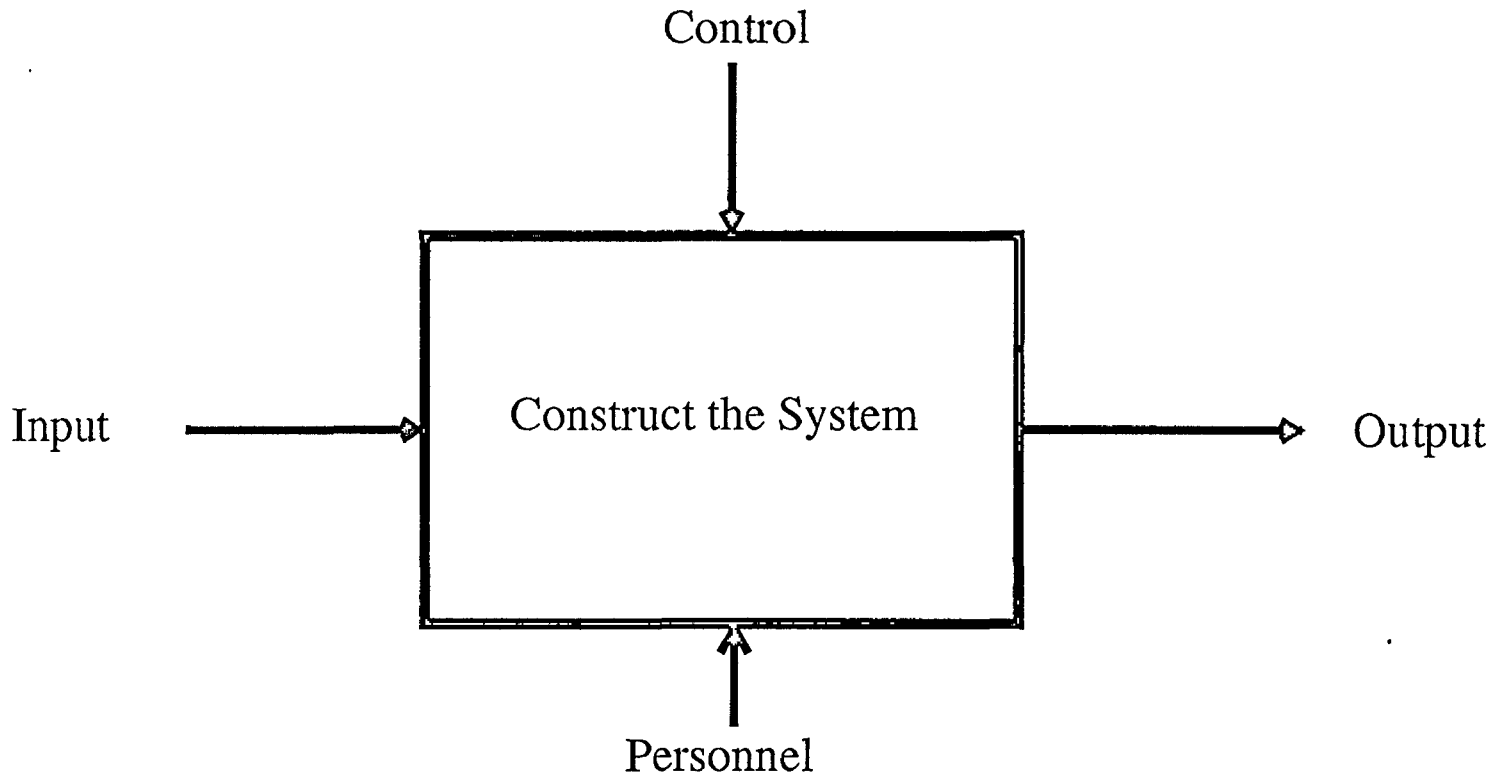
Level 1: Initial (ad hoc/chaotic)

Baseline metrics collected:

- Product size
- Staff effort

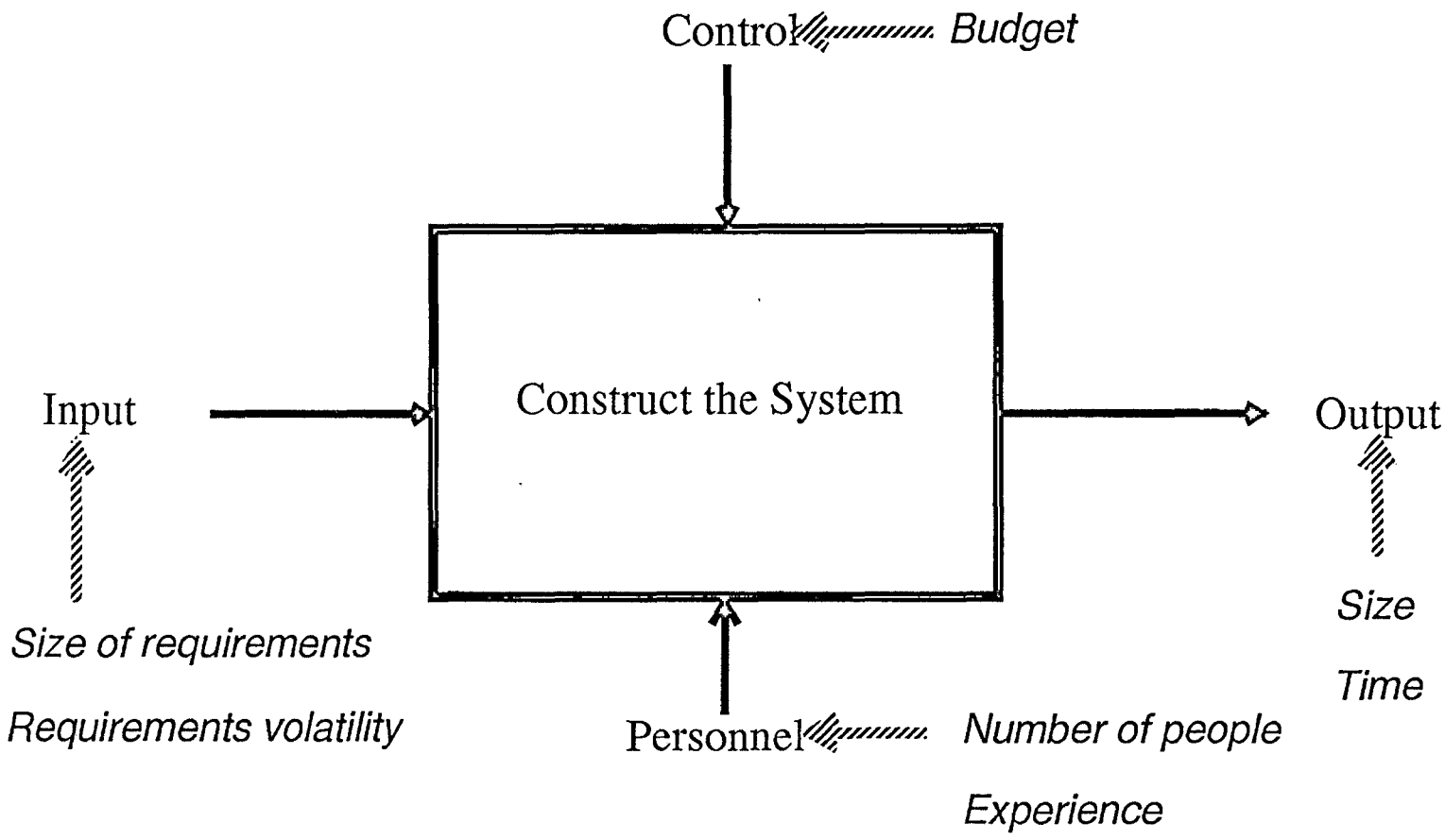
General focus: Structure and control

Level 2: Repeatable (process dependent on individuals)



. . . project management metrics

Metrics for Level 2



Level 2: Project Management Metrics

- Software Size

  Non-commented source lines of code

  Function points

  Object and method count

- Effort

  Actual person-months of effort

  Reported person-months of effort

- Requirements Volatility

  Requirements changes

Possible Additional Project Management Metrics

- Experience

  -- with domain/application

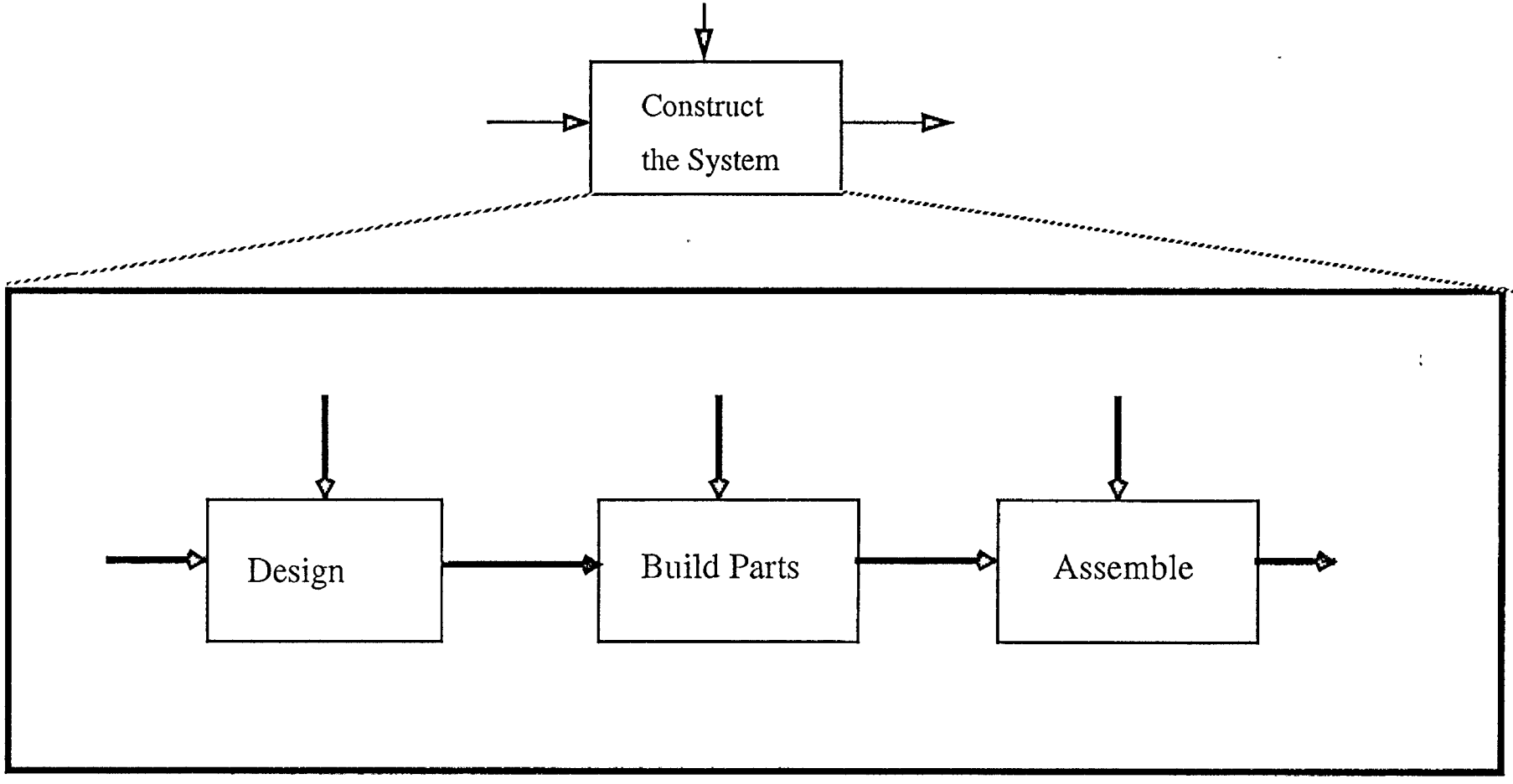
  -- with development architecture

  -- with tools/methods

  -- overall years of experience

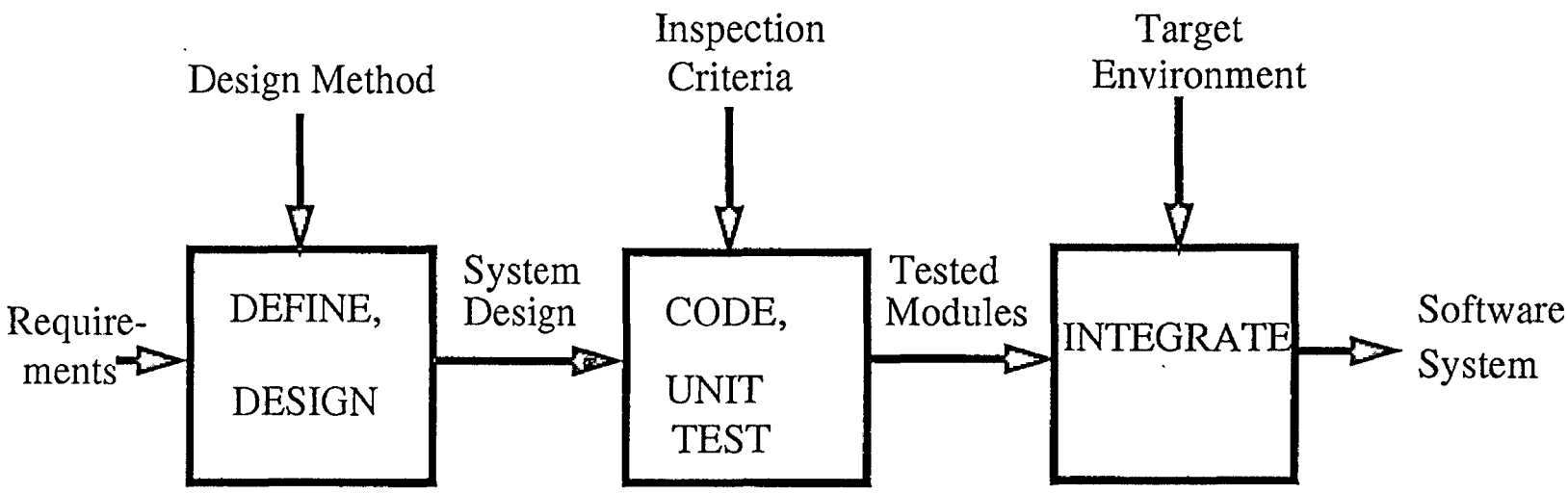
- Employee turnover

Level 3: Defined (process defined and institutionalized)

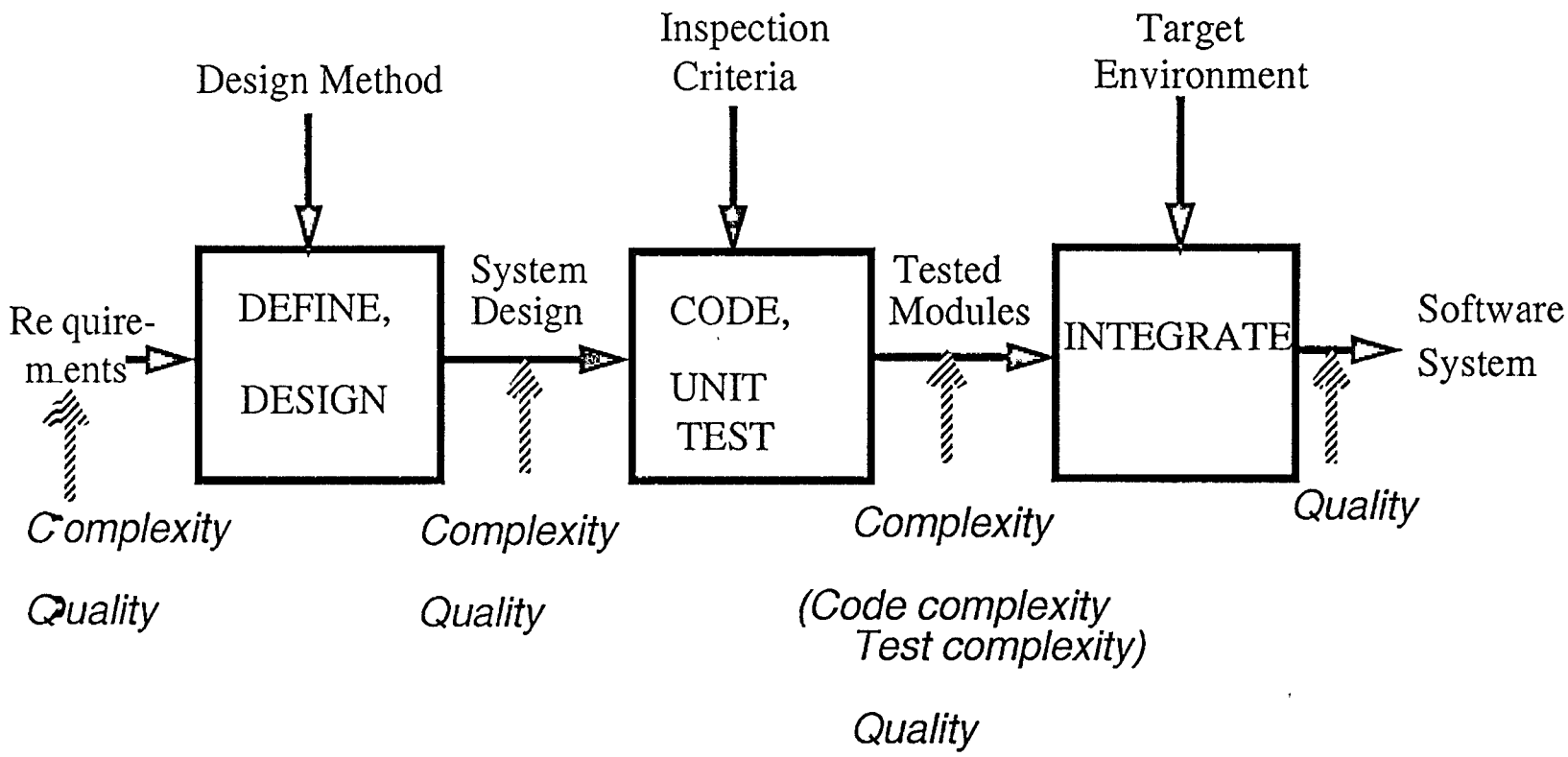


. . . product metrics

Example of a Level 3 Process



Metrics for a Level 3 Process



Level 3: Product Metrics

- Complexity
- Quality

Level 3: Product Metrics -- Expanded

Complexity:

- Requirements complexity

  Number of distinct objects and actions

- Design complexity

  Number of design modules

  Cyclomatic complexity

  McCabe design complexity

- Code complexity

  Number of code modules

  Cyclomatic complexity

- Test complexity

  Number of paths to test

  If object-oriented development, number of object interfaces to test

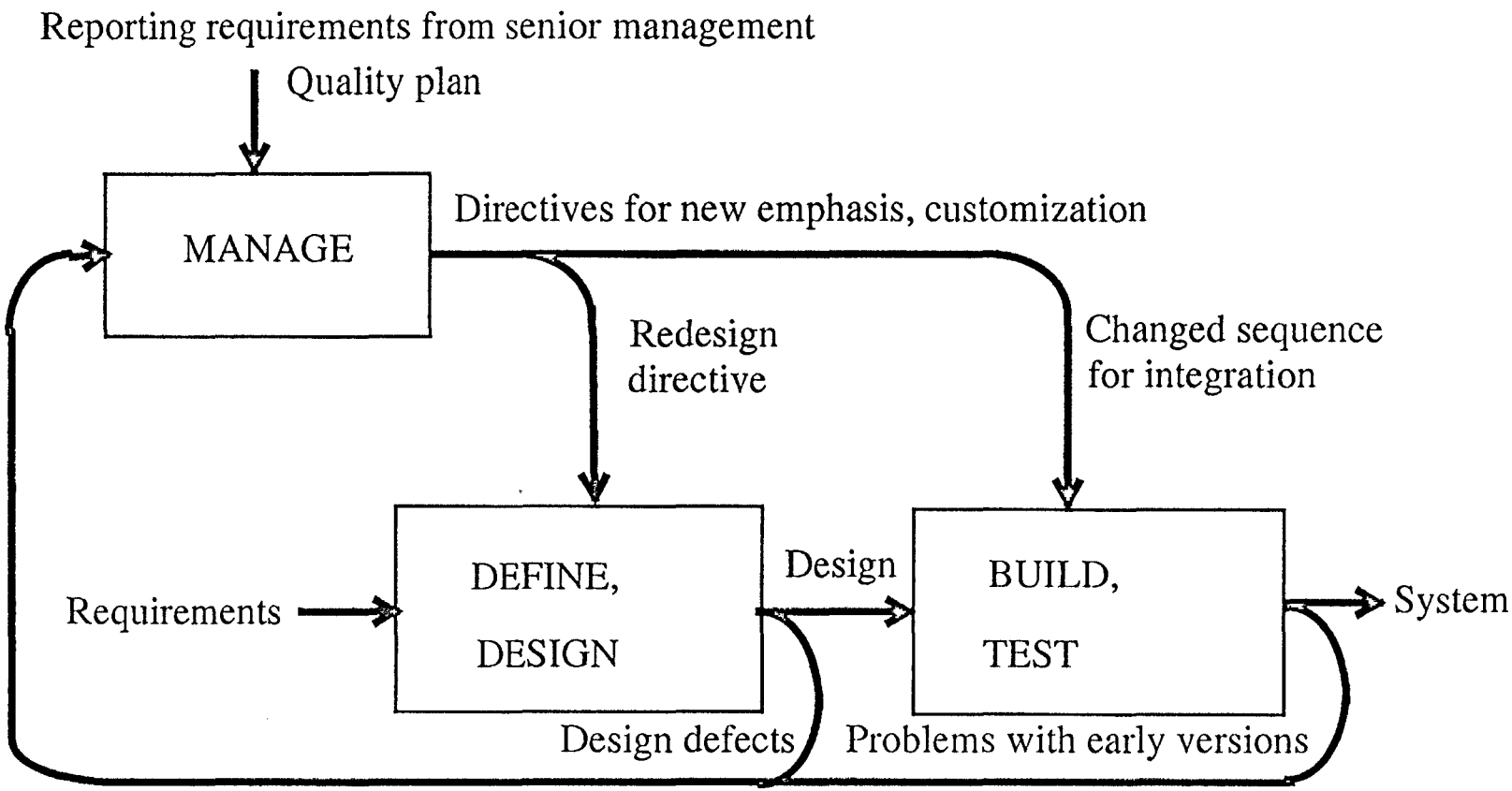
Quality:

- Defects discovered
- Defects discovered per unit size
- Requirements faults discovered
- Design faults discovered
- Code faults discovered

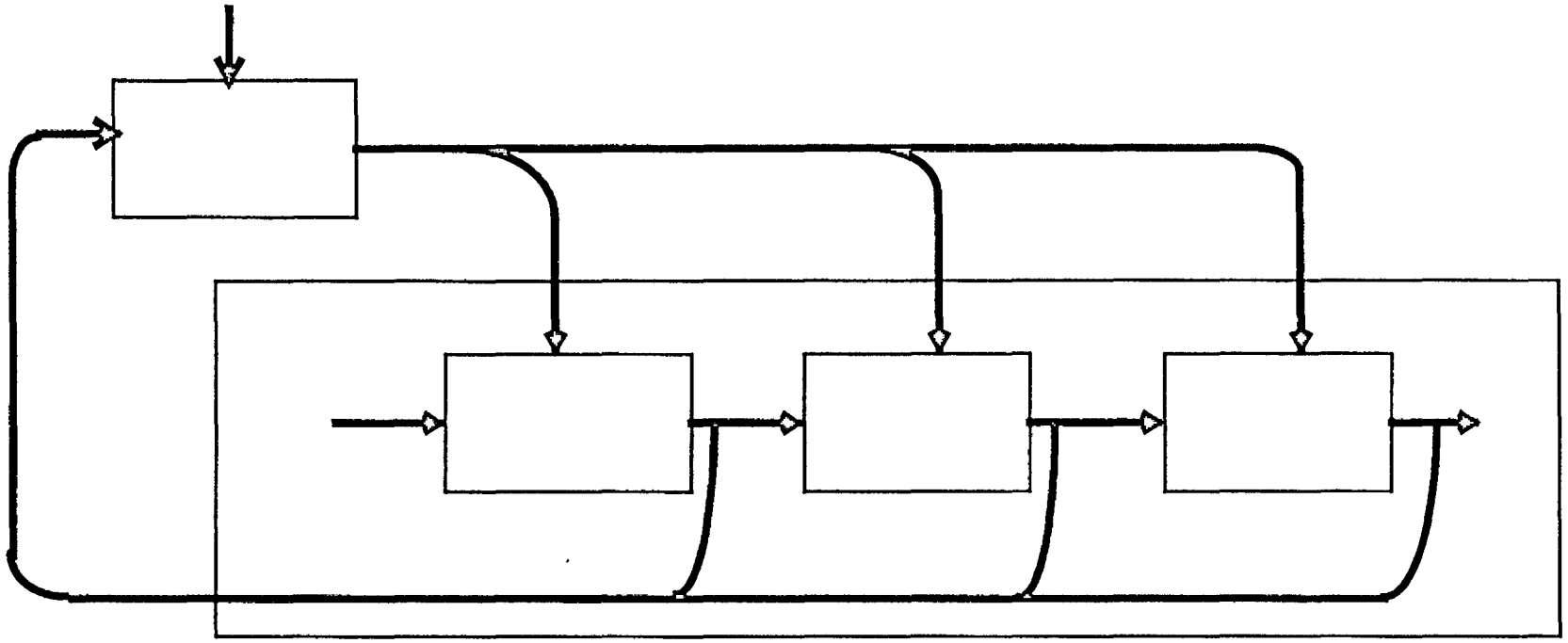
Possible additional metric:

- Pages of documentation

Example of Level 4 Process

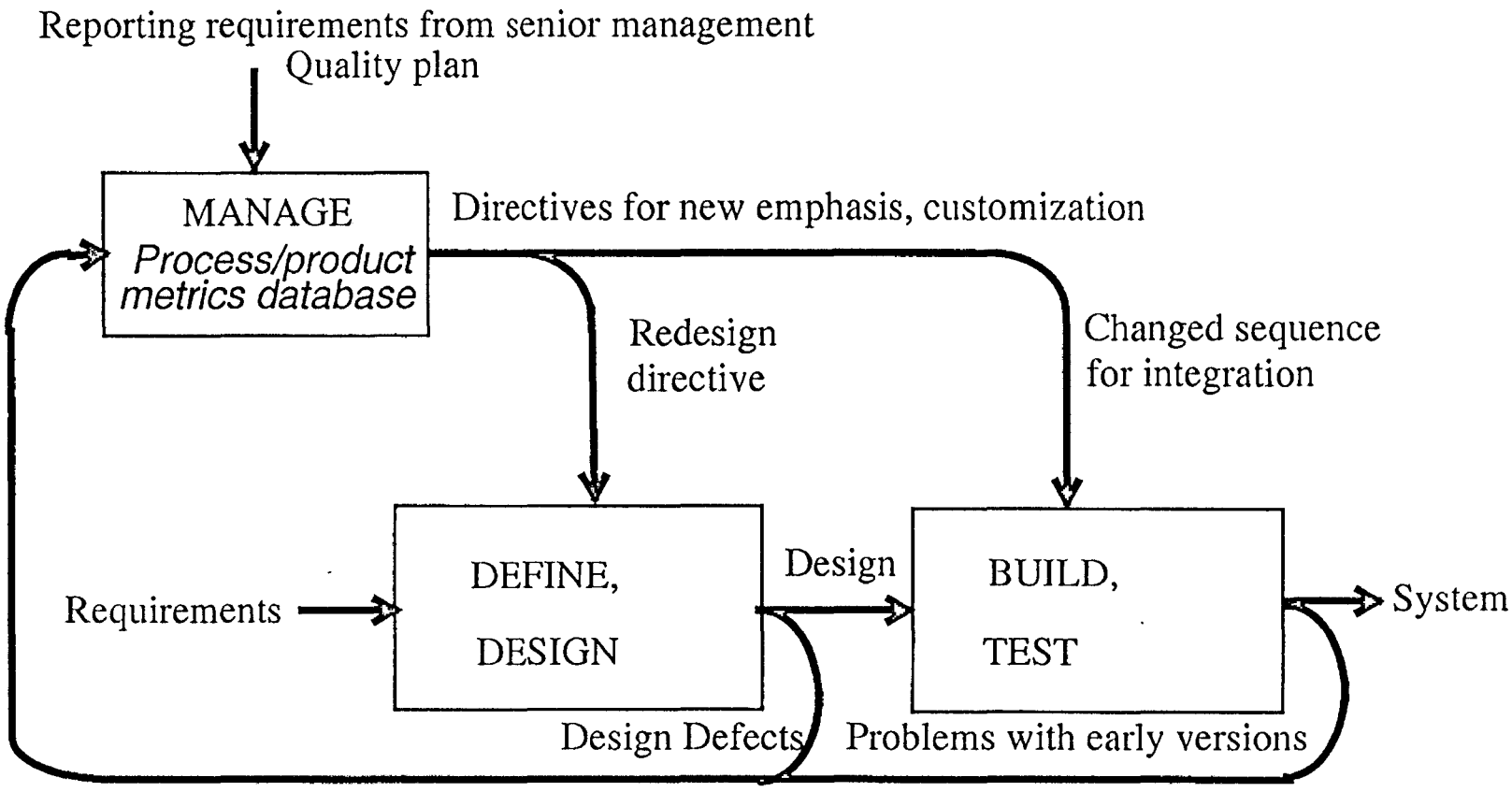


Level 4: Managed (measured process - quantitative)



. . . process metrics with feedback

Metrics for a Level 4 Process

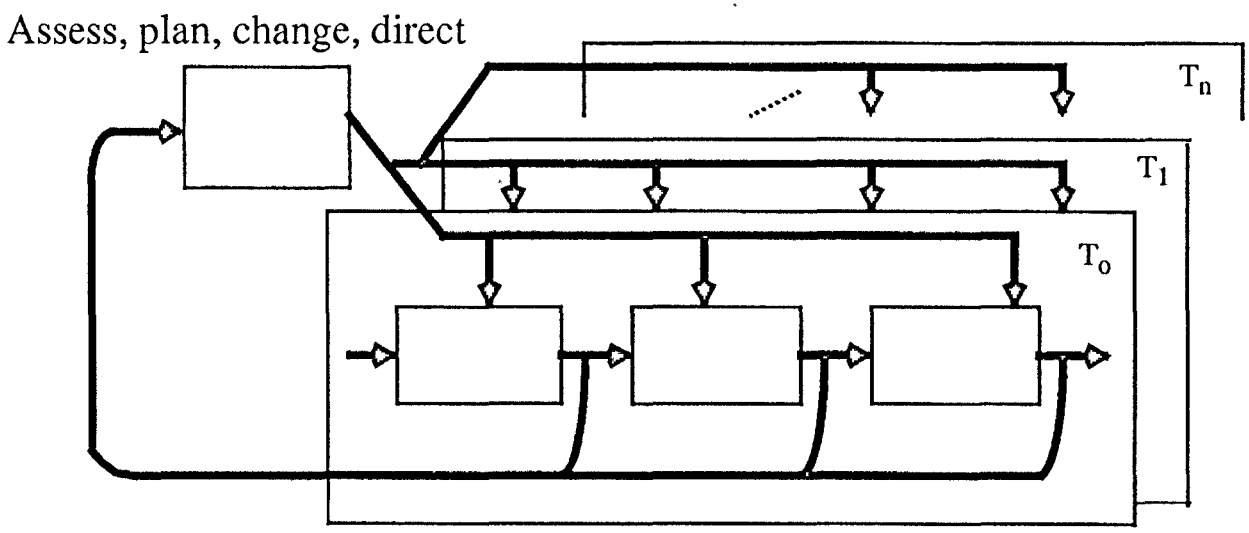


Distribution of defects, productivity of tasks, plans vs. actuals, resource allocations

Level 4: Process Metrics

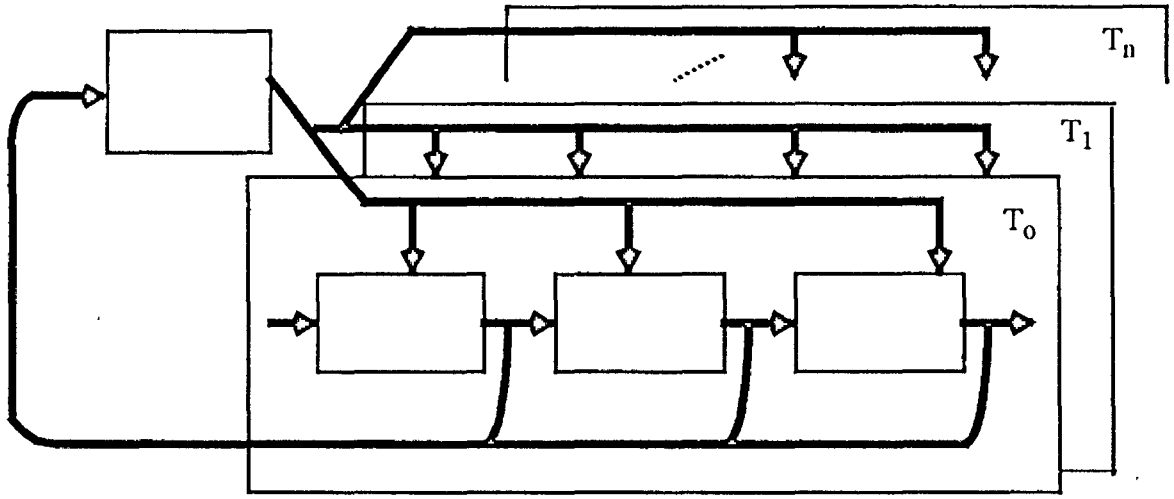
- Process type
- Amount of producer reuse
- Amount of consumer reuse
- Defect identification
- Use of defect density model for testing
- Use of configuration management
- Module completion over time - identified, designed, coded, tested, integrated

Level 5: Optimizing (improvement fed back to process)



. . . process metrics with feedback for control

Metrics for a Level 5 Process



Measure IMPROVEMENT of use of one process over another
Measure EFFECTIVENESS of techniques, tools, methods

Software Engineering Institute:

113 software development projects surveyed

- 85% at level 1
- 14% at level 2
- 1% at level 3
- 0% at levels 4 or 5

No level 3 ORGANIZATIONS!

Therefore, concentrate on

- implementation of metrics at levels 1, 2 and 3
- understanding role of metrics in levels 4 and 5

Steps to Take in Using Metrics

1. Assess the process
2. Determine the appropriate level of metrics collection
3. Recommend metrics, tools, techniques
4. Estimate project cost and schedule
5. Collect appropriate level of metrics
6. Contruct project database
7. Cost and schedule evaluation
8. Evaluate productivity and quality
9. Form a basis for future estimates

Expected Benefits

- Enhanced understanding of the process
- Increased control of the process
- A clear migration path to more mature process levels
- More accurate estimates of project cost and schedule
- More objective evaluations of changes in technique, tool or method
- More accurate estimates of the effects of changes on project cost and schedule
- and ***HIGHER QUALITY PRODUCTS***!

Paper 3-A-2

USING METRICS MEASUREMENTS TO ALLOCATE TESTING RESOURCES

Mr. Warren Harrison
SET Laboratories, Inc.

Dr. Warren Harrison is an Assistant Professor of Computer Science at Portland State University. He has been involved in software measurement for over ten years, and has published the results of his research extensively. He helped organize and is the current chair of the Annual Oregon Workshop on Software Metrics and has served as an editor for special issues of the Journal of Systems and Software on software metrics. He is a frequent speaker at conferences and workshops, and has advised a number of organizations in their use of software metrics. He has held positions at Bell Laboratories and Lawrence Livermore National Laboratory. Warren holds a Ph.D. in Computer Science from Oregon State University.

Using Metrics to Allocate Testing Resources in a Resource Constrained Environment

Warren Harrison
Department of Computer Science
Portland State University
Portland, OR 97207-0751
warren@cs.pdx.edu

Abstract

While many people have suggested innovative ways to help test computer software, there has been little, if any attention paid to *resource constrained testing environments* in which the amount of resources and/or calendar time is tightly constrained. In this paper we have described and evaluated a metric-based technique for allocating constrained testing resources among the components of a software system. Based on our evaluation of a single software system, we conclude that Software Science Effort is perhaps the best basis upon which to allocate testing resources, though the LOC- and Cyclomatic- based techniques also exhibited reasonable performance. To help make the technique more accessible to users, we also describe a novel technique for separating the code analysis and metric generation activities.

Introduction

Software development, as with any complicated human activity, offers numerous opportunities to make mistakes. To help alleviate the impact of programming errors, virtually all organizations have adopted a phase of the development process called *software testing*, which typically begins after the software has been programmed, and before it is released to the marketplace.

The goal of the software testing phase is to find as many errors as possible in the product before it is released to the customer. This activity is highly labor intensive since most test sets require not only the test data to be developed, but also an explanation of the behavior which is expected from program being tested, for each piece of test data.

Numerous techniques to engineer specific sets of test data which improve one's chances of finding errors have been proposed. However, the single critical factor which is missing from most of this research is the practical consideration that most testing budgets/schedules are fixed, and often cannot be enlarged or extended. For example, one of the simpler testing techniques is called *statement coverage*. The goal behind this technique is to guarantee that every statement in the program is executed at least one time by the selected test data. Unfortunately, if due to budget and/or schedule constraints, it is not possible to engineer sufficient data to execute every statement, the technique fails to offer advice as to which statements should be executed and which should not to get the most benefit out of those test cases which *are* developed.

We refer to such an environment as a *resource constrained testing environment*. Such budget and schedule constraints are quite common. Budget constraints for the testing phase are encountered for the same reason that budget constraints are encountered during the design and/or coding phases. A lack of a clear definition of the problem when the budget was originally formulated,

coupled with constant changes to the requirements during development make it difficult to accurately budget for the testing activity. Schedule constraints are often caused by artificial marketing windows such as tradeshows and/or competitors' products.

The primary difference is that the design and coding phases are not the last activities in the development process. Thus, any slack which *may* have been in the budget will usually be quickly soaked up by these activities. Further, because it is difficult to argue with the reasoning that "without the software, there is nothing to test", many product managers will "borrow" schedule and resources from the testing activity to make sure design and coding get completed within the overall budget and schedule. This reasoning is validated by the recent experience of several commercial software vendors who significantly delayed delivery of new versions of their flagship products to extend the testing phase. The result was a decrease in cash flow as potential customers deferred purchases of the existing products in favor of waiting for the new versions, and a corresponding loss of market share as new, competing products were introduced by other vendors.

The goal of this paper is to explore techniques with which testing resources can be most effectively allocated within a resource constrained testing environment. The approach we choose to take is allocation of resources based on quantifiable characteristics of the software. In the following sections, we further expand upon the goals and issues related to resource allocation; introduce techniques for quantifying the characteristics of computer software; describe a testing resource allocation scheme; and apply the scheme, after the fact, to a moderate sized commercial software product.

Allocation of Testing Resources

Any discussion of allocating testing resources must first begin with a definition of what a testing resource is. The testing activity requires a number of resources: people; computer time; calendar time; money for tools, etc. Further, the literature is clear that such resources are not usually interchangeable. For example, a project that would take 3 test engineers 6 months to complete may not necessarily require 3 months with 6 test engineers. Nevertheless, in this paper, we have decided to lump the resources together into generic testing resources, and discuss units of such resources as dollars for convenience. The reader should interpret our discussion as generally as possible when we mention "dollars of testing resources".

The manager of the testing activity is responsible for ensuring that testing resources are spent in the most *effective* way possible. Given the following set of assumptions (which may not always hold) we can suggest a definition for *effective allocation of testing resources*. These assumptions are:

1. All errors cost roughly the same amount to find. The simplicity of this assumption is only rivaled by the complexity of the converse. If errors do indeed cost different amounts to find, how does one assign a cost given each test is not independent? For example, does the cost to find the second error include the cost to find the first error, since the test activity to find the first error helped eliminate many of the test points before the second error was encountered?

2. In our analysis we make the assumption that the number of errors actually encountered, while not necessarily all the errors, are at least representative of the relative proportions of errors in each module. Thus for example, if 10% of the errors found were found in module A, we assume that module A will actually account for 10% of all the errors in the product. Without this assumption, any analysis would be impossible since we must then struggle with exactly the same problem as any test engineer: "when do you know all the bugs have been found?".

Given these assumptions, our definition of an *effective allocation of testing resources* is an allocation such that every module of the software has the same distribution of resources per error. For example, if we have three modules, A, B and C; and A and B have 5 bugs and C has 50 bugs,

an effective allocation of testing resources would be one in which C has 10 times the testing resources allocated to it as does A and B, and A and B have equal testing resources allocated to them. Thus, if 60 units of resources were available to be allocated, A and B should both receive 5 units of resources, and C should receive 50 units of resources.

Perfect allocation is quite easy after the fact when actual error counts are available, but it is a challenging problem to attempt before testing begins. In fact, it is probably too optimistic to expect a perfect allocation. Rather, one should instead strive for an allocation which minimizes the variability of the allocation. This can be measured after the fact by computing the standard deviation of the average amount of resources available per error for each part of the system.

A technique which minimizes this variability will obviously be quite valuable for environments with constrained testing budgets and/or schedules. However, such a technique, to be usable, must adhere to several restrictions. First, it must be easy to apply. Probably no technique in the foreseeable future will ever improve over the judgment of an experienced project manager. However, many projects are managed by new project managers who do not have access to such experience. Thus, usable techniques must not require a significant level of judgment calls on the part of the user. Second, very few organizations consistently maintain an historical record of past projects, thus a technique which must be calibrated over several years of prior projects before it reaches an acceptable level of accuracy will also be less useful in the short run. Finally, the application of the technique should require relatively little overhead. In the sort of environment we are targeting the technique, time is a precious resource, and techniques which require significant effort will simply not be used.

A solution to these restrictions is a technique which suggests resource allocation based on quantifiable characteristics of the system's source code. Techniques which purport to quantify certain characteristics of a program are referred to as *software metrics*. The allocation technique we propose uses certain software metrics to predict the relative number of errors in each part of the software system, and then allocates corresponding proportions of the testing resources.

Software Metrics

Software metrics are measures of certain software characteristics. Measurement is simply the process of associating a symbol with an object, based upon some property of the object. Some examples of measurement include:

Personality Types (Type A, B)
Tire Quality (good, better, best)
Temperature (72 degrees)
Length of a Trip (462 miles)

In each case, certain characteristics of the objects being measured are used to determine which symbol should be associated with the object. For example, in the personality type measure, the number of hours a person usually works, or their ability to relax will assign them either an 'A' or a 'B'. In the tire quality example, various characteristics of a tire, including rubber content and expected tread life will result in either a 'good', 'better' or 'best' classification. Software metrics work in a similar manner. Based upon the degree to which certain characteristics in the program exist, a number will be assigned.

Software metrics are often classified according to the type of characteristics they consider. Perhaps the two most common classifications of metrics are *Token Based Metrics* and *Control Flow Metrics*.

Token Based Metrics

Token-based metrics view certain units of text within the program as *tokens*. These metrics then

assign a number to a program based upon a summarization of the occurrence of these tokens. Lines of Code is undoubtedly the most popular token based metric. In this case, a line of text is considered a token. Often the token will be weighted in some manner to account for certain aspects of the product or environment. For example, the *E*quivalent *A*ssembler *S*ource *L*ines (EASL) which is becoming popular as an industry-wide measurement technique in Japan [Matsumoto, 1989] adjusts lines of high level code by appropriate weights so that systems written in different languages can be compared.
Another approach to token based metrics is referred to as *Software Science* [Halstead, 1977]. Software Science distinguishes two classes of tokens: *operators* and *operands*. The tokens are summarized using four parameters:

η_1 - *unique* operators
η_2 - *unique* operands
N_1 - *total* operators
N_2 - *total* operands

these parameters are then synthesized into a variety of metrics. In particular, two metrics we will be interested in are *Volume* and *Effort*.

$$\text{Volume} = (N_1 + N_2) \log_2 (\eta_1 + \eta_2)$$
$$\text{Level}^\wedge = (2/\eta_1) \times (\eta_2/N_2)$$
$$\text{Effort} = \text{Volume} / \text{Level}^\wedge$$

Control Based Metrics

A weakness with token based metrics is that they usually are not capable of assessing other aspects of a program, such as the flow of execution. To remedy the problem, a series of *control based metrics* have been suggested as well. Perhaps the best known of these metrics is the Cyclomatic Complexity [McCabe, 1976].

The cyclomatic complexity summarizes the control flow of a program and associates a number with a program's flowgraph based upon the following computation.

$$V(g) = e + n + 2$$

where e is a count of the number of edges and n is a count of the number of nodes in the program's flowgraph. A simplified calculation for programs is the number of predicates plus 1. An alternate calculation for multiple exit programs is the number of predicates minus number of exits plus 2 [Harrison 1984].

All three of these metrics, lines of code, Software Science and the Cyclomatic Complexity are popular among users of software metrics.

A Generic Allocation Technique

The discussion of software metrics have given us a foundation with which to describe our testing resource allocation technique. In short, the jth module is provided with resources tr_j:

$$tr_j = \frac{\mu(mj)}{\sum_{i=1} \mu(m_i)} \times \text{Total Resources}$$

where $\mu(m)$ is a particular software metric applied to module m. For example, if the lines of code metric were selected as μ, and we wished to allocate a portion of our five day testing window to a 500 line module out of a system of 5,000 lines, we would allocate 10% of our schedule, or half a day to its testing. A similar allocation may be made in terms of people, computer time, etc. as well. Obviously this assumes that each part of the system is equally important. If this is not the case, a subjective weighting could be used to transfer a portion of the allocation from modules viewed as least important to modules considered most important.

This is a fairly generic allocation technique since almost any metric can be used as μ. For example, Software Science Effort or Cyclomatic Complexity could both be just as easily used in place of lines of code. Additionally, a measure of some other artifact besides the program could be used, such as a count of number of words for each requirement, count of function points, etc.

Evaluating the Performance of the Technique

To assess the ability of the proposed testing resource allocation technique, we applied it (after the fact) to a moderately sized commercial development project. The project consisted of 41,234 lines of C code, divided into 20 logical modules. During testing 249 programming errors were encountered. The use of unconstrained testing (in fact, the budget and schedule were extended to allow additional testing) increases our confidence that the number of errors encountered in each module are representative.

The following table describes the characteristics of each module, and shows the distribution of errors across the system.

Module	Lines	Effort	VG	Errors
A	1,491	3,315,443	126	6
B	3,459	16,410,628	208	32
C	3,724	18,746,624	292	22
D	504	386,167	14	1
E	1,158	2,794,050	89	10
F	1,127	2,479,381	64	6
G	3,625	18,728,417	320	17
H	3,531	65,295	14	12
I	3,149	501,413	38	20
J	1,569	5,330	4	12
K	337	110,596	22	1
L	2,452	4,256,003	128	3
M	1,816	4,634,226	115	4
N	718	2,178,060	75	3
O	1,371	724,419	33	14
P	4,611	31,581,032	255	26
Q	3,146	16,737,746	356	38
R	1,112	3,475,167	82	8
S	1,253	1,838,235	70	3
T	1,081	3,684,379	84	11

We applied four variations of the testing resource allocation technique described in the previous section to this data. The variations were as follows:

Uniform Allocation ($1/k$)

This approach simply allocates the same level of resources to each module. Thus, if there are k

modules, this variation would give $^1/_k$ of the resources to each module.

LOC Based Allocation

This approach uses a count of the lines of code making up a particular module as μ. This is the closest to the way things are usually done in a resource constrained testing environment.

Metric Based Allocation Using Effort

This approach replaces μ with the Software Science Effort Measure.

Metric Based Allocation Using Cyclomatic Complexity

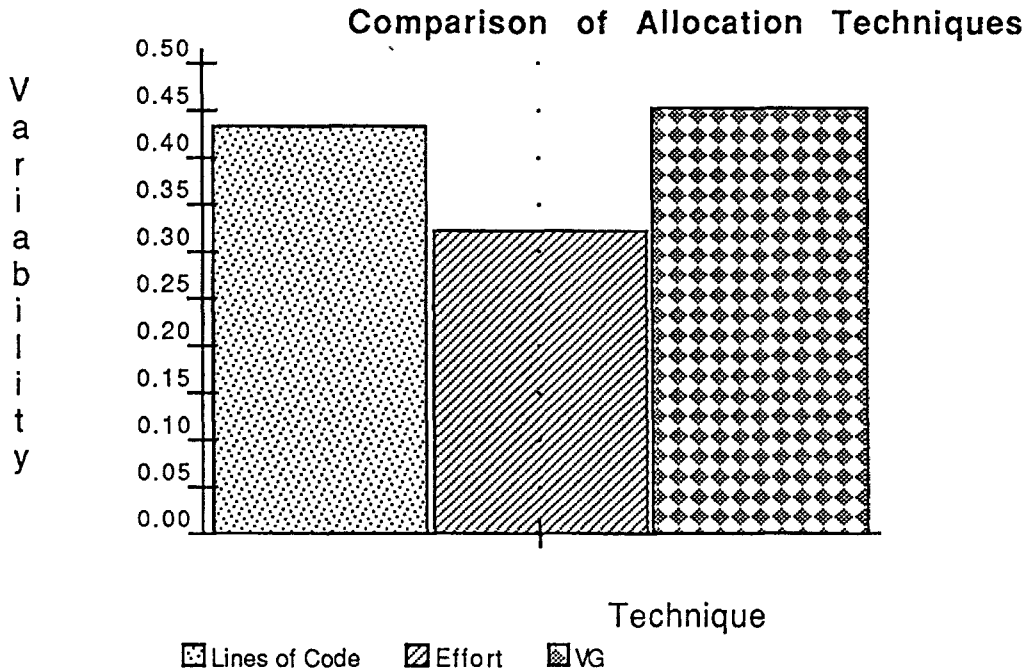
This approach replaces μ with the Software Science Effort Measure.

An Evaluation of the Performance of Each Technique

The following table shows the allocation of $100 testing budget for each module and the amount of resources this will allocate to each error:

Module	Unif.	$/Bug	LOC	$/Bug	Effort	$/Bug	VG	$/Bug
A	$5.00	$0.83	$ 3.62	$0.60	$ 2.50	$0.42	$ 5.27	$0.88
B	$5.00	$0.16	$ 8.39	$0.26	$12.37	$0.39	$ 8.71	$0.27
C	$5.00	$0.23	$ 9.03	$0.41	$14.13	$0.64	$12.22	$0.56
D	$5.00	$5.00	$ 1.22	$1.22	$ 0.29	$0.29	$ 0.59	$0.59
E	$5.00	$0.50	$ 2.81	$0.28	$ 2.11	$0.21	$ 3.73	$0.37
F	$5.00	$0.83	$ 2.73	$0.46	$ 1.87	$0.31	$ 2.68	$0.45
G	$5.00	$0.29	$ 8.79	$0.52	$14.12	$0.83	$13.39	$0.79
H	$5.00	$0.42	$ 8.56	$0.71	$ 0.05	$0.00	$ 0.59	$0.05
I	$5.00	$0.25	$ 7.64	$0.38	$ 0.38	$0.02	$ 1.59	$0.08
J	$5.00	$0.42	$ 3.81	$0.32	$ 0.00	$0.00	$ 0.17	$0.01
K	$5.00	$5.00	$ 0.82	$0.82	$ 0.08	$0.08	$ 0.92	$0.92
L	$5.00	$1.67	$ 5.95	$1.98	$ 3.21	$1.07	$ 5.36	$1.79
M	$5.00	$1.25	$ 4.40	$1.10	$ 3.49	$0.87	$ 4.81	$1.20
N	$5.00	$1.67	$ 1.74	$0.58	$ 1.64	$0.55	$ 3.14	$1.05
O	$5.00	$0.36	$ 3.32	$0.24	$ 0.55	$0.04	$ 1.38	$0.10
P	$5.00	$0.19	$11.18	$0.43	$23.81	$0.92	$10.67	$0.41
Q	$5.00	$0.13	$ 7.63	$0.20	$12.62	$0.33	$14.90	$0.39
R	$5.00	$0.63	$ 2.70	$0.34	$ 2.62	$0.33	$ 3.43	$0.43
S	$5.00	$1.67	$ 3.04	$1.01	$ 1.39	$0.46	$ 2.93	$0.98
T	$5.00	$0.45	$ 2.62	$0.24	$ 2.78	$0.25	$ 3.52	$0.32
Std Dev		*1.39*		*0.43*		*0.32*		*0.45*

As can be seen from the standard deviations, the Effort based technique appears to provide the most uniform allocations. However, Lines of Code and Cyclomatic Complexity based allocations also seem to display acceptable allocations. The relative performance of each of these three techniques are graphically illustrated in the following diagram:



Clearly this evaluation has only included a single system and required a number of simplifying assumptions. More work needs to be done on resource allocation in a resource constrained testing environment to establish which techniques are the most effective.

Tools to Facilitate the Metric-Based Resource Allocation Technique

In an earlier section, we observed that any allocation technique, to be used, must be easy to apply. The key to the acceptance of the Metric-Based Resource Allocation Technique is ready availability of tools to compute metrics and manipulate the measures for convenient resource allocation.

A software metric tool is an instrument used to facilitate the generation, collection, analysis or use of metrics. Tools to facilitate the collection and application of software metrics have progressed, and will continue to progress, through several stages:

Manual Computation. Metrics are computed by hand. This phase is usually characterized by "ad hoc" counting rules, thus measurements are often inconsistent. This approach is infeasible to apply to any but "toy" programs, and discourages experimentation with metrics.

Simple Metric Generators. The computation of the metrics are automated, but usually result in simple lists of numbers. Analysis typically requires either rekeying the measurements or writing additional, special-purpose analysis tools. Since the approach is automated, it enforces consistency, but exploration is still discouraged since analysis of the measures requires significant effort on the part of the user. Typically these sorts of tools have difficulty dealing with systems of components.

Early Metric Environments. In addition to collecting the measures, some attention is given to using them. This approach typically consists of a front end consisting of a metric analyzer and a back end which provides either batch or interactive access to some "hard-wired" analysis facility such as correlations, descriptive statistics and simple summary graphics. More useful for the user than simple metric generators, this approach still limits the analysis to things the developer anticipated. Such tools usually have some provisions for dealing with software systems as opposed to single modules.

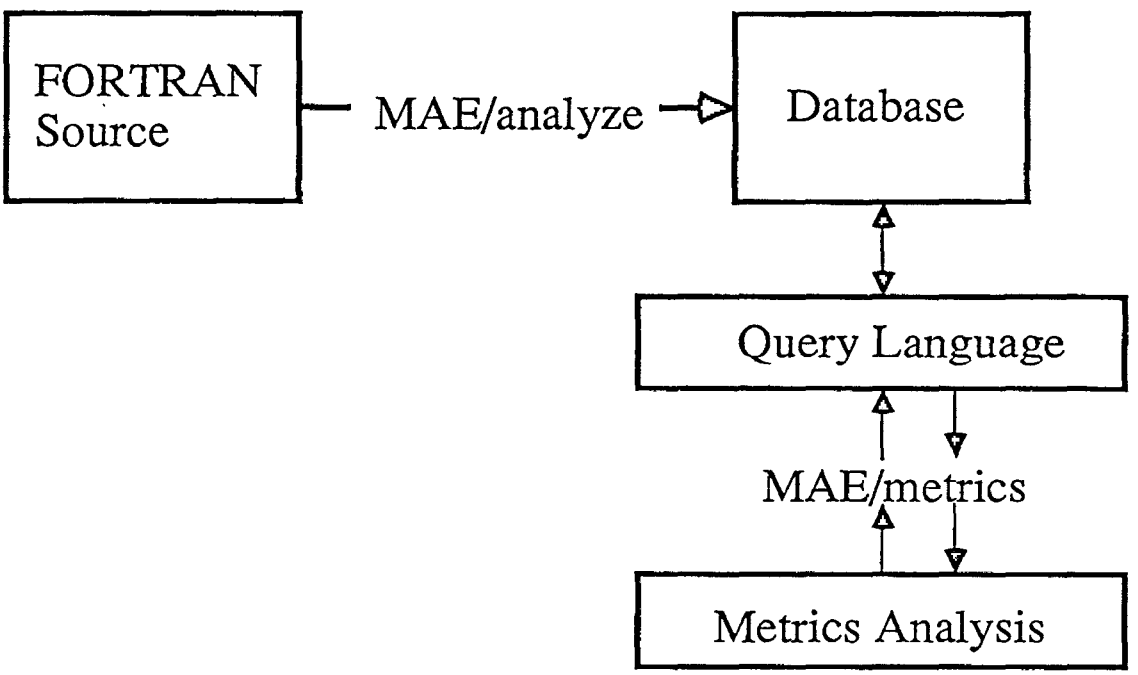
Mature Metric Environments. Integrates the tools with third-party analysis/data-manipulation tools. In particular databases and spreadsheets. Provides the maximum level of support to the interested user since it is easy to get the sort of analysis they want, when they want it, the way they want it. The third-party tools often support "what-if", statistical and selective analysis. At this point, metrics become a true management tool. Often these environments may be integrated with bug-tracking and/or configuration control systems.

The following section describes a prototype Mature Environment tool and discusses how it can be used to support Metric-Based Resource Allocation.

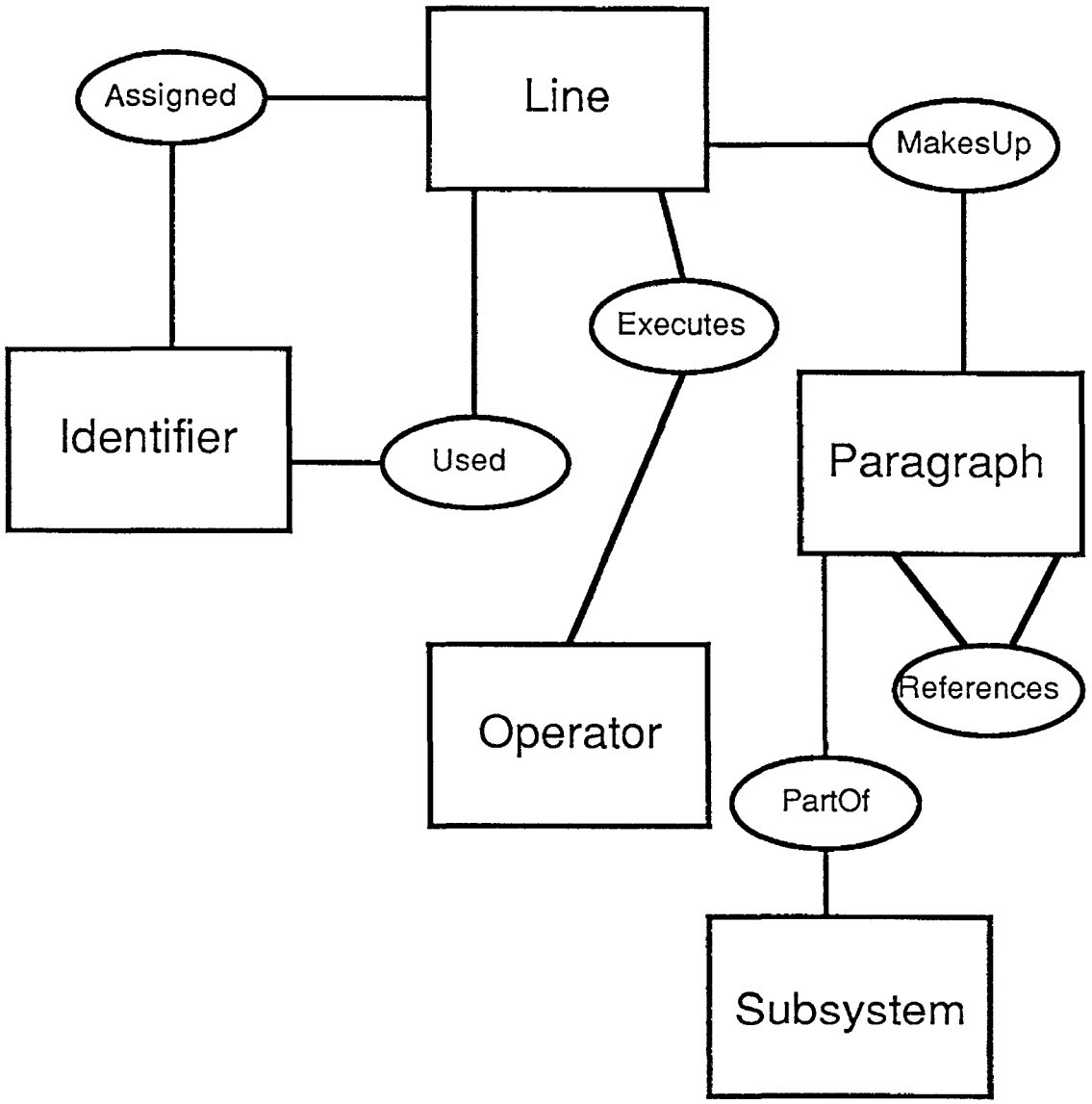
MAE: The *M*etric *A*nalysis *E*nvironment

MAE [Harrison 1987, 1988] is a prototype metrics analysis environment implemented Under VAX VMS for COBOL. MAE is best suited for the collection and use of static syntactic metrics, such as Software Science. The philosophy which drove the development of MAE is that of "late-binding". In this view, the actual computation of the metrics should be delayed as long as possible.

In fact, MAE allows the user to delay selecting an analysis method until the metrics are to be used. This is done by maintaining static characteristics of the program in a Datatrieve database, and then using the built-in query language to collect, manipulate and analyze the metrics as desired. This is accomplished by employing MAE/analyze (written in COBOL under VMS) to analyze a COBOL program, and populate a database representing the program. A set of prewritten Datatrieve query scripts called MAE/metrics (or user-written Datatrieve scripts) can then be used to extract the metrics desired. This process is shown below:



The schema used by MAE was designed and implemented using Chen's Entity-Relationship Model [Chen, 1974]. The schema essentially captures information about the essential entities in the system as well as the relationships between them. An **Identifier** is *used* and *assigned* in a particular line, an **Operator** is *executed* in a particular line, a set of **Lines** *makesup* a **Paragraph**, a **Paragraph** *references* other **Paragraph** via PERFORM and GOTO statements and a **Paragraph** is *partof* a **Subsystem**. This can be seen in the following entity-relationship diagram where the entities are represented by boxes and the relationships are represented as ovals:



The schema was then translated into Datatrieve, and a tool written to analyze a COBOL program and populate the database. Various metrics can then be derived by querying the schema to compute the desired metrics. For example, a query for Software Science Effort for an entire system would be:

```
Eta1 = COUNT OF Operator
Eta2 = COUNT OF Identifier
N1 = TOTAL Cnt OF Operator CROSS Executed OVER Oid
N2 = TOTAL Cnt OF Identifier CROSS Used OVER Iid +
     TOTAL Cnt OF Identifier CROSS Assigned OVER Iid
Eta = Eta1 + Eta2
N = N1 + N2
V = N * (FN$LOG10(Eta)/FN$LOG10(2))
Lh=(2/eta1)*(eta2/N2)
Effort=V/Lh
PRINT "Software Science Effort is: ", Effort
```

A query for Noncommentary Lines of Code would be written as:

```
PRINT "NCSL is: ", COUNT OF Line
```

The benefit of this approach is that changes to the metrics and/or counting rules can be tried without reanalyzing the entire set of source code. This obviously requires less machine time, but it is of particular importance to users who may not have convenient, direct access to the source. For example, academic researchers and consultants may only be allowed infrequent access to a company's proprietary code, and so will usually encounter significant difficulties attempting to duplicate a study using new metrics. On the other hand, by "aliasing" operators and identifier/paragraph names (this can be done by the analysis tool), the actual content of the software can be protected, allowing wide distribution of the database for other researchers.

Additionally, the MAE system also allows the user to customize the delivery of the metric report by using Datatrieve's Report Writer and/or graphical presentation facilities. Likewise, the reports may be written to a file in a format which can be analyzed using a statistical package such as SPSS or SAS.

Due to performance limitations, MAE is currently being revamped to utilize PROLOG in place of a third-party Database Management System. A FORTRAN version of the PROLOG-based tool is described in [Harrison, 1989].

Summary

In this paper we have described and evaluated a metric-based technique for allocating constrained testing resources among the components of a software system. Based on our evaluation of a single software system, we conclude that Software Science Effort is perhaps the best basis upon which to allocate testing resources, though the LOC- and Cyclomatic- based techniques also exhibited comparable performance. To help make the technique more accessible to users, we also described a novel technique for separating the code analysis and metric generation activities.

Even though our results are encouraging, we must to stress the need for larger, more comprehensive studies to establish more conclusive evidence regarding the viability of any testing strategy.

References

[1] Chen, P., "The Entity-Relationship Model - Towards a Unified View of Data", *ACM Transactions on Database Systems,* January 1976, pp 9-36.

[2] Halstead, M., *Elements of Software Science*, Elsevier North Holland, New York 1977.

[3] Harrison, W., "Applying McCabe's Complexity Measure to Multiple-Exit Programs", *Software-Practice & Experience*, October 1984, pp 1004-1007.

[4] Harrison, W., "An Extensible Static Analysis Tool for COBOL Programs", *Proceedings of the 1987 ACM Computer Science Conference,* February 1987, pp 285-291.

[5] Harrison, W., "MAE: A Syntactic Metric Analysis Environment", *The Journal of Systems and Software,* January 1988, pp 57-62.

[6] Harrison, W., "PDSS: A Programmer's Decision Support System", *Data & Knowledge Engineering*, December 1989, pp 115-123.

[7] Matsumoto, Y., "An Overview of Japanese Software Factories", in *Japanese Perspectives in Software Engineering,* (Matsumoto and Ohno, editors), Addison-Wesley Publishing Co., 1989.

[8] McCabe, T., "A Complexity Measure", *IEEE Transactions on Software Engineering,* December 1976, pp 308-320.

Paper 3-A-3

A BENCHMARKING PROCESS FOR SOFTWARE ENGINEERING AND QUALITY

Mr. Rodger Drabick
Senior Software Quality Assurance Staff
Eastman Kodak Corporation

Mr. Rodger Drabick is Senior Software Quality Assurance Staff in the Kodak Federal Systems Division. He has worked for Kodak for over 25 years, and served as Manager Quality Management Services in the Kodak Software Systems Division, and as Quality Assurance Manager for Special Programs in the Kodak Federal Systems Division. Mr. Drabick organized and supervised the first formal software quality assurance and testing group in Federal Systems Division in 1981; prior to that, he was supervisor of the Software and Operations Group, responsible for development of the Factory-FAN Software Suite. He is chairman of the Kodak Software Quality Improvement Group, is Vice-president of the Rochester Software Quality Association, and is a member of the Data Processing Management Association.

A BENCHMARKING PROCESS FOR SOFTWARE ENGINEERING AND QUALITY

A PRESENTATION FOR "QUALITY WEEK"

RODGER DRABICK, CQA
MAY 1990

FEDERAL SYSTEMS DIVISION

SOFTWARE BENCHMARK

FEDERAL SYSTEMS DIVISION

INTRODUCTION

SOFTWARE BENCHMARK

INTRODUCTION:

MANY COMPANIES ARE BEGINNING TO REALIZE THAT SOFTWARE IS BECOMING A MORE SIGNIFICANT CONTRIBUTOR TO THEIR PERFORMANCE. IT IS THEREFORE NECESSARY TO SIGNIFICANTLY IMPROVE SOFTWARE ENGINEERING AND QUALITY, HENCE A "CHANGE IN CULTURE" RELATIVE TO SOFTWARE IS MANDATED.

IT IS NECESSARY TO MOVE FROM THE CONCEPT OF SOFTWARE DEVELOPMENT AS A "BLACK ART" WITH PROGRAMMERS AS "HIGH PRIESTS AND PRIESTESSES" INTO A CULTURE WHERE SOFTWARE IS APPROACHED AS AN ENGINEERING DISCIPLINE WITH DOCUMENTED POLICIES, STANDARDS, AND PROCEDURES.

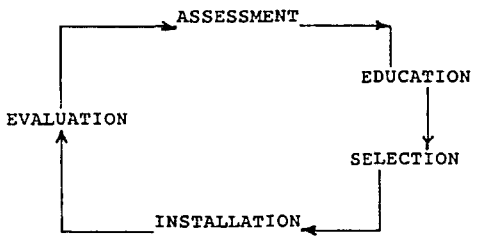
FEDERAL SYSTEMS DIVISION

SOFTWARE BENCHMARK

INTRODUCTION (CONT):

IN SHORT, WE MUST FOLLOW THE LEAD OF DR. ROGER PRESSMAN AS EXPRESSED IN HIS BOOK "MAKING SOFTWARE ENGINEERING HAPPEN."

IN HIS BOOK, DR. PRESSMAN NOTES THAT THERE IS A DEFINED PROCESS TO MAKE SOFTWARE ENGINEERING HAPPEN. IT INCLUDES:



SOFTWARE BENCHMARK

INTRODUCTION (CONT):

THUS, WHEN WE IN KODAK'S FEDERAL SYSTEMS DIVISION DECIDED TO IMPLEMENT IMPROVEMENTS TO YIELD BETTER SOFTWARE, WE BEGAN WITH AN ASSESSMENT, OR "BENCHMARK."

SOFTWARE BENCHMARK

INTRODUCTION (CONT.):

SIMPLY STATED, THE OBJECTIVES OF BENCHMARKING ARE TO DETERMINE WHERE YOU RANK RELATIVE TO AN ACCEPTED SET OF CRITERIA.

AFTER ALL, IF YOU DON'T KNOW WHERE YOU ARE, A MAP IS OF NO USE TO YOU!

AND

IF YOU DON'T KNOW WHERE YOU'RE GOING, ANY ROAD WILL TAKE YOU THERE.

SOFTWARE BENCHMARK

INTRODUCTION (CONT.):

THERE ARE TWO RELATIVELY WELL-KNOWN METHODS FOR PERFORMING A SOFTW ENGINEERING ASSESSMENT:

1) THE METHOD DESCRIBED IN APPENDIX B.1 OF DR. PRESSMAN'S BOOK
2) THE METHOD DEVELOPED BY THE SOFTWARE ENGINEERING INSTITUTE AT CARNEGIE MELLON UNIVERSITY FOR THE DEPARTMENT OF DEFENSE.

BENCHMARKING PROCESS

FEDERAL SYSTEMS DIVISION

SOFTWARE BENCHMARK

BENCHMARKING PROCESS:

THE PURPOSE OF A SOFTWARE ENGINEERING AUDIT/ASSESSMENT/BENCHMARK IS TO PROVIDE A RIGOROUS DEFINITION OF AN ORGANIZATION'S SOFTWARE ENGINEERING CAPABILITIES. IT SHOULD PROVIDE AN UNBIASED VIEW OF SOFTWARE TECHNOLOGY WITHIN THE ORGANIZATION.

IN SHORT, A BENCHMARK SHOULD PROVIDE YOUR ORGANIZATION AN OPPORTUNITY TO "LOOK IN THE MIRROR" RELATIVE TO YOUR SOFTWARE ENGINEERING CAPABILITIES, AS SHOWN BY HISTORICAL DATA.

FEDERAL SYSTEMS DIVISION

SOFTWARE BENCHMARK

BENCHMARKING PROCESS (CONT.):

THE TWO MOST WIDELY RECOGNIZED BENCHMARKING METHODS ARE THOSE DEVELOPED BY DR. ROGER PRESSMAN (R. S. PRESSMAN AND ASSOCIATES, INC.) AND WATTS HUMPHREY'S TEAM AT THE SOFTWARE ENGINEERING INSTITUTE AT CARNEGIE MELLON UNIVERSITY.

DR. PRESSMAN'S APPROACH IS DESCRIBED IN CHAPTER 4 OF HIS BOOK, "MAKING SOFTWARE ENGINEERING HAPPEN."

THE SOFTWARE ENGINEERING INSTITUTE PROCESS IS DESCRIBED IN "A METHOD FOR ASSESSING THE SOFTWARE ENGINEERING CAPABILITY OF CONTRACTORS."

SOFTWARE BENCHMARK

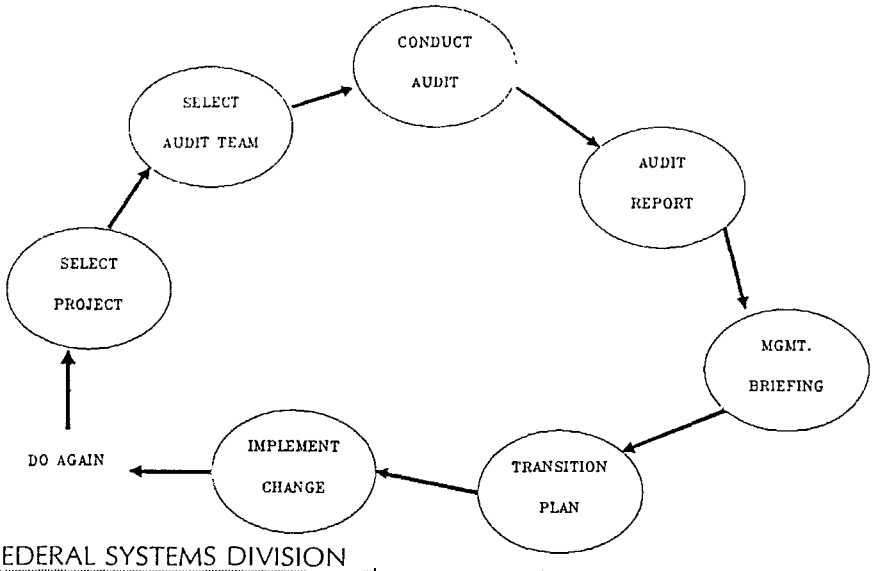
BENCHMARKING PROCESS (CONT.):

DR. PRESSMAN'S PROCESS INVOLVES:

1) SELECTING SUBJECT AREAS FOR THE AUDIT
2) SELECTING AN AUDIT TEAM (USE AN OUTSIDE CONSULTANT)
3) QUALITATIVE AUDIT
4) QUANTITATIVE AUDIT
5) AUDIT REPORT
6) MANAGEMENT PRESENTATION
7) TRANSITION PLAN
8) IMPLEMENT THE PLAN
9) REPEAT THE AUDIT.

SOFTWARE BENCHMARK

BENCHMARKING PROCESS (CONT.):



SOFTWARE BENCHMARK

BENCHMARKING PROCESS (CONT.):

THE SOFTWARE ENGINEERING INSTITUTE'S APPROACH CONCENTRATES ON THE SOFTWARE ENGINEERING PROCESS, AND INVOLVES:

1) ASSESSMENT TRAINING

2) ASSESSMENT PLANNING

3) ASSESSMENT

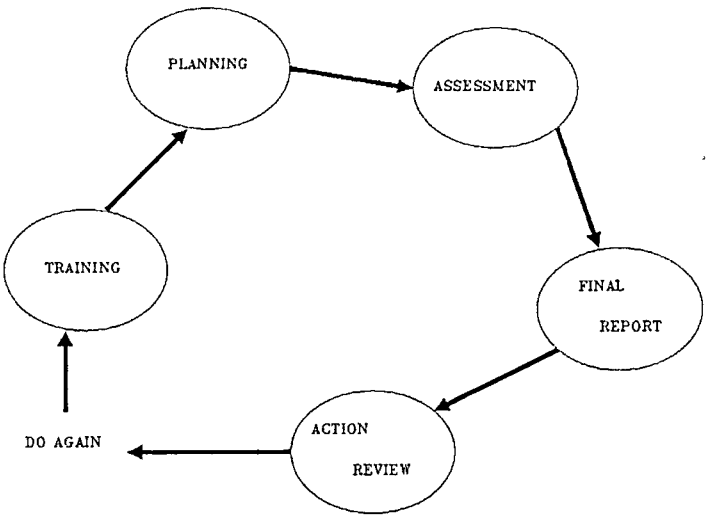
   A) IN-BRIEFING
   B) PROJECT LEADER DISCUSSIONS
   C) FUNCTIONAL AREA DISCUSSIONS
   D) PROJECT FEEDBACK
   E) OUT-BRIEFING

4) FINAL REPORT DELIVERY AND RECOMMENDATIONS BRIEFING

5) ACTION PLAN REVIEW

6) FOLLOW-UP ASSESSMENT

BENCHMARKING PROCESS (CONT.):



SOFTWARE BENCHMARK

BENCHMARKING PROCESS (CONT.):

THE TWO APPROACHES ARE QUITE SIMILAR, AT THE HIGH LEVEL.

BOTH CONSIDER ASSESSMENT/BENCHMARKING AS THE STARTING POINT FOR A CONTINUOUS IMPROVEMENT PROGRAM.

BOTH APPROACHES REQUIRE A STRUCTURED IMPLEMENTATION WITH A SIGNIFICANT EFFORT DEVOTED TO PLANNING AND DEVELOPING RECOMMENDATIONS TO IMPROVE YOUR ORGANIZATION'S STATE OF THE PRACTICE.

WE CHOSE THE SEI APPROACH FOR TWO REASONS:

1) THIS APPROACH PROVIDES A QUANTITATIVE MEASURE
2) FSD WORKS FOR DOD CUSTOMERS, IN GENERAL.

SOFTWARE BENCHMARK

BENCHMARKING PROCESS; THE SEI METHOD:

THE SEI ASSESSMENT IS AN APPRAISAL BY A TRAINED TEAM OF EXPERIENCED SOFTWARE PROFESSIONALS OF AN ORGANIZATION'S CURRENT SOFTWARE ENGINEERING PROCESS.

THE ASSESSMENT IS BASED ON:

1) REVIEW OF 5 OR 6 KEY PROJECTS
2) RESPONSES TO AN ASSESSMENT QUESTIONNAIRE
3) DISCUSSIONS WITH PROJECT MANAGERS AND PRACTITIONERS
4) KNOWLEDGE AND EXPERIENCE OF THE ASSESSMENT TEAM.

SOFTWARE BENCHMARK

BENCHMARKING PROCESS; THE SEI METHOD (CONT.):

THE SEI ASSESSMENT IS BASED ON THE FOLLOWING PRINCIPLES:

1) SENIOR MANAGEMENT INVOLVEMENT
2) PROCESS FRAMEWORK BASIS
3) CONFIDENTIALITY
4) COLLABORATION
5) ACTION ORIENTATION.

SOFTWARE BENCHMARK

BENCHMARKING PROCESS; THE SEI METHOD (CONT.):

SEI PROVIDES THE FOLLOWING DEFINITIONS OF PROCESS MATURITY:

PROCESS MATURITY LEVEL 1 - INITIAL; SOFTWARE ENVIRONMENT HAS ILL DEFINED PROCEDURES AND CONTROLS.

PROCESS MATURITY LEVEL 2 - REPEATABLE; ORGANIZATION CAN MANAGE COST AND SCHEDULES, PROCESS IS REPEATABLE.

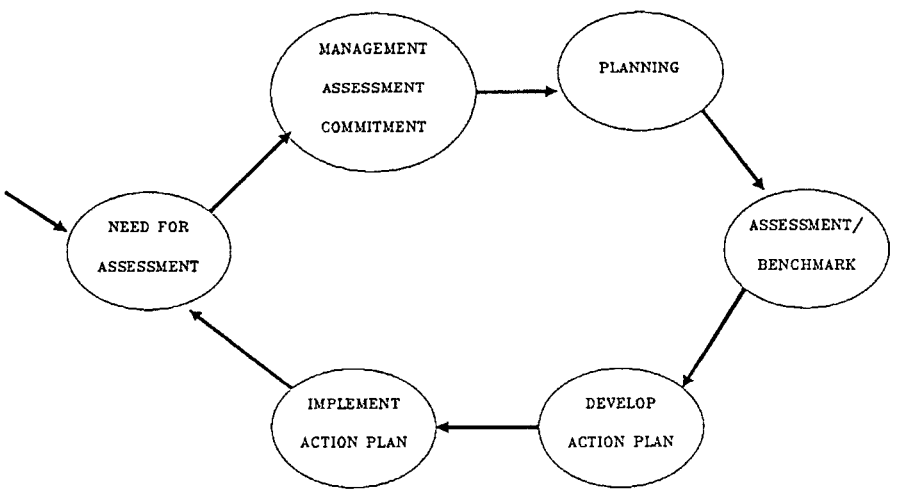
PROCESS MATURITY LEVEL 3 - DEFINED; PROCESS DEFINED IN TERMS OF SOFTWARE ENGINEERING STANDARDS AND METHODS.

PROCESS MATURITY LEVEL 4 - MANAGED; PROCESS IS UNDERSTOOD, QUANTIFIED, MEASURED, AND CONTROLLED.

PROCESS MATURITY LEVEL 5 - OPTIMIZING; ORGANIZATION IS WORKING TO IMPROVE THE PROCESS.

SOFTWARE BENCHMARK

BENCHMARKING PROCESS (CONT.):



SOFTWARE BENCHMARK

BENCHMARKING PROCESS; THE SEI METHOD (CONT.):

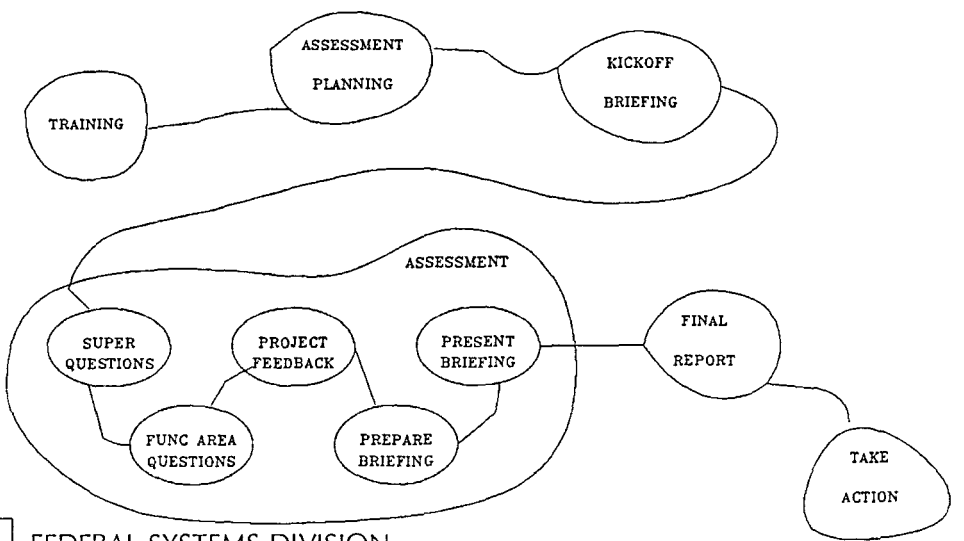
THE SEI PROCESS INVOLVES A SIGNIFICANT PLANNING FUNCTION, AND ALSO INVOLVES ACTIVE FEEDBACK OF DATA TO THE PARTICIPANTS IN THE ASSESSMENT PROCESS.

THE PROCESS INCLUDES THE FOLLOWING STEPS:

1) TRAINING
2) ASSESSMENT PLANNING (1 MONTH ELAPSED TIME)
3) PRE-ASSESSMENT BRIEFING
4) ASSESSMENT
5) FINAL REPORT (WITH IMPROVEMENT RECOMMENDATIONS)
6) TAKE ACTION.

SOFTWARE BENCHMARK

BENCHMARKING PROCESS; THE SEI METHOD (CONT.):



SOFTWARE BENCHMARK

BENCHMARKING PROCESS; THE SEI METHOD (CONT.):

A SIGNIFICANT POINT TO REMEMBER IS,

"BE PREPARED TO TAKE ACTION,
OR, DON'T ASSESS."

SOFTWARE BENCHMARK

EXAMPLE PROCESS

SOFTWARE BENCHMARK

EXAMPLE PROCESS, FSD:

AT FSD, WE BEGAN BY LAYING OUT A PLAN TO CONDUCT THE ASSESSMENT IN GENERAL COMPLIANCE WITH THE SEI PROCESS. AFTER DEVELOPING AND REVIEWING THE PLAN, WE CUSTOMIZED THE PLAN IN ORDER TO ACHIEVE OUR GOALS IN OUR DESIRED TIMEFRAME.

WE CONVENED A GROUP OF MANAGERS TO SELECT OUR INTERVIEWING TEAMS, AND SELECT THE SET OF PROJECTS WE WOULD SUBJECT TO INTERVIEWS. WE DECIDED EARLY ON THAT OUR INTERVIEWS WOULD CONSIST OF ASKING THE PROJECT REPRESENTATIVES THE SET OF QUESTIONS FROM THE SEI DOCUMENT.

SOFTWARE BENCHMARK

EXAMPLE PROCESS (CONT.):

AT THE TOP LEVEL, OUR APPROACH INCLUDED:

1) DETAILED PLANNING
2) ASSESSMENT TRAINING
3) PRE-BENCHMARK ACTIVITIES
   A) PERFORM READINESS SURVEY
   B) SUBMIT BENCHMARK QUESTIONS TO PROJECT SUPERVISORS
4) PERFORM BENCHMARK
   A) CONDUCT INTERVIEWS WITH PROJECT PERSONNEL
   B) ANALYZE INTERVIEW DATA
   C) BRIEF PROJECT PERSONNEL AND MANAGERS
5) PREPARE RECOMMENDATIONS TO MANAGEMENT
   A) PRESENT RECOMMENDATIONS
   B) SUGGEST FOLLOW-UP ASSESSMENT.

EXAMPLE PROCESS (CONT.):

FSD PROCESS	SEI PROCESS
DETAILED PLANNING	TRAINING
↓	↓
ASSESSMENT TRAINING	ASSESSMENT PLANNING
↓	↓
PRE-BENCHMARK ACTIVITIES	PRE-ASSESSMENT BRIEFING
↓	↓
BENCHMARK	ASSESSMENT
↓	↓
RECOMMENDATIONS	FINAL REPORT
↓	↓
TAKE ACTION	TAKE ACTION

SOFTWARE BENCHMARK

EXAMPLE PROCESS, PRE-BENCHMARK ACTIVITIES (CONT.):

SEI PROVIDES A SET OF QUESTIONS CALLED A "SELF-ASSESSMENT READINESS SURVEY" THEY SUGGEST YOU ADMINISTER BEFORE YOU BEGIN A SELF-ASSESSMENT PROJECT.

THE SET OF THIRTY QUESTIONS IS DIVIDED INTO 5 SECTIONS:

SPONSORSHIP
CULTURE
RESISTANCE
SYNERGY
ORGANIZATIONAL ISSUES.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, PRE-BENCHMARK ACTIVITIES (CONT.):

SAMPLE QUESTIONS INCLUDE:

SPONSORSHIP - "THE SPONSOR IS WILLING TO COMMIT RESOURCES TO DO THE ASSESSMENT AND THE FOLLOW-UP IMPROVEMENT ACTIVITIES. "

CULTURE - "THERE IS CONSISTENCY BETWEEN THE WAY GOALS, TASKS, AND ROLE ASSIGNMENTS ARE CURRENTLY DEFINED AND THE WAY THEY'RE EXPECTED TO BE DEFINED WHEN BEGINNING A PROCESS IMPROVEMENT EFFORT. "

RESISTANCE - "THE ANTICIPATED IMPACT ON BUDGETS AND SCHEDULES IS SEEN BY MANAGEMENT AS A REASONABLE COST OF PROCESS IMPROVEMENT. "

SOFTWARE BENCHMARK

EXAMPLE PROCESS, PRE-BENCHMARK ACTIVITIES (CONT.):

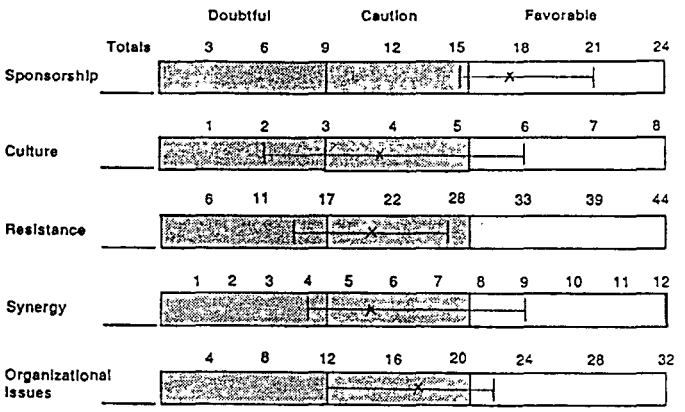
TO EVALUATE OUR READINESS FOR BENCHMARKING, WE ROUTED THE READINESS SURVEY QUESTIONS TO SUPERVISORS AND MIDDLE MANAGERS KNOWLEDGEAI IN OUR SOFTWARE BUSINESS AND ORGANIZATIONAL STRUCTURE.

WE THEN COMPILED THE DATA, AND PLOTTED THE AVERAGES ON THE FORM SUPPLIED BY SEI. THE AVERAGE AND "SPREAD" DATA IS SHOWN IN THE FOLLOWING CHART.

WE CONCLUDED FROM THE DATA THAT WE COULD SUCCESSFULLY CONDUCT A BENCHMARK.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, SURVEY RESULTS (CONT.):



SOFTWARE BENCHMARK

EXAMPLE PROCESS, PRE-BENCHMARK ACTIVITIES (CONT.):

AFTER CONCLUDING THE READINESS SURVEY, WE ROUTED THE ASSESSMENT QUESTIONNAIRE (101 QUESTIONS) TO SUPERVISORS OF THE SIX PROJECTS WE HAD SELECTED TO BENCHMARK.

THE BENCHMARK TEAM MET AND REVIEWED THE ANSWERS TO DETERMINE IF THE RESPONSES COULD SUGGEST AREAS FOR DEEPER PROBING DURING THE INTERVIEWS OF THE DEVELOPMENT TEAMS.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, PRE-BENCHMARK ACTIVITIES (CONT.):

AS A FINAL NOTE, ALL LOGISTICAL PREPARATIONS WERE COMPLETED TO SUPPORT THE PROJECT INTERVIEWS. THIS INCLUDED:

1) CONFERENCE ROOM SCHEDULING
2) PROJECT INTERVIEW TIMING
3) SCHEDULING OF DATA ANALYSIS ACTIVITIES.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, INTERVIEWS:

INTERVIEWS WERE CONDUCTED IN AUGUST, 1989.

EACH INTERVIEWER TEAM HAD AT LEAST ONE MEMBER FROM THE DEVELOPMENT ORGANIZATION WHO WAS KNOWLEDGEABLE ON THE SPECIFIC PROJECT BEING INTERVIEWED.

WE CONDUCTED TWO INTERVIEWS PER DAY.

AFTER COMPLETING THE INTERVIEWS, THE TEAM MET TO ANALYZE THE DATA FROM THE INTERVIEWS.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, INTERVIEWS (CONT.):

INTERVIEWER TEAMS OF THREE TO FOUR SENIOR TECHNICAL PERSONNEL ASKED A SERIES OF 101 QUESTIONS (FROM THE SEI "A METHOD FOR ASSESSING THE SOFTWARE ENGINEERING CAPABILITY OF CONTRACTORS") OF TEAMS OF TWO TO FOUR MEMBERS OF THE SOFTWARE DEVELOPMENT STAFF.

BOTH INTERVIEWER AND INTERVIEWED TEAMS WERE COMPOSED OF A MIX OF DEVELOPERS AND SOFTWARE QA PERSONNEL.

ANSWERS TO THE SURVEY QUESTIONS WERE ARRIVED AT USING A CONSENSUS PROCESS.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, RESULTS:

USING THE SEI APPROACH TO BENCHMARK AN ORGANIZATION PROVIDES A QUANTITATIVE MEASURE OF THAT ORGANIZATION'S PROCESS CAPABILITY.

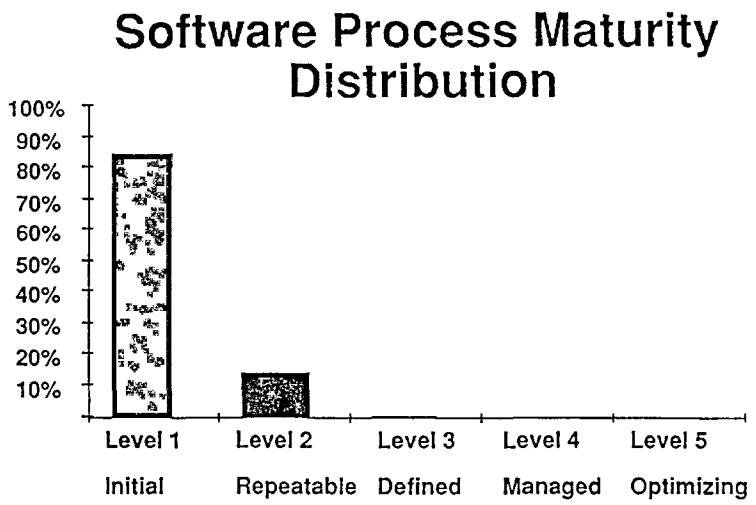
TO PUT OUR DATA IN PERSPECTIVE, HERE IS SOME DATA COMPILED BY SEI, AND NOLAN AND NORTON BASED ON SEI'S METHOD.

SEI COMPILED INFORMATION ON ASSESSING APPROXIMATELY 100 ORGANIZATIONS. THIS DATA WAS COLLECTED THROUGH NOVEMBER, 1988.

NOLAN AND NORTON SURVEYED 11 ORGANIZATIONS IN 1989, USING THE SEI QUESTIONNAIRE.

SOFTWARE BENCHMARK

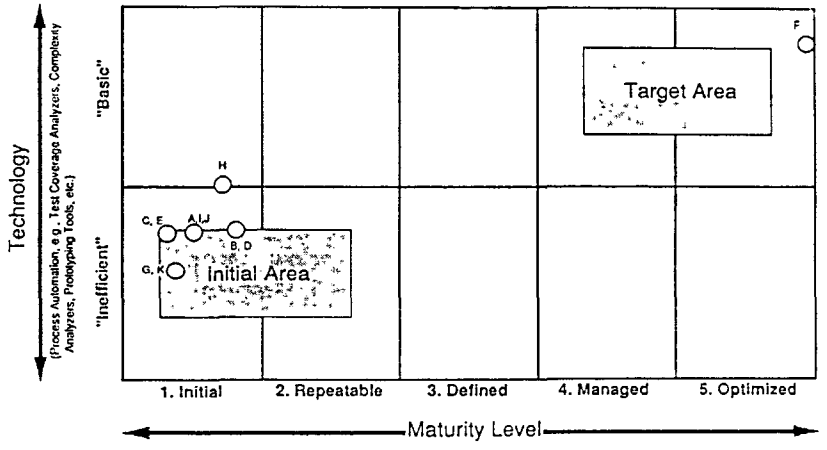
SEI DATA:



FEDERAL SYSTEMS DIVISION

SOFTWARE BENCHMARK

NOLAN AND NORTON DATA:



FEDERAL SYSTEMS DIVISION

SOFTWARE BENCHMARK

EXAMPLE PROCESS, RESULTS (CONT.):

AS PREVIOUSLY NOTED, THE SEI QUESTIONNAIRE CONTAINS 101 QUESTIONS. A SUBSET OF THE QUESTIONS IS ASSIGNED TO EACH OF THE FIVE "PROCESS MATURITY LEVELS."

THE QUESTIONNAIRE CONTAINS CERTAIN QUESTIONS IN EACH "PROCESS MATURITY LEVEL" THAT ARE DESIGNATED AS "CRITICAL."

THE SEI METHOD REQUIRES THAT AN ORGANIZATION ANSWER "YES" TO 80% OF THE QUESTIONS ASSIGNED TO EACH LEVEL, AND ANSWER "YES" TO 90% OF THE "CRITICAL" QUESTIONS TO QUALIFY AT A PARTICULAR MATURITY LEVEL.

THE FOLLOWING CHART DISPLAYS THE VALUES OBTAINED DURING OUR BENCHMAR AND COMPARES THEM WITH THE SEI "SPECS."

SOFTWARE BENCHMARK

EXAMPLE PROCESS, RESULTS (CONT.):

SUMMARY OF INTERVIEW DATA

	SEI PROCESS MATURITY LEVELS							TECHNOLOGY LE	
	1	2		3		4			
		ALL	CRIT	ALL	CRIT	ALL	CRIT	ALL	CRIT
FSD OVERALL	OK	28%	29%	24%	26%	21%	21%	36%	51
SEI SPECIFICATION		*80*	*90*	*80*	*90*	*80*	*90*	*80*	*90*

AS WITH THE MAJORITY OF ORGANIZATIONS MEASURED, OUR ORGANIZATION CURRENTLY SHOWS A PROCESS MATURITY LEVEL OF 1.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, RESULTS (CONT.):

THE SEI SURVEY QUESTIONS HAVE A STRICT INTERPRETATION OF THE TERM "MECHANISM." THE BENCHMARK SHOWED THAT IN MANY CASES, FSD HAS A "SYSTEM" FOR DOING BUSINESS, BUT THE LACK OF DOCUMENTATION OR AN ASSURANCE METHOD FORCES A "NO" ANSWER TO THE ASSESSMENT QUESTION.

SOFTWARE DEVELOPERS PERCEIVE THAT SENIOR MANAGEMENT DOES NOT HAVE THE SAME INVOLVEMENT IN SOFTWARE DEVELOPMENT AS IN HARDWARE DEVELOPMENT. (THIS PROBABLY RESULTS FROM THE FACT THAT OUR MANAGERS HAVE MORE EXPERIENCE ON HARDWARE PROGRAMS.)

SOFTWARE DEVELOPERS PERCEIVE THEY ARE NOT RESPECTED BY MANAGERS OR HARDWARE ENGINEERS.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, ANALYSIS:

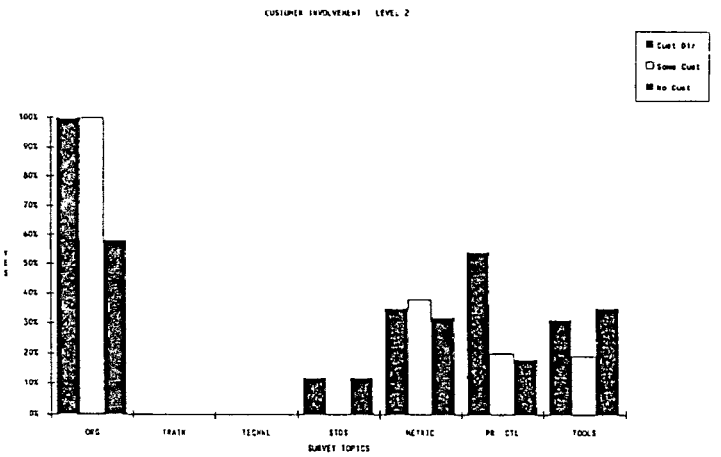
TO HELP US UNDERSTAND THE BENCHMARK DATA, WE DISPLAYED THE RESULTS FROM THE "PROCESS MATURITY LEVEL 2" QUESTIONS GRAPHICALLY.

INITIALLY, WE INVESTIGATED TO SEE IF A HIGH DEGREE OF CUSTOMER CONTROL ON PROJECTS LED TO A PROCESS MORE IN TUNE WITH SEI'S PRESCRIPTION.

WE THEORIZED THAT A CUSTOMER MIGHT IMPOSE LIFECYCLE AND PROCESS REQUIREMENTS ON PROJECTS, IN AN ATTEMPT TO MAINTAIN COST AND SCHEDULE, AND MINIMIZE RISK. THOSE PROJECTS MIGHT DEMONSTRATE A HIGHER "PROCESS MATURITY LEVEL."

DATA INDICATES PROJECTS WITH STRONG CUSTOMER DIRECTION MORE CLOSELY RESEMBLE SEI'S MODEL FOR ORGANIZATION AND PROCESS CONTROL.

SOFTWARE BENCHMARK



FEDERAL SYSTEMS DIVISION

SOFTWARE BENCHMARK

EXAMPLE PROCESS, ANALYSIS:

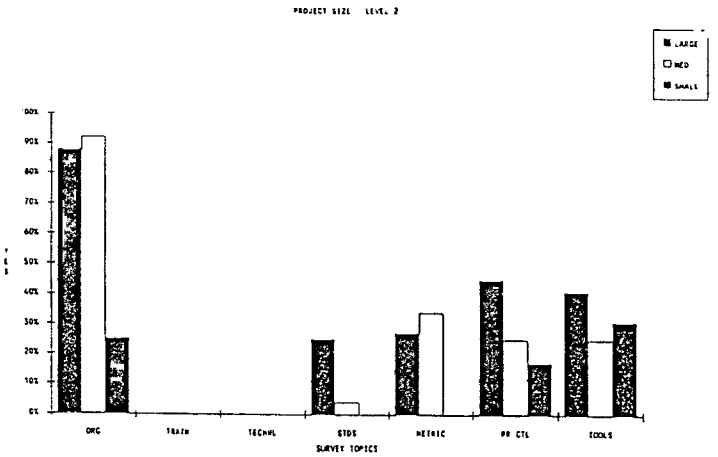
WE THEN INVESTIGATED TO SEE IF THE "SIZE" OF A SOFTWARE PROJECT LED TO A PROCESS MORE IN TUNE WITH THE SEI MODEL.

WE THEORIZED THAT A MANAGER MIGHT IMPOSE A MORE RIGOROUS PROCESS ON LARGE PROJECTS, IN AN ATTEMPT TO BETTER CONTROL COST AND SCHEDULE, AND MINIMIZE RISK. THOSE PROJECTS MIGHT THEN DEMONSTRATE A HIGHER "PROCESS MATURITY LEVEL."

THE DATA INDICATES OUR LARGER PROJECTS ARE MORE CLOSELY ATTUNED TO THE MODEL, IN AREAS OF "ORGANIZATION", "STANDARDS", AND "PROCESS CONTROL."

FEDERAL SYSTEMS DIVISION

SOFTWARE BENCHMARK



FEDERAL SYSTEMS DIVISION

SOFTWARE BENCHMARK

EXAMPLE PROCESS, ANALYSIS:

TO HELP US DETERMINE WHAT AREAS WOULD PROVIDE THE BEST IMPROVEMENT IN OUR PROCESS, WE DISPLAYED THE BENCHMARK QUESTIONS (TO QUALIFY FOR LEVEL 2) AGAINST WHICH WE DID NOT SCORE WELL.

WE ASSOCIATED A TOP LEVEL "SHORT PHRASE" THAT CAPTURED THE ESSENCE OF THE QUESTION.

EXAMINATION OF THIS DATA INDICATED THAT WE COULD SIGNIFICANTLY IMPROVE OUR PROCESS MATURITY LEVEL BY DEVELOPING APPROPRIATE STANDARDS AND PROCEDURES.

FEDERAL SYSTEMS DIVISION

SOFTWARE BENCHMARK

LEVEL 2 QUESTION NUMBER	TOP LEVEL CATEGORY	PERCENT "YES" ANSWERS
1.2.2	TRAINING	0
1.3.1	TECHNOLOGY	0
* 2.1.3	PROCEDURE	0
2.1.4	PROCEDURE	0
2.1.5	SUBCONTRACTOR CONTROL	0
* 2.1.14	PROCEDURE	0
* 2.1.15	PROCEDURE	0
* 2.1.16	PROCEDURE	0
2.1.17	MECHANISM -REQUIREMENTS	0
2.2.1	STAFFING PROFILES	33
* 2.2.2	METRICS	17
2.2.7	METRICS	17
2.2.8	METRICS	0
2.2.9	METRICS	17
2.2.10	METRICS	17
2.2.11	METRICS	33
2.2.12	METRICS	17
2.2.16	SPR TRACKING	33
2.2.18	TEST TRACKING	0
2.2.19	METRICS	50
* 2.4.1	MGMT STATUS REVIEW	0
2.4.5	MECHANISM - CUSTOMER INTERCHANGE	17
2.4.20	PROCEDURE	17

SOFTWARE BENCHMARK

EXAMPLE PROCESS, ANALYSIS CONCLUSIONS:

DOD PERFORMANCE OF AN SEI ASSESSMENT FSD WOULD SHOW WE HAVE A "PROCES: MATURITY LEVEL" OF "ONE."

TWO SIGNIFICANT FACTORS WERE IDENTIFIED THAT WILL LEAD TO MAJOR IMPROVEMENTS IN OUR PROCESS MATURITY LEVEL:

1) ESTABLISHING A POLICY, AND A SET OF STANDARDS AND PROCEDURES TO SPECIFY OUR DEVELOPMENT PROCESS
2) DEVELOPING A PROGRAM TO ACQUIRE AND ANALYZE PRODUCT AND PROCESS METRICS DATA.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, ACTIONS TAKEN:

A TEAM OF SENIOR SOFTWARE PROFESSIONALS PARTICIPATED IN TWO JOINT APPLICATION DEVELOPMENT (JAD) SESSIONS TO DEVELOP A STRATEGY FOR IMPROVING OUR PROCESS.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, ACTIONS TAKEN (CONT.):

SIGNIFICANT DELIVERABLES FROM THE JAD WERE:

1) A SOFTWARE POLICY FOR OUR ORGANIZATION
2) A SET OF SEVENTEEN "PREFERRED SOFTWARE PRACTICES" TO SERVE AS SOFTWARE STANDARDS.

NOTE THAT THE HEART OF THE PROCESS THAT THE FSD PREFERRED SOFTWARE PRACTICES SUPPORT IS A PROCESS THAT REQUIRES:

1) MANAGEMENT ATTENTION TO SOFTWARE DEVELOPMENT
2) DEFINED AND DOCUMENTED REQUIREMENTS
3) A DEFINED AND DOCUMENTED SCHEDULE, WITH MILESTONES
4) DEFINITION OF THE SOFTWARE LIFECYCLE AND ENVIRONMENT.

SOFTWARE BENCHMARK

EXAMPLE PROCESS, ACTIONS TAKEN (CONT.):

TOP MANAGEMENT WILL INCLUDE SOFTWARE ENGINEERING AND SOFTWARE QUALIT ASSURANCE ACTIVITIES WITHIN OUR FORMAL AUDIT PROGRAM.

TOP MANAGEMENT HAS CHARTERED A COMMITTEE OF SENIOR SOFTWARE ENGINEER AND SUPERVISORS TO:

1) ADVISE AND RECOMMEND ON SOFTWARE TRAINING PROGRAMS
2) ADVISE AND RECOMMEND ON SOFTWARE QA, AND FUTURE BENCHMARKS
3) ADVISE AND RECOMMEND ON SOFTWARE TOOLS AND TECHNOLOGY
4) LIAISON WITH OTHER KODAK SOFTWARE CENTERS OF EXCELLENCE.

SOFTWARE BENCHMARK

REVIEWING THE PROCESS

SOFTWARE BENCHMARK

REVIEWING THE PROCESS, LESSONS LEARNED:

AN SEI-STYLE SELF-ASSESSMENT CAN BE RUN IN A TIMELY FASHION TO EVALUATE THE STATE OF YOUR SOFTWARE ENGINEERING PRACTICE, USING AN INTERNAL TEAM OF SENIOR SOFTWARE PERSONNEL.

THE BENCHMARK (ASSESSMENT) DATA PROVIDES A QUANTITATIVE BASIS FOR THE INTUITIVE FEELINGS YOU MAY HAVE REGARDING WEAK SPOTS IN YOUR ENVIRONMENT.

USE SEI DEFINITIONS (E.G., MECHANISM) CONSISTENTLY TO OBTAIN UNIFORM BENCHMARK RESULTS ACROSS A SET OF SOFTWARE PROJECTS. YOUR INTERVIEWER TEAM HAS A CRITICAL ROLE TO PERFORM IN THIS AREA.

SOFTWARE BENCHMARK

REVIEWING THE PROCESS, LESSONS LEARNED (CONT.):

CONDUCT "REAL-TIME" INTERVIEWS WITH THE SUPERVISORS, RATHER THAN HAVING THEM JUST ANSWER THE QUESTIONNAIRE UNASSISTED, SO THAT THEIR ANSWERS MAKE STRICT USE OF SEI DEFINITIONS. THIS SHOULD YIELD BETTER CORRELATION WITH WORKERS' ANSWERS TO THE SAME QUESTIONS.

PLANNING IS CRITICAL TO BENCHMARKING SUCCESS, BUT DON'T GET MIRED IN "THE PLANNING TAR PIT."

BE SURE YOU PROVIDE TIMELY FEEDBACK TO BOTH MANAGERS AND PROJECT TEAM MEMBERS.

SOFTWARE BENCHMARK

REVIEWING THE PROCESS, PROS AND CONS:

PROS:

1) WITH STRONG MANAGEMENT BACKING, YOU CAN QUANTIFY YOUR SOFTWARE ENGINEERING PROCESS WITH A BENCHMARK.

2) A BENCHMARK SHOWS YOUR STRENGTHS AND WEAKNESSES; YOU CAN CAPITALIZE ON THE STRENGTHS AND WORK TO IMPROVE WEAK AREAS.

3) THE BENCHMARK PROCESS CAN HAVE CONSCIOUSNESS-RAISING EFFECTS IN YOUR ORGANIZATION.

CONS:

1) THIS PROCESS REQUIRES A SIGNIFICANT INVESTMENT IN TIME AN ENERGY FROM MANY PEOPLE.

2) THIS PROCESS REQUIRES A SIGNIFICANT AMOUNT OF PLANNING; I CANNOT BE ACCOMPLISHED QUICKLY

3) IF A PROJECT YOU WANT TO BENCHMARK IS ON A TIGHT TIMELINE BENCHMARKING COULD IMPACT DELIVERY.

4) YOU SHOULD BE PREPARED TO REPEAT THE BENCHMARK; ASSESSMEI IS A CONTINUOUS PROCESS.

SOFTWARE BENCHMARK

CONCLUSIONS REGARDING BENCHMARKING

SOFTWARE BENCHMARK

CONCLUSIONS REGARDING BENCHMARKING:

BENEFITS RESULTING FROM THE BENCHMARKING PROJECT INCLUDE:

1) RAISED AWARENESS OF THE SOFTWARE COMMUNITY RELATIVE TO SOFTWARE ENGINEERING ISSUES
2) RAISED AWARENESS OF SUPERVISION AND MANAGEMENT AS TO THE WORTH OF DISCIPLINED SOFTWARE MANAGEMENT
3) KNOWLEDGE OF THE STRENGTHS AND WEAKNESS IN THE SOFTWARE PROCES ENABLING MOVEMENT TO EFFICIENTLY IMPROVE THAT PROCESS.

SOFTWARE BENCHMARK

SUMMARY

SOFTWARE BENCHMARK

SUMMARY:

AN ORGANIZATION THAT IS SERIOUS ABOUT IMPROVING THE QUALITY OF ITS SOFTWARE ENGINEERING PROCESS MUST BEGIN WITH AN ASSESSMENT OR BENCHMARK TO ESTABLISH ITS CURRENT STATE-OF-THE-PRACTICE.

AT KODAK'S FEDERAL SYSTEMS DIVISION, TOP MANAGEMENT PERCEIVED THE NEED FOR IMPROVEMENTS IN OUR PROCESS, AND ASKED FOR A BENCHMARK.

THAT SOFTWARE ENGINEERING BENCHMARK INDICATED THAT WE DID NOT HAVE A UNIFORM, SYSTEMATIC PROCESS TO USE IN DEVELOPING SOFTWARE.

WE TOOK ACTION TO IMPROVE OUR SOFTWARE ENGINEERING PROCESS.

SOFTWARE BENCHMARK

SUMMARY (CONT.):

A TEAM DEVELOPED A POLICY AND SET OF PREFERRED SOFTWARE PRACTICES TO DESCRIBE/SPECIFY THE FSD PROCESS.

THE POLICY AND PRACTICES HAVE BEEN APPROVED AND ENDORSED BY THE MANAGEMENT OF FSD. WE EXPECT THIS WILL RESULT IN IMPROVED SOFTWARE DUE TO OUR IMPROVED PROCESS.

REFERENCES

SOFTWARE BENCHMARK

REFERENCES:

THE SOFTWARE ENGINEERING INSTITUTE AT CARNEGIE MELLON UNIVERSITY FOR INFORMATION ON SELF-ASSESSMENTS. PHONE TIM KASSE, 412-268-5877.

"THE SOFTWARE ENGINEERING INSTITUTE", MARCH 1989 ISSUE OF THE AMERICAN PROGRAMMER, CHILDREN'S COMPUTER COMPANY, LTD.

"MAKING SOFTWARE ENGINEERING HAPPEN", DR. ROGER PRESSMAN, PRENTICE HALL, COPYRIGHT 1988

FEDERAL SYSTEMS DIVISION

Paper 4-T-1

SOFTWARE TOOLS FOR VALIDATION AND VERIFICATION OF ANSI-C COMPILERS

Dr. Manoochehr Ghiassi
Associate Professor of Information Sciences
Santa Clara University

Prof. Manoochehr Ghiassi is currently an Associate Professor of Information Systems in th e Department of Decision and Information Sciences at Santa Clara University. He received a B.S. from Tehran University, Iran, in 1970 and an M.S. in Economics from Southern Illinois University at Carbondale, Illinois, in 1974. He also received an M.S. in Computer Science in 1979 and Ph.D. in Industrial Engineering in 1980, both from the University of Illinois at Urbana-Champaign. Dr. Ghiassi teaches courses in Structured Programming, Operating Systems, Computer Simulation and Management Information Systems. His current research interests involve software engineering and simulation modeling. He has published several papers in IEEE, AIIE, Computer Design, and Operations Research publications. He has consulted for AT&T, U.S. Army, National Semiconductor, Signetics and Microtec Research on a variety of software engineering and simulation projects. He has also developed methodology and tools necessary for some of the local computer companies to establish a software quality assurance program for validation and verification of system software ranging from language tools (compilers, assemblers, debuggers, etc.) to operating systems.

Software Tools for Validation and Verification of ANSI C Compilers

M. Ghiassi, Ph.D.
Associate Professor of Information Sciences
Santa Clara University
Santa Clara, CA 95053

Abstract

Developing reliable, high performance ANSI C compilers require the availability of software tools specifically designed for testing such compilers. This paper presents a "testing view" of the compiler development process and identifies those tools necessary for the validation and verification of C compilers during each stage of this process.

Keywords: C compiler, compiler validation, software tools, software quality assurance.

Introduction

The American National Standards Institute's (ANSI) recent approval of the standard for the C programming language [2], the introduction of new and powerful microprocessors for workstations based on RISC (Sun-SPARC, MIPS, Motorola 88000, etc.) and CISC (Intel 80x86, Motorola 680x0, etc.) architectures, and the use of dedicated microprocessors in new markets such as imaging, desktop publishing, and robotics have all contributed to an increased demand for reliable, high performance native and/or cross C compilers. To achieve this reliability and high performance, software tools are needed that can validate and verify ANSI C compilers. This paper presents a "testing view" of the compiler development process using a hybrid testing approach of white-box and black-box testing methods. Tools needed for the validation and verification of C compilers during each stage of this development process are identified.

Testing View of Compiler Development Process

To produce high quality compilers, quality must be designed in from the start. To design quality into a complex software product such as a compiler, testing must be a paramount objective throughout the entire development process.

The testing process starts with the development of a plan that reflects a "testing view" of the compiler construction process. The test plan outlines the testing responsibilities and activities of the compiler development team, the software quality assurance team (SQA), the technical support group, and an independent testing team. The plan states when each phase of the testing process should take place and the stopping rules for completion of the test process. It also describes software tools needed to achieve successful compiler validation and verification.

Figure 1 shows the testing view of the compiler development process.

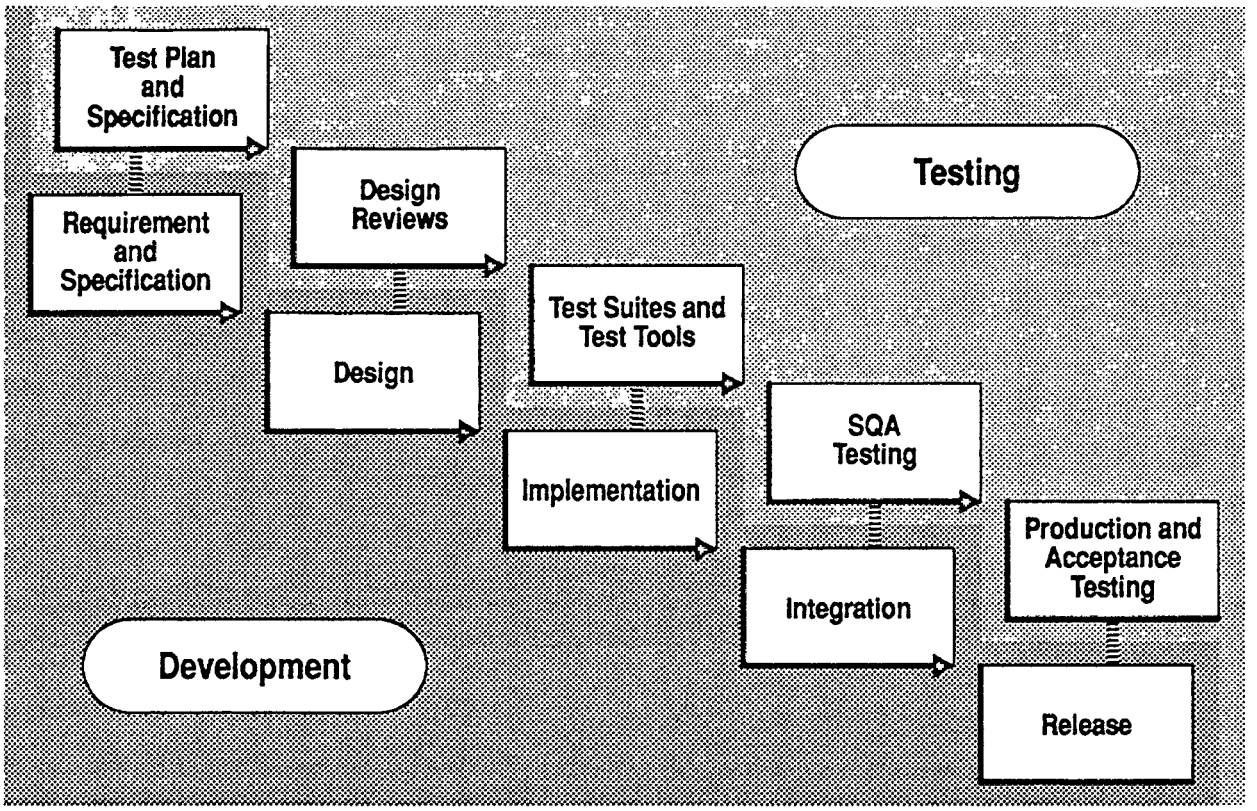


Figure 1. Testing View of the Compiler Development Process

The next sections describe in more detail the testing activities of each group and identifies those software tools required to implement testing at each step.

Test Plan Development

As mentioned previously, the first step in the C compiler validation process is the development of a test plan. The SQA group is responsible for the development, execution, and completion of this plan. The plan gives the overall structure and schedule of the testing activities required of each team member from the compiler development team through the independent testing team.

A major component of the test plan is to set up validation goals for each group and stage of the validation process. The validation goals include correctness, accuracy and precision, efficiency, test coverage and completeness, and acceptance goals. The test plan also identifies software tools necessary for achieving these goals. Some of these tools can be obtained from outside resources such as commercial and public domain software, and other tools need to be developed by the test groups.

Development Testing

Development testing begins at the design stage of the compiler development process. The availability of a well-defined language specification such as the ANSI C standard document [2] facilitates the validation process. This document is the primary specification for C compiler validation. It should be augmented with implementation-dependent information and extensions and exceptions allowed by the language implementation. The ANSI document and its augmentations serves as the basis for test development by the development team and other groups.

Development testing follows a white-box approach. The initial focus of development testing is to satisfy the correctness goal of the validation process. This goal starts with traditional review techniques such as design reviews, code reviews, and walk throughs that manually test the design and coding of the software. Software development tools such as UNIX "lint" can also assist the development engineers in developing portable software. This portability is important to engineers who are developing cross compilers that will be ported to other hosts.

Additionally, the development team requires an automated test suite that can easily and selectively validate implementation of the components of the software as they are developed. Test suites such as the Plum Hall ANSI C validation suite [14] and the Perennial test suite for the C programming language [13] are commercially available. The Plum Hall test suite, in particular, provides a test generator. This generator can generate many test cases that exercise variations and mutations of the language constructs. Other similar test suites are also available. These suites use the ANSI C standard document as their reference point and have a large set of test programs that will automatically and selectively test each and every structure of the language. Most of these test programs are small to medium size and are self checking. The suites also provide a final report summarizing the test results.

These test suites do not test for extensions to the language that an implementation has introduced. The ANSI standard itself allows for some language constructs to be implementation defined. Obviously, generic test suites will not be able to check for the accuracy of such cases. Test cases that validate such constructs need to be developed and included in the development testing process.

Software Quality Assurance Testing

The software quality assurance testing of the compiler begins at the design phase of the development process. One or more SQA engineers need to be assigned to the project. Their activities include both test development and test execution.

Additional tests beyond the test suites are developed or gathered by this group. Tests for extensions and exceptions to the language should also be designed and developed by this team to provide an independent evaluation of these features. Aside from this task, which is based on the white-box approach to testing, the SQA testing follows the functional, black-box testing approach.

Alpha testing is also conducted in conjunction with the SQA testing. To start SQA and alpha testing, the first step is to establish the unit under test (UUT). The method recommended here is the compiler self-booting process. The self-booting process begins with the source code for the compiler and a production-level C compiler on the host computer. The first generation binaries for

the UUT is produced by compiling the UUT with this production compiler. Separate procedures are needed for the native and cross compilers with the cross compilers requiring downloading capabilities to the target board as well as some I/O support system. In a native environment, the source for the UUT is compiled using the first generation compiler binaries to produce second generation binaries. If successful, the process is repeated by replacing the first generation binaries with the second generation to produce the third generation binaries. If the second and third generation binaries are identical, the UUT is stable. The fact that it has bootstrapped itself is further evidence of its functionality. The source for this release of the compiler is then frozen and copies are distributed for both alpha testing and SQA testing.

The alpha testing of the compiler is performed by either the SQA team members or software engineers other than the developers of the software. One effective use of the alpha testing is to compile the supporting language tools that will be used with this compiler. These tools include the assembler, libraries, and debuggers. Successful compilation provides further evidence of compatibility with other members of the language tools.

The SQA testing team may choose to select and execute portions of the test suites used by the development team for assurance purposes. In particular, the SQA testing should concentrate on testing features of the compiler that are not covered by the generic test suites. Candidate components include the compiler driver, the optimizer, the libraries, and architecture- and application-specific features.

Compiler drivers provide an automatic alternative to the compilation process. These drivers usually provide a set of switches that are unique to the compiler. Test programs must be developed that validate the correct behavior of these switches. In particular, test programs must be designed and developed that examine the behavior of the driver when the driver is invoked with multiple switches.

Optimizers also need to be tested for correctness and efficiency. The correctness of the optimizers can be demonstrated during the overall compiler testing process. Test programs are compiled with and without optimization in effect and the results are compared for correctness. It is more difficult, however, to test the efficiency of optimizers. Small programs are usually too trivial for the optimizers. Large programs may demonstrate an improvement in either speed or space, but the difficulty lies in measuring the improvements achieved versus the improvements expected. Deciding on how much improvement to expect requires detailed knowledge of how the optimizer was designed, a tedious task that requires examination of both the algorithms used and the coding scheme selected to design and develop the optimizer. While it is difficult to measure the efficiency of optimizers, it is not impossible. This efficiency can be measured during the performance evaluation of the compiler and is described in more detail later in this paper.

Testing for the accuracy and precision of compilers requires specific software. Software tools that verify conformance to the ANSI/IEEE std 754 standard have been developed and are available through public domain software. In particular, one package appeared in the Communications of ACM [5, 11], and the other is a number of software tools (paranoia, elefunt, etc.) resulting from the efforts of a University of California at Berkeley group [10]. These tools should be used to achieve the accuracy and precision goal of the compiler validation process.

Architecture-Specific Testing

Architecture-specific testing examines the uniqueness of the individual microprocessors. For example, the Intel 8086 family of microprocessors uses segmentation schemes to control memory addressing. In this architecture, addresses are calculated using a segment register and a 16-bit offset. The CPU provides four different segment registers. These segment registers are used for the various types of memory access which is unique to the 8086 family of compilers. In C, the keywords near (offset only) and far (segment:offset) express the mode of addressing to be used. Different memory models are also available with most 8086 C compilers. Test cases that examine memory addressing and different memory models need to be developed to ensure the correctness of the 8086 family of C compilers. Other microprocessors have their own unique architectural anomalies requiring similar treatment.

Application-Specific Testing

Most of the testing performed so far has focused on unit and module testing. SQA testing should also perform application-specific, system-level testing. This testing should include compilation of large and complex software. Several public domain software programs such as the EMACS and Rogue programs are available that can be used for this testing. These programs are considered to have complex C constructs not typically found in most C programs and will help to exercise the compiler. These programs should be compiled and the resulting executable tested for correctness and efficiency. These tests will also demonstrate the behavior of the compiler when given large input. Successful compilation of these programs should build confidence in the compiler and thus encourage use of it by other groups while additional testing is in progress.

Application-specific system testing of compilers varies depending on whether the compiler is a native or cross version. The native C compiler system testing could be achieved by building the UNIX operating system. Compiling the entire source to UNIX is considered to be the ultimate test of a native C compiler. One problem with testing such a large software system is the verification of the compiled version for both correctness and efficiency. The procedure recommended here requires a careful consideration of the build process. The process begins by first establishing a well-defined reference system for future comparison. Consider a System V based UNIX system. The first step is to obtain a UNIX validation suite such as the System V Validation Suite (SVVS) [20]. This test suite should be run to completion on the reference system and the results saved for future comparison. Additionally, other suites that will measure the performance level of the UNIX operating system should also be obtained and run. The AIM benchmark is an excellent candidate for such evaluation. This program should also be compiled and executed on the same reference system. The native C compiler on the reference system can now be replaced with the new C compiler. The entire source to the UNIX operating system can now be compiled with the new C compiler. Once the UNIX source is successfully compiled, the reference system can be booted with the new binaries. The same UNIX validation suites (SVVS and AIM) are run once more and the results are compared with the previous ones. This process will measure both the correctness and the efficiency of the native C compiler.

A final consideration for native C compiler validation is conformance to the System V Interface Definition (SVID) [19]. Compiler options should be examined to ensure their conformance to such defacto standards.

Livermore Loops, Puzzle, Sieve, Stanford, and Whetstone. These benchmarks are used to measure compiler performance and are also used for compiler tuning. Although useful, these small benchmarks are losing their appeal. Current compiler technology has resulted in compilers that are well tuned toward achieving high results for these specific benchmarks. Real-life applications of such compilers often fail to achieve such high performance levels.

Recently, a new set of large and complex benchmarks have been developed by the System Performance Evaluation Cooperative (SPEC) [17]. The SPEC benchmarks can be used to measure the system-level performance of compilers and provide a more meaningful basis for evaluating such compiler efficiency measures. The C portion of the current release (release 1.0A) includes sources for the GNU C compiler named "gcc", a software package called "Espresso" that performs CAD operations, a LISP interpreter called "Li" that is written in C, and an integer-intensive program called "Eqntott" that requires YACC and which performs sorting. These tools are each more than 1/4 Megabytes in size and are used to measure the various operations of a C compiler. A set of reference values is also provided that will allow for easy comparison with similar compilers.

Beta Testing

The compiler under test is now ready for distribution to selected users for beta use. There are several advantages to beta testing. One advantage of beta testing is the use of the compiler by independent groups outside the development organization and under real-life circumstances. This testing will expose the new C compiler to a wide variety of users and application programs. In this step, the compiler is used as a limited production compiler. Another advantage of beta testing is exposure of the compiler to a diverse group of applications. Beta sites, therefore, should be selected to reflect wide areas of application for the compiler.

Production and Acceptance Testing

The final step in C compiler validation is production and acceptance testing. In this step, the compiler is viewed as a complete product and is tested as such. The first step is to check for completeness of the product which includes a complete building from the source and archiving the source of the compiler and its documentation. Several additional test procedures need to be developed to ensure that the compiler can be easily verified for completeness. Self-checking test programs should accompany the compiler that will automatically verify complete installation of the software. A multi-user test of the compiler should also be performed. The multi-user test of the compiler is achieved when several users are assigned to use the compiler over some time period. This testing examines how the C compiler utilizes the underlying operating system including the number of temporary files opened, the scratch and swap space requirements, and the memory requirements with or without optimization. Also examined is how the performance of the compiler is affected when several users use it simultaneously.

Additional production testing includes stress and compatibility testings. Stress testing is used to identify the boundary values of compiler parameters such as the number of external names in one source file, number of local variables per block, number of macros per source file at any time, number of characters in a "logical" source line, etc. Backward compatibility tests should include a test of the compiler's compliance to the "K&R" [9] definition of the C compiler as well as compatibility with supporting language tools such as assemblers and debuggers.

Finally, the testing team as a whole needs to review the entire test process to ensure that all the defects found during the testing are resolved. This team should also make sure that a complete record of outstanding issues are documented in the release notes that will accompany the C compiler to the final users.

After production testing, acceptance goals are examined to see if further testing is required. Test coverage and analysis tools should be used throughout the testing process to provide informative metrics that will measure the confidence level achieved as a result of the test process. Tools such as Software Research's Test Coverage and Analysis Tool (TCAT) [18] and McCabe & Associates' Analysis of Complexity Tool (ACT) [1] provide test coverage analysis and are commercially available. These tools show the extent to which the test suites have exercised all of the segments of the compiler. A coverage of 90% and above is considered to be adequate for complex software programs such as C compilers.

Conclusions

This paper introduced a testing view of the compiler development process. It presented a set of goals for validating ANSI C compilers and outlined the activities of the testing teams. Software tools needed for this validation during each stage of the development process were also identified.

Acknowledgments

The author would like to thank Lucille Ching for her comments and Microtec Research, Inc. for their support of this research.

References

1. ACT, "The Analysis of Complexity Tool," McCabe & Associates, Inc., Maryland.
2. ANSI/X3.159-1989, ANSI Standard for the Programming Language C.
3. ANSI/IEEE std. 754-1985, IEEE standard for Binary Floating-Point Arithmetic.
4. Gay, D. M. and T. Sumner, "Floating Point Test 'Paranoia'," A C version of Kahan's Floating Point Test "Paranoia", Dept. of EECS, University of California, Berkeley, California. 1986.
5. Gentleman, M. and S. Marovich, "More on Algorithms that Reveal Properties of Floating-Point Arithmetic Units," Communications of the ACM, Vol. 17, No. 5, pp. 276-277.
6. Ghiassi, M., "Software Quality: Design it in from the Start," Computer Design, Vol. 23, No. 9, July 1985, pp. 91-97.
7. Ghiassi, M., "Validating Optimizing Compilers," Proceedings of the 1987 IEEE-WESCON Conference, Vol. 31, No. 2, pp. 1-13.
8. Ghiassi, M. and M. S. Arratoon, "Developing a C compiler Validation Suite," Proceedings of the 1985 IEEE-WESCON Conference, Vol. 29, No. 37, pp. 1-7.
9. Kernighan, B. W., and D. M. Ritchie, "The C programming Language," Prentice-Hall, 1978.
10. Lui, Z., "Floating Point Tests," Dept. of EECS, University of California, Berkeley, California.

11. Malcolm, M., "Algorithms to Reveal Properties of Floating-Point Arithmetic," Communications of the ACM, Vol. 15, No. 11, pp. 949-951.

12. Myers, G. J., "The Art of Software Testing," New York, Wiley, 1979.

13. Perennial, "C compiler Validation Suite," Perennial Corporation, Santa Clara, California.

14. Plum Hall, "The Plum Hall Validation Suite," Plum Hall Inc., Cardiff, New Jersey.

15. Pressman, R. S., "Software Engineering," 2nd Edition, New York, McGraw-Hill, 1987.

16. Seyfer, H. K., "Tailoring Testing to a Specific Compiler — Experiences," ACM Proc. SIGPLAN Symp. on Compiler Construction, 1982.

17. SPEC, "SPEC benchmark Suite Release 1.0a," System Performance Evaluation Cooperative, Fremont, California.

18. TCAT, "Test Coverage and Analysis Tool," Software Research, Inc., San Francisco California.

19. "UNIX System V Interface Definition," AT&T, 1986.

20. "UNIX System V Validation Suite," AT&T, 1985.

21. Wichmann, B. A. and Z. J. Ciechanowicz, "Pascal Compiler Validation," Wiley, 1983.

Paper 4-T-2

PRACTICAL MODULE REGRESSION TESTING TOOLS & TECHNIQUES

Prof. Daniel Hoffman
Dept. of Computer Science
University of Victoria

Dr. Daniel Hoffman received the Ph.D. degree in computer science in 1984, from the University of North Carolina, Chapel Hill, and is currently an Assistant Professor of Computer Science at the University of Victoria in B.C., Canada. His research area is software engineering, emphasizing software specification and testing.

Practical Module Regression Testing Tools and Techniques

Dr. Daniel Hoffman

University of Victoria
Department of Computer Science
Victoria, B.C. Canada

Talk Overview

- Modules and interfaces
- Testing strategy
- Test case description
- Test harness generation
- Design for testability
- Experience
- Conclusions

Modules and interfaces

- Module

  A programming work assignment

- Module interface

  The set of assumptions users are permitted to make about the module

- Call-based interface

  Communication through *access programs*

- Exception handling

  Exceptions explicitly specified, detected, signaled

- Module traces

  Sequences of calls on the module

Symbol Table Example

Acc. pgms.	Inputs	Outputs	Exceptions
s_init			
s_addsym	string		maxlen tblfull
s_delsym	integer		
g_cnt		integer	
g_legsym	string	boolean	
g_legid	integer	boolean	
g_sym	integer	string	notlegid
g_id	string	integer	notlegsym

- Two traces

```
s_init().s_addsym("cat").g_legsym("cat")
s_init().s_addsym("cat").g_id("dog")
```

Testing Tasks

(1) Build test harness

(2) Select inputs

- Structural coverage, dataflow coverage
- Functional testing

(3) Determine expected outputs

- The *test oracle* problem

(4) Execute tests

(5) Compare actual outputs against expected outputs

(6) Evaluate results

○ Costs of all 6 steps must be considered

Testing Principles

- Systematic module testing
  - For both development and maintenance
- Design for testability
  - Module designer responsible for ensuring testability
  - Controllability and observability
- Cost-effective automation
  - Test harness construction
  - Test execution
  - Actual output checking
- Practical constraints
  - Widely applicable now or in near future
  - Minimal special training required
  - Not dependent on compiler modifications

Test Case Description

- Test case syntax

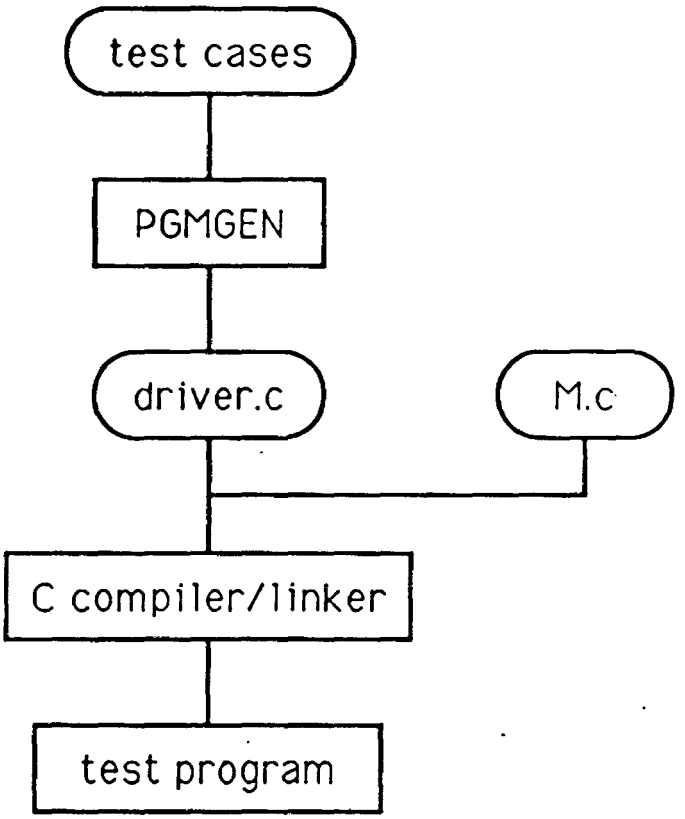
  $< trace, expexc, actval, expval >$

  - *trace*
    trace used to exercise the module
  - *expexc*
    exception that *trace* is expected to generate
  - *actval*
    expression evaluated after *trace*
    — the "actual value" of the trace
  - *expval*
    the value that *actval* is expected to have

- Two test cases

```
<s_init().s_addsym("cat"),
     noexc, g_legsym("cat"), 1, boolean>

<s_init().s_addsym("cat").g_id("dog"),
     notlegsym, dc, dc, dc>
```

Test Harness Generation



Test Harness Generation

Generate code to record exception occurrences

For each test case of the form:

$$< c_1.c_2.....c_N, expexc, actval, expval, type >$$

generate code to:

```
invoke c1, c2, ..., cN
compare actual exception occurrences against expexc
if there are any differences
        print a message
else
        if actval ≠ expval
                print a message
        if exceptions occurred since cN was invoked
                print a message
update summary statistics
```

Generate code to print summary statistics

Design for testability — case study

- The GRADES system
  - Maintain assignment, student, score information
  - Spreadsheet-style screen editor
  - Detail and summary reports
  - Implemented in C; 19 modules; 10K lines source

- Applying design for testability
  - Initialization calls
  - Controllability problems

    Keyboard input
  - Observability problems

    Screen, file output

Design for testability — case study

- Call-based input/output — 9 modules
  - Few controllability or observability problems
  - Many test cases required to handle input combinations
- Keyboard input — 2 modules
  - Controllability: keystrokes supplied manually
  - Observability: echoing checked manually
  - Approach: isolate; test manually
- Screen output — 5 modules
  - Observability: screen contents checked manually
  - Approach: isolate; display patterns
- Report output — 3 modules
  - Observability: output difficult to check from driver
  - Approach: isolate; use diff

Experience

- Modules tested
  - Over 50 modules of various kinds
  - Cost
- Structural coverage
  - Statement coverage necessary, *not* sufficient
  - With system testing —
    100% coverage difficult to achieve
  - With module testing —
    100% coverage easily attained
- Design for testability
  - Identify C & O problems early
  - Eliminate where possible
  - Isolate problems that cannot be eliminated
- Teaching testing
  - Scripts written by instructor
  - Scripts written by students

Conclusions

- Standardized module interfaces
  - For reusable test scaffolding
  - As the basis for automated support
- Systematic module testing
  - Trace-based test case language
  - Automatic driver generation
- Design for testability
  - Designer responsible for ensuring testability
  - Importance of controllability and observability
- Work underway
  - Industrial applications
  - Executable test oracles

REFERENCES

Scaffolding construction

[1] D.J. Panzl. Automatic software test drivers. *Computer*, 11(4):44–50, April 1978.

[2] M.M. Gorlick, C.D. Kesselman, D.A. Marotta, and D.S. Parker. Mockingbird: a logical methodology for testing. *Journal of Logic Programming (to appear)*, 1989.

[3] A. Jagota and V. Rao. TCL and TCI: a powerful language and an interpreter for writing and executing black box tests. In *Proc. Pacific Northwest Software Quality Conf.*, pages 147–166, IEEE Computer Society, 1986.

Input selection

[1] W.E. Howden. Functional program testing. *IEEE Trans. Soft. Eng.*, SE-6(2):162–169, March 1980.

[2] J.C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, September 1975.

[3] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Soft. Eng.*, SE-11(4):367–375, April 1985.

Test oracles

[1] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, July 1981.

[2] W.T. Tsai, D. Volovik, and T.F. Keefe. Automated test case generation for programs specified by relational algebra queries. *IEEE Trans. Soft. Eng.*, SE-16(3):316–324, March 1990.

Available from the speaker on request

[1] D.M. Hoffman. A CASE study in module testing. In *Proc. Conf. Software Maintenance*, pages 100–105, IEEE Computer Society, October 1989.

[2] P. Brown and D.M. Hoffman. Application of module regression testing at TRIUMF. In *Proc. Intl. Conf. Accelerators and Large Experimental Physics Control Systems*, November 1989.

[3] D.M. Hoffman and C. Brealey. Module test case generation. In *Proc. 3rd Symp. on Software Testing, Analysis, and Verification*, ACM SIGSOFT, 1989.

[4] D.M. Hoffman. On criteria for module interfaces (to appear 5/90). *IEEE Trans. Soft. Eng.*, 1990.

Paper 4-T-3

DEVELOPER TESTING VS. QA TESTING: WHO DOES WHAT AND WHY?

Mr. Keith Stobie
Senior Technical Staff Member
Tandem Computers

Mr. Keith Stobie is a Senior Technical Staff member within Software Development QA at Tandem Computers. He received his BSCS from Cornell University in 1979, and spent the next several years as a Quality Assurance Developer at Tandem, where he positively impacted product quality in a variety of Online Transaction Processing products, including Transaction Monitoring Facility (TMF), TRANSFER, and Distributed Systems Management (DSM) products. Keith was a founding QA developer of the Systems QA group. Keith is a leader in testing methodology and tools technology at Tandem resulting in high levels of testing automation today. Keith is a direction setter in the areas of training including formal classes, a lecture series and informal discussion sessions. Keith brough formal Inspections to Tandem and has worked towards their implementation and acceptance.

QW-90 - Quality Week 1990

Developer testing VS QA testing
Who does what and why

Many books and articles describe types of testing: unit, system, structural, functional, white-box, black-box, internal, external, statistical, anecdotal, etc. But few address who does what kind and why. This talk will address similarities and differences between the types of testing by various organizations.

It discusses the kind of testing an in-house independent QA organization or a third party independent V&V organization might do. Also the kind of testing individual developers should do of their own code and of their colleagues and team members code. The purpose of the types of testing the various groups do will be delineated.

Keith Stobie
TANDEM COMPUTERS
10555 Ridgeview Court, LOC 100-11
Cupertino, CA 95014

Keith Stobie is a Senior Technical Staff member within Software Development QA at Tandem Computers. He received his BSCS from Cornell University in 1979, and spent the next several years as a Quality Assurance Developer at Tandem, where he positively impacted product quality in a variety of Online Transaction Processing products, including Transaction Monitoring Facility (TMF), TRANSFER, and Distributed Systems Management (DSM) products. Keith was a founding QA developer of the Systems QA group.

Keith is a leader in testing methodology and tools technology at Tandem resulting in high levels of testing automation today. Keith is a direction setter in the areas of training including formal classes, a lecture series and informal discussion sessions. Keith brought formal Inspections to Tandem and has worked towards their implementation and acceptance.

Developer testing VS QA testing

Who does what and why

TANDEM COMPUTERS
19333 Vallco Parkway
Cupertino, CA 95014

Keith Stobie

Developer testing
VS
Independent testing

Who tests?
Why test twice?

Many books and articles describe types of testing: unit, system, structural, functional, white-box, black-box, internal, external, statistical, anecdotal, etc. But few address who does what kind and why. This talk will address similarities and differences between the types of testing by various organizations.

It discusses the kind of testing an in-house independent QA organization or a third party independent V&V organization might do. Also the kind of testing individual developers should do of their own code and of their colleagues and team members code. The purpose of the types of testing the various groups do will be delineated.

Testing?

Can testing be avoided?
- Proof of correctness
- Cleanroom

Independent Testing?
- Government (DOD)
- Industry

Independent QA
- Government (DOD)
- Industry

TANDEM COMPUTERS 4/16/90 2

This talk assumes a large amount of 'traditional' code testing will be performed. Techniques like Proof of correctness [Date74] and Cleanroom will not be further addressed. Cleanroom allows no programmer testing and only randomly generated tests applied by an independent group [Selb87]

This talk deals with the Testing tasks within a company. DOD may be able to afford independent verification & validation, IV&V, but it is very rare in industry.

Similarly it is assumed that QA or auditing, is a company function, not an independent function.

Characteristics

Product Developer <-> Product Development <-> Product QA

Design Testing

- Externals vs Internals

Code Testing

- Level of Testing
  - Subroutine, Module, Program, Product, Integration, System
- Structural Coverage
  - Path, Branch, Statement
- Repeatability
- Externals vs Internals
- Test Tracking
- Product Statistics

Documentation Testing

- Externals vs Internals

TANDEM COMPUTERS 4/16/90 3

At least two different possible goals for testing:

1) Defect Removal
2) Measurement
   i) Defect Density [Haml89]
   ii) Reliability (MTBF) [Musa87]

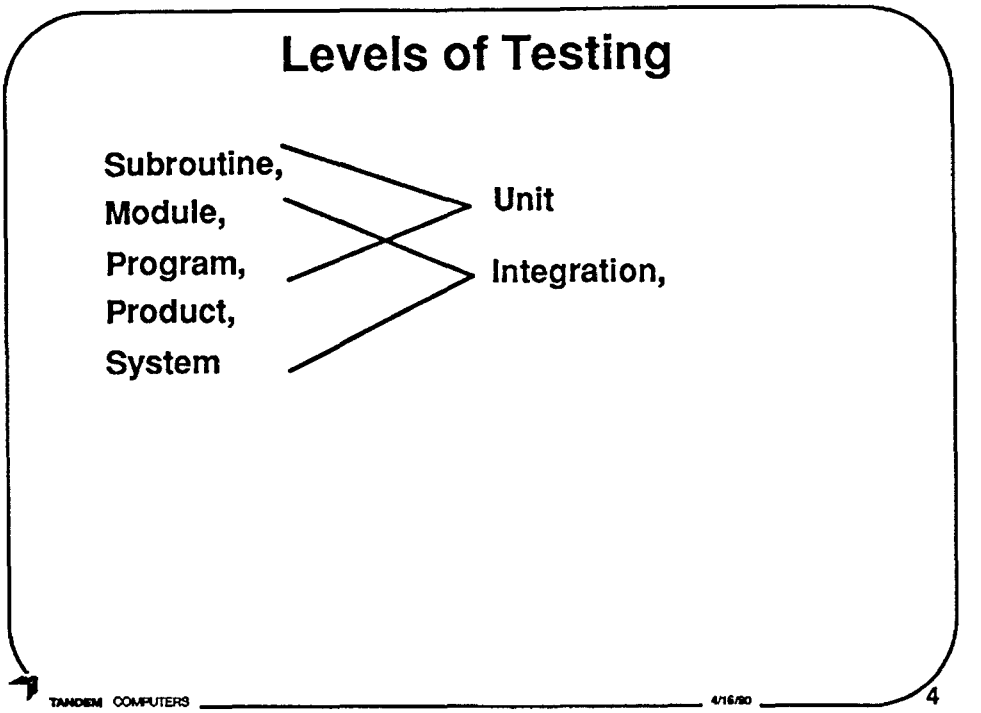
Design Testing

Code Testing

- Level of Testing
  - Developer (Subr, Mod, Prog)
  - Development (Prod, Int), QA (Prod, Int)
  - Gremlin/Alpha/Beta (System)
- 100% Branch Developer <-> 80% Statement QA
- Repeatability is everyone's goal
- QA External view <-> Developer/ment Internal view
- Test Documentation & Defect Tracking
- Stats: Memory, Disc, performance

Documentation Testing

Will not discuss Design & Doc testing unless time permits



"Software unit testing is a process that includes the performance of test planning, and acquisition of a test set, and the measurement of a test unit against its requirements."

TEST UNIT

"A set of one or more computer program modules together with associated control data, (for example, tables), usage procedures and operating procedures that satisfy the following conditions:

1) All modules are from a single computer program
2) At least one of the new or changed modules in the set has not completed unit test
3) The set of modules together with its associated data and procedures are the sole object of a testing process."

"A test unit may occur at any level of the design hierarchy from a single module to a complete program. Therefore, a test unit may be a module, a few modules, or a complete computer program along with associated data and procedures."

"A test unit may contain one or more modules that have already been unit tested."

[IEEE Standard for Software Unit Testing, IEEE87]

"A unit is the work of one programmer." [Biez84]

Subroutine

```
{ index -- find position of a character c in
  string s }
function index( var s: string; c: character ) :
  integer;
var
    i : integer;
begin
    i := 1;
    index := 0;
    while (s[i] <> c) and (s[i] <> ENDSTR) do
        i := i+1;
    if (s[i] <> ENDSTR) then
        index := i
end;
```

Adapted from *Software Tools in Pascal*, P. 48

TANDEM COMPUTERS 4/16/90 5

Subroutines are the smallest parts of a language to test. Other names: Procedures, Functions

Usually tested by a 'driver' that calls the procedure with various parameters.

This example is used in the Coverage Graphs.

Module

Symbol Table

table := Create_Table(size);
Insert(Symbol, value)
value := Retrieve(Symbol)
Delete(Symbol)
Destroy(Table)

TANDEM COMPUTERS 4/16/90 6

Modules are a basic concept now. Some languages, Ada, Modula, OOPs, e.g. C++, SmallTalk, have syntactic support.

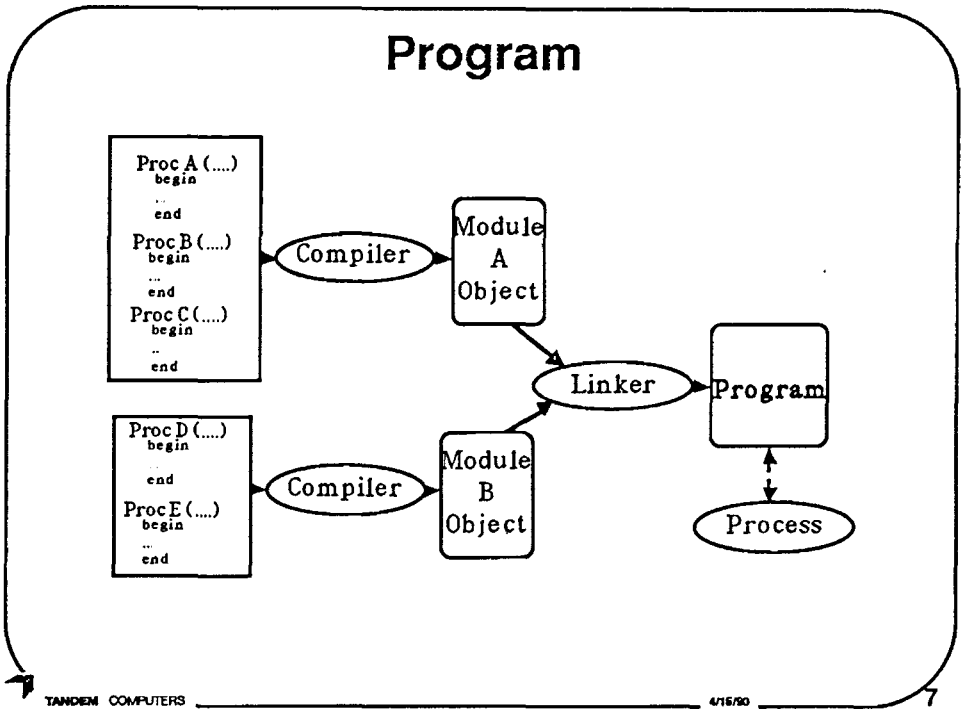
Frequently a Module is separately <u>compilable.</u>

A module is a 'programming work assignment' [Hoff89]. Most common lowest level of testing by programmers.

"(1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine. (ANSI)

(2) A logically separable part of a program."

[IEEE Standard Glossary of Software Engineering Terminology IEEE87]

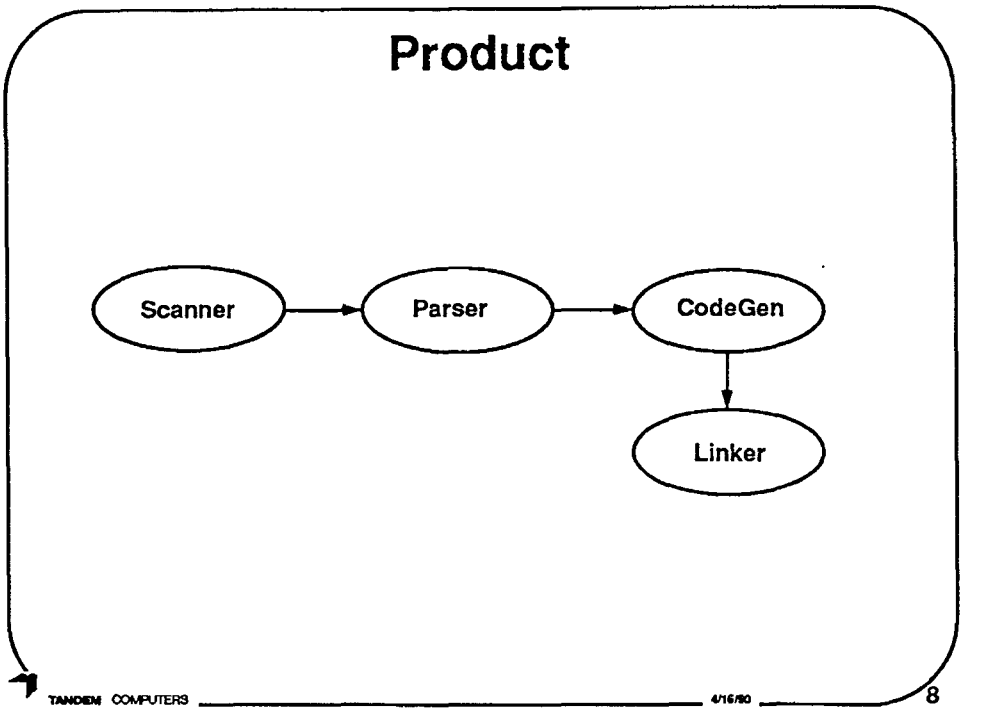


A Computer Program:

A sequence of instructions suitable for processing by a computer. Processing may include the use of an assembler, a compiler, an interpreter, or a translator to prepare the program for execution as well as to execute it. See also (ISO) program.

[IEEE Standard Glossary of Software Engineering Terminology IEEE87]

A Program is instantiated as a Process when it is run.

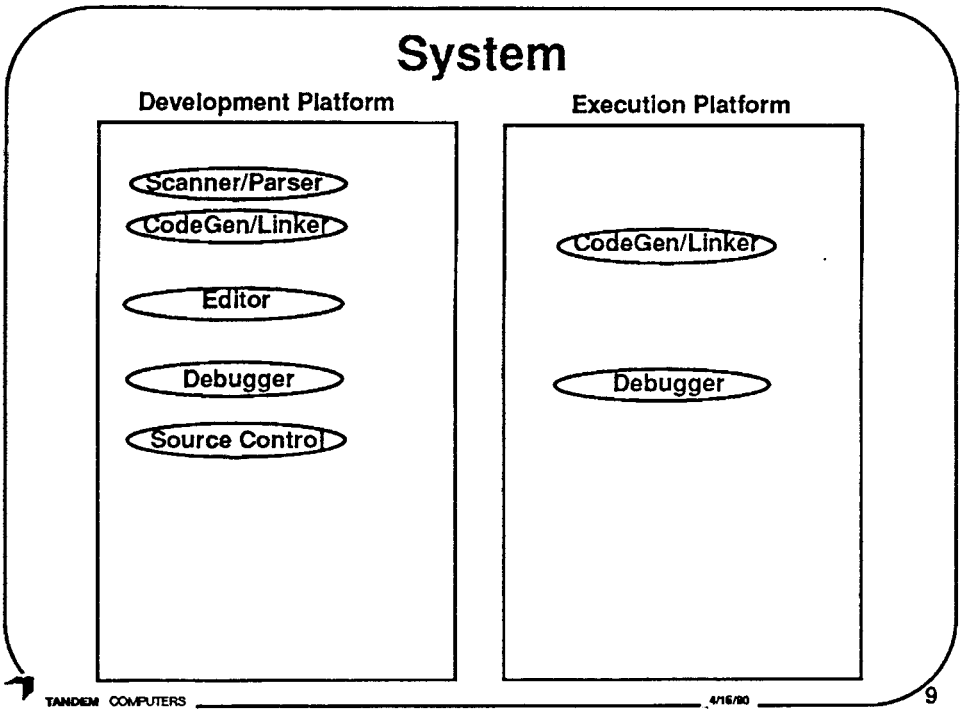


A product is usually something that is sold separately by a company.

In general terms, this also includes the documentation, marketing sheets, etc.

There may not be any distinction between program and product, if the product is only one program.

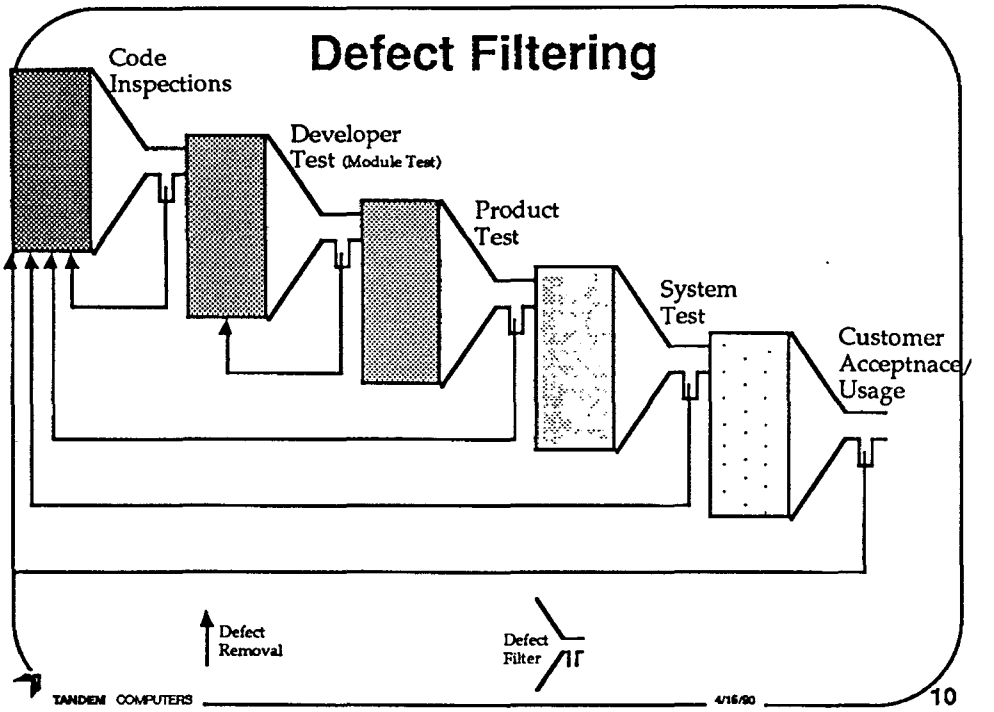
"A **subsystem** is a collection of programs that as a collection accomplish some specified processing." [Beiz84]



Interaction among products

Interaction among different nodes, platforms, etc.

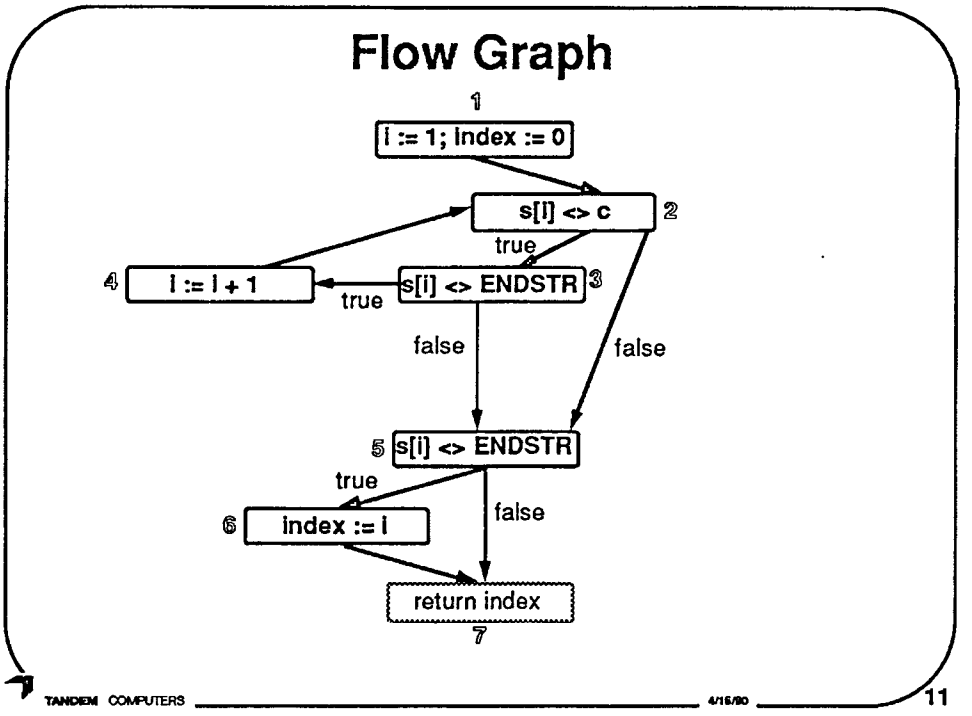
May use customer applications



Picture shows code having fewer and fewer defects as defect removal filters are applied. Number of defects is represented by number of dots (shading)

The amount and type of testing is a cost / risk tradeoff.

You should first select the most effective techniques (e.g. Inspections >60% of defects caught). You should add or tailor other techniques to complement the others for the most cost effective process. For example, Inspections don't find many timing and interface defects. Perhaps Product Test should be geared for these.



100% Statements:

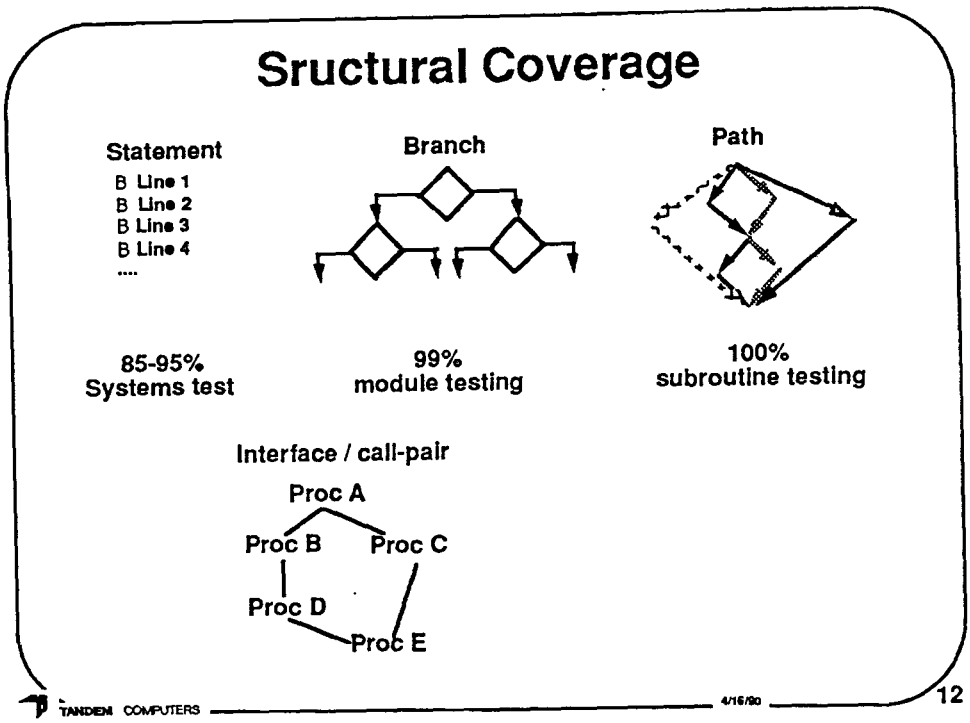
1) s = 'ab' , c= 'b' 1 2 3 4 5 6

100% Branches

1) 2 true, 3 true, 2 false, 5 true
2) s = '', c = 'b' 1 2 3 5
   2 true, 3 false, 5 false

100% Paths

1) 1⇨2⇨3⇨4⇨2⇨5⇨6⇨7
2) 1⇨2⇨3⇨5⇨7
3) s = 'a', c='a'
   1⇨2⇨5⇨6⇨7



Statement is the most basic. Has every statement in the program been executed at least once.

Should show 85-95% coverage in Systems test.

Branch is the most common. In addition to every statement, has every branch been executed in each direction.

Should show 99% in module testing.

Path is generally restricted from full Path coverage. Other examples of coverage might be data-use pairs, etc.

Should show 100% in subroutine testing.

Interface (call-pair) coverage can be used at the system test level (e.g. S-TCAT [Mill86])

Repeatability

- Written procedure
  Level of Detail?

- Automated Script
  Comments?
  Parameterized?

Is the test case repeatable?

1) A written procedure to follow.
   How detailed?
   For expert developer, not very
   For in-expert technician, very precise
2) Automated test script.
   Well commented?
   Parameterized?

Externals vs Internals

- Black Box (at specification time)
- White/Glass Box (knowledge of internals)

- Compatibility across releases

TANDEM COMPUTERS 4/16/80 14

Black box (e.g. Functional) vs White/Glass box (e.g. Structural) (vs Gray box)

Developer testing can be Black Box when done at specification time. Then followed by White/Glass box testing. Similarly for independent test group or QA.

Externals are usually more constant. Externals compatibility.

Internals may be very release specific.

Test Tracking

Documentation
IEEE:

Plan,	Design Spec,
Case Spec,	Procedure Spec,
Log,	Item Transmittal Form,
Incident Report,	Summary Report

Privacy
Public when code goes under CM

Cost

TANDEM COMPUTERS 4/16/90 15

What level of documentation is produced?

IEEE Standard of Software Test Documentation

"Each organization using the standard will need to specify the classes of software to which it applies and the specific documents required for a particular test phase." [IEEE87]

IEEE Standard for Software Unit Testing [IEEE87]

Requires: Design Spec & Summary Report

Beizer's concept of "Public and Private Bugs" [Beiz84]

A possible rule of thumb: all defects are public, i.e. formally tracked, when the code goes under configuration management (CM).

Contrast with Fagan Inspection method - Counts, but not database logs. [Faga76]

Cost of tracking: producing documentation, recording and tracking defects

"the cost of correcting a bug is geometrically related to its degree of public exposure" [Beiz84]

Product Statistics

Disk Space
Memory Size
Speed/Performance

TANDEM COMPUTERS 4/16/90 16

Who determines and records the product statistics?

Development? QA? Test? Performance group?

Which ones?
Why?

Ould & Unwin's Testing in Software Development

4 views and roles

Manager	structure, organization, planning, control
User	Requirements Analysis, System Specification, Acceptance Testing & Plan
Designer	System Design & Specification, Integration and System Testing & Plan
Programmer	Module Design & Specification & Coding, Module Testing & Plan

[Ould86]

An independent testing group participates usually as an internal version of the User Acceptance test, i.e. companies want their independent test groups to find problems, not their user upon acceptance.

Metzger's
Managing Programming People

Manager	
Analyst	
Designer	
Programmer	Unit & Integration test
Tester	Program System test
Support Staff	
Customer	

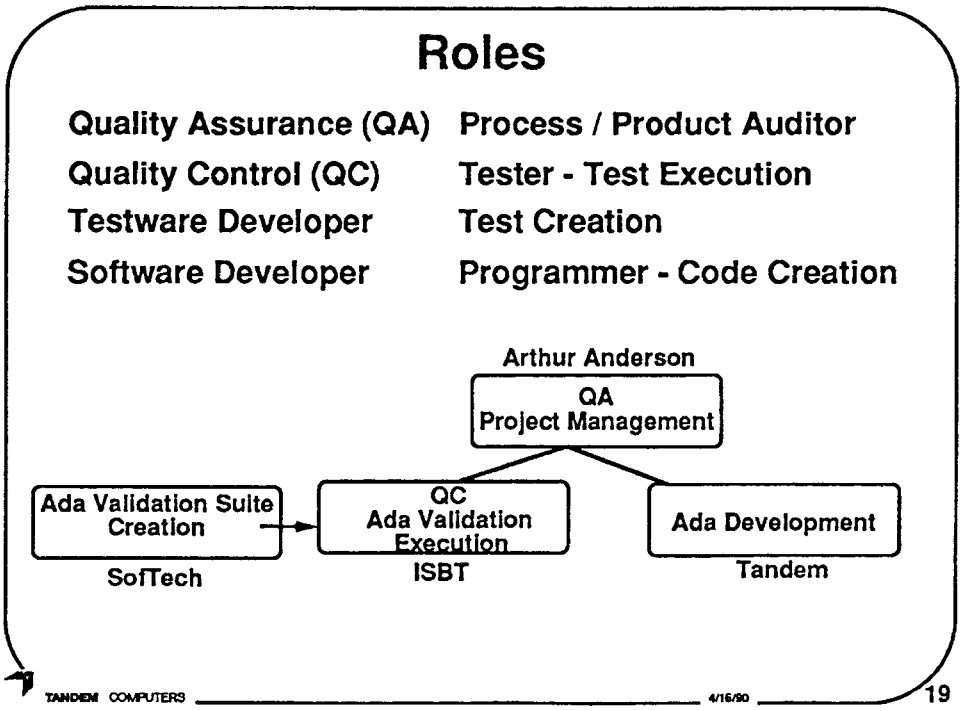
PHASES

Definition	
Design	
Programming	A system of programs is written and tested following the blueprint produced during the Design Phase.
System Test	The program system is ***retested*** by other than the programmers who produced the programs
Acceptance	The program system is tested again, this time as a demonstration for the customer, to show that the system precisely fulfills the customer's requirements agreed upon back in the Definition Phase.
Installation and Operation	

[Metz87]

TANDEM COMPUTERS 4/16/90 18

It is conceivable that the System Test is performed by an independent programming team, i.e. team A tests B's system and team B tests A's system, but an independent test group allows more focused attention and skills.



4 Roles delineated.

Extreme (hypothetical) case is demonstrated where each role is handled by a separate company, the ultimate IV&V.

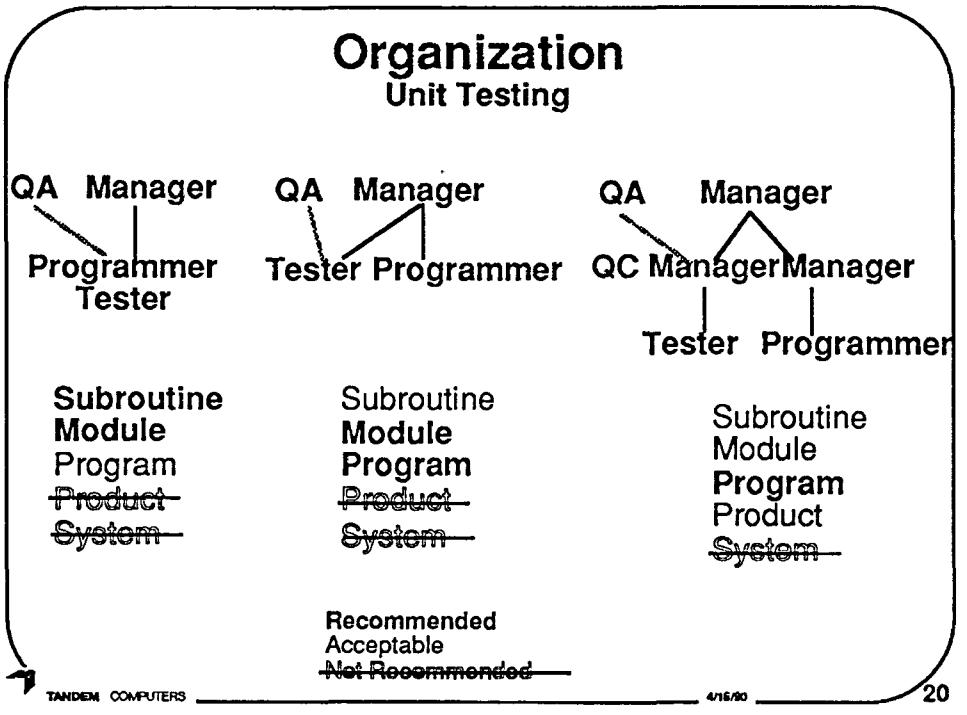
"Quality Assurance

The act of leading, teaching, auditing the process by which the product is produced to provide confidence in the conformance of a specific product to a design or specification.

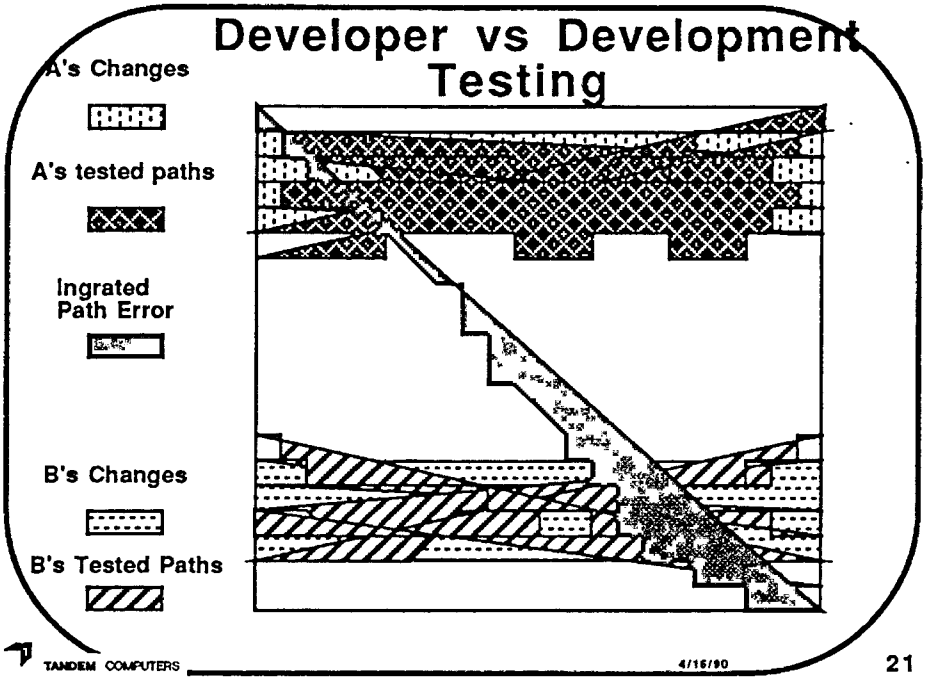
Quality Control

The act of directing, influencing, verifying, and correcting for ensuring the conformance of a specific product to a design or specification."

[Basili - Improving the software process and product in a measurable way]

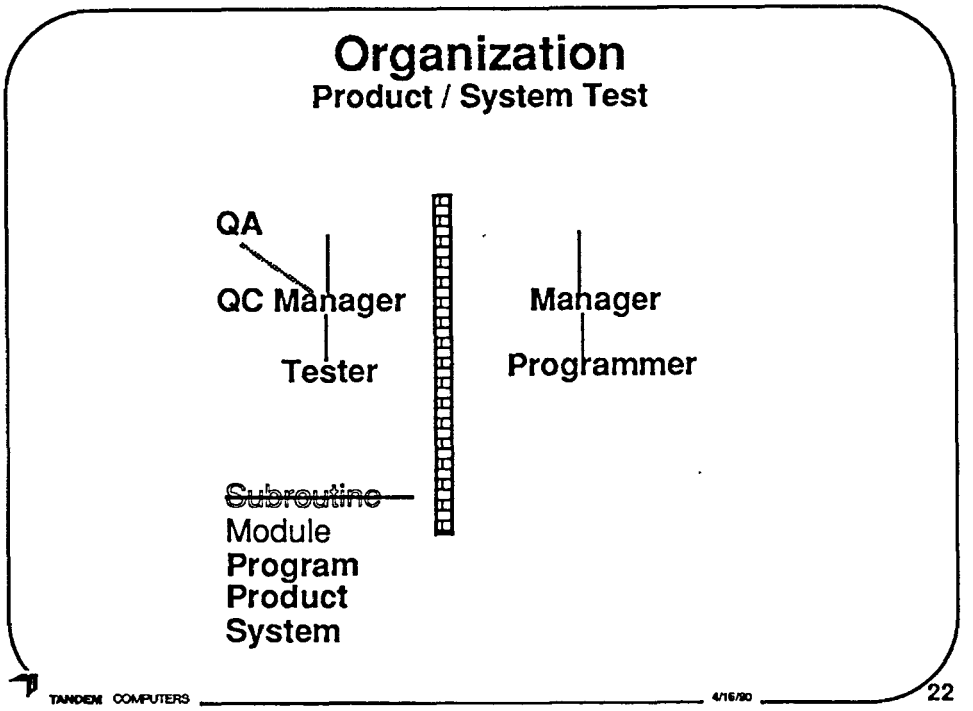


The IEEE Standard for Software Unit Testing "is applicable whether or not the unit tester is also the developer." [IEEE Std]



Who tests the areas not changed by any programmer?

Programmers rarely test all paths that intersect the changes they made.



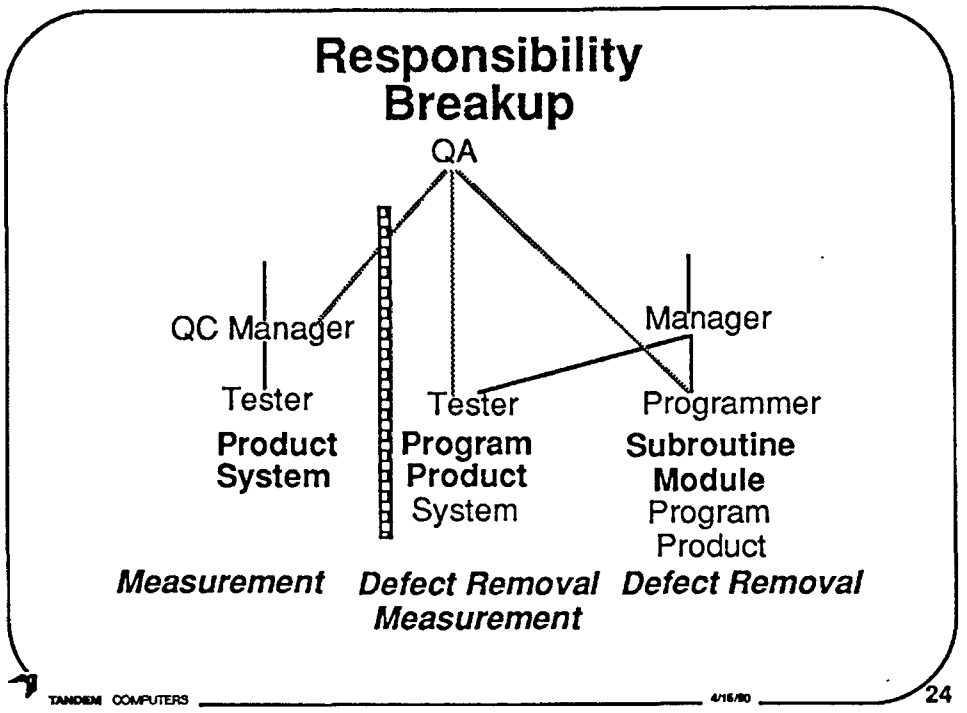
At some point, I believe an independent test organization must exist. Separated to some extent by the legendary 'wall'. This 'wall' gives the test group the independence to be rewarded for their findings regardless of whether it is positive or negative.

If the independent test organization is in the same company (as typical for industrial software), then at some point the testers and programmers report into the same point, e.g. at least the CEO (Chief Executive Officer). At what point they connect is dependent on the size and organization of the company. The higher up, the more independent.

DeMarco

Defect data is collected as a by-product of fault detection by someone other than the person or team responsible for the code.

"Defect data is collected as by-product of fault detection by someone other than the person or team responsible for the code. You can't expect the individual to note and record his own defects and then allow them to be charged against him. Any flaw that is caught and corrected by the person responsible ought not to be thought of as a defect at all." [P.213 Controlling Software Projects [DeMa82]]



This is my recommended model:

Programmers test their own code. Generally this is at the subroutine and module level, but may also include program and product for small products. Defect removal is the primary goal of this testing effort.

A test organization within development offloads developers from testing the program and product. This is not required, but can prove useful. It covers the question of who tests the areas not changed by any programmer. It also allows specialization by the testers. This organization has both measurement and defect removal as its goals.

An independent test organization is essential as already stressed on the previous slides. Measurement should be the primary goal of this testing effort.

Responsibilities

	Documentation	Automation	Coverage	Tracking
QC	TP, TDS TL, IR, Log	Runnable By technician	85% stmt 100% Interface	Formal IR
Tester	TP, Log	Runnable		Formal IR
Programmer	TDS, Summary	Repeatable (by expert)	100% branch	

TP	Test Plan
TDS	Test Design Specification
TP	Test Procedure Specification
TL	Test Library
IR	Incident Report
Log	Test Log
Summary	Test Summary Report

TANDEM COMPUTERS 4/16/90 25

Independent Test, QA, should follow a thorough documented test process. Use statement coverage as a (not the) measure of test library completeness. Can also or alternatively use Interface (call-pair) coverage.

Development's testers goals are not necessarily to get any type of complete coverage, but to complement the Programmers tests by provide testing of overlapped areas that are no single programmers responsibility.

Programmers should adhere to IEEE Unit Test minimums.

Bibliography

[Beiz84] B. Beizer, *Software System Testing and Quality Assurance*, New York: Van Nostrand Reinhold, 1984

[Date74] C.J.Date, M.A. McMorran, and G.C.H. Sharman, "Program Proving and Formal Development: A tutorial introduction," TR-12.127, IBM U.K. Laboratories, Hursley, England, 1974

[DeMa82] T. DeMarco, *Controlling Software Projects*, Englewood Cliffs N.J.: Prentice-Hall, 1982

[Faga76] M.E.Fagan,, "Design and code inspections to reduce errors in program development", *IBM System Journal,* V15#3, 1976

[Faga85] M.E.Fagan, "Advances in Software Inspections", *IEEE Transactions on Software Engineering*, V12#7, July 1986.

[Haml89] R. Hamlet, "Theoretical Comparison of Testing Methods", *Proceedings of ACM SIGSOFT '89: Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, Dec 1989

[Hoff89] D. Hoffman, and C. Brealey, 'Module Test Case Generation', *Proceedings of ACM SIGSOFT '89: Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, Dec 1989

[IEEE87] IEEE, *Software Engineering Standards*, New York: IEEE (Wiley-Interscience), 1987

[Kern81] B. Kernigan, and P.J. Plauger, *Software Tools in Pascal*, Reading MA: Addison-Wesley, 1981

[Metz87] P. W. Metzger, *Managing Programming People - A Personal View*, Englewood Cliffs N.J.: Prentice-Hall, 1987

[Musa87] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, New York: McGraw-Hill, 1987

[Ould86] M.A. Ould, and C. Unwin, *Testing in Software Development,* Cambridge: Cambridge University Press, 1986

[Selb87] R.W. Selby, V.R. Basili, and F.T. Baker, "Cleanroom Software Development: An Empirical Evaluation", *IEEE Transactions on Software Engineering*, V13#9, Sept. 1987

[Mill86] E. Miller, *S-TCAT,* Software Research, Inc., UM-1189/2, 1986

Paper 4-T-3

DEVELOPER TESTING VS. QA TESTING: WHO DOES WHAT AND WHY?

Mr. Keith Stobie
Senior Technical Staff Member
Tandem Computers

Mr. Keith Stobie is a Senior Technical Staff member within Software Development QA at Tandem Computers. He received his BSCS from Cornell University in 1979, and spent the next several years as a Quality Assurance Developer at Tandem, where he positively impacted product quality in a variety of Online Transaction Processing products, including Transaction Monitoring Facility (TMF), TRANSFER, and Distributed Systems Management (DSM) products. Keith was a founding QA developer of the Systems QA group. Keith is a leader in testing methodology and tools technology at Tandem resulting in high levels of testing automation today. Keith is a direction setter in the areas of training including formal classes, a lecture series and informal discussion sessions. Keith brough formal Inspections to Tandem and has worked towards their implementation and acceptance.

QW-90 - Quality Week 1990

Developer testing VS QA testing
Who does what and why

Many books and articles describe types of testing: unit, system, structural, functional, white-box, black-box, internal, external, statistical, anecdotal, etc. But few address who does what kind and why. This talk will address similarities and differences between the types of testing by various organizations.

It discusses the kind of testing an in-house independent QA organization or a third party independent V&V organization might do. Also the kind of testing individual developers should do of their own code and of their colleagues and team members code. The purpose of the types of testing the various groups do will be delineated.

Keith Stobie
TANDEM COMPUTERS
10555 Ridgeview Court, LOC 100-11
Cupertino, CA 95014

Keith Stobie is a Senior Technical Staff member within Software Development QA at Tandem Computers. He received his BSCS from Cornell University in 1979, and spent the next several years as a Quality Assurance Developer at Tandem, where he positively impacted product quality in a variety of Online Transaction Processing products, including Transaction Monitoring Facility (TMF), TRANSFER, and Distributed Systems Management (DSM) products. Keith was a founding QA developer of the Systems QA group.

Keith is a leader in testing methodology and tools technology at Tandem resulting in high levels of testing automation today. Keith is a direction setter in the areas of training including formal classes, a lecture series and informal discussion sessions. Keith brought formal Inspections to Tandem and has worked towards their implementation and acceptance.

Developer testing VS QA testing

Who does what and why

TANDEM COMPUTERS
19333 Vallco Parkway
Cupertino, CA 95014

Keith Stobie

Developer testing
VS
Independent testing

Who tests?
Why test twice?

Many books and articles describe types of testing: unit, system, structural, functional, white-box, black-box, internal, external, statistical, anecdotal, etc. But few address who does what kind and why. This talk will address similarities and differences between the types of testing by various organizations.

It discusses the kind of testing an in-house independent QA organization or a third party independent V&V organization might do. Also the kind of testing individual developers should do of their own code and of their colleagues and team members code. The purpose of the types of testing the various groups do will be delineated.

Testing?

Can testing be avoided?
- Proof of correctness
- Cleanroom

Independent Testing?
- Government (DOD)
- Industry

Independent QA
- Government (DOD)
- Industry

TANDEM COMPUTERS 4/16/90 2

This talk assumes a large amount of 'traditional' code testing will be performed. Techniques like Proof of correctness [Date74] and Cleanroom will not be further addressed. Cleanroom allows no programmer testing and only randomly generated tests applied by an independent group [Selb87]

This talk deals with the Testing tasks within a company. DOD may be able to afford independent verification & validation, IV&V, but it is very rare in industry.

Similarly it is assumed that QA or auditing, is a company function, not an independent function.

Characteristics

Product Develop<u>er</u> <-> Product Develop<u>ment</u> <-> Product QA

<u>Design Testing</u>

- Externals vs Internals

<u>Code Testing</u>

- Level of Testing
  - Subroutine, Module, Program, Product, Integration, System
- Structural Coverage
  - Path, Branch, Statement
- Repeatability
- Externals vs Internals
- Test Tracking
- Product Statistics

<u>Documentation Testing</u>

- Externals vs Internals

TANDEM COMPUTERS 4/16/90 3

At least two different possible goals for testing:

1) **Defect Removal**
2) **Measurement**
   - i) **Defect Density** [Haml89]
   - ii) **Reliability (MTBF)** [Musa87]

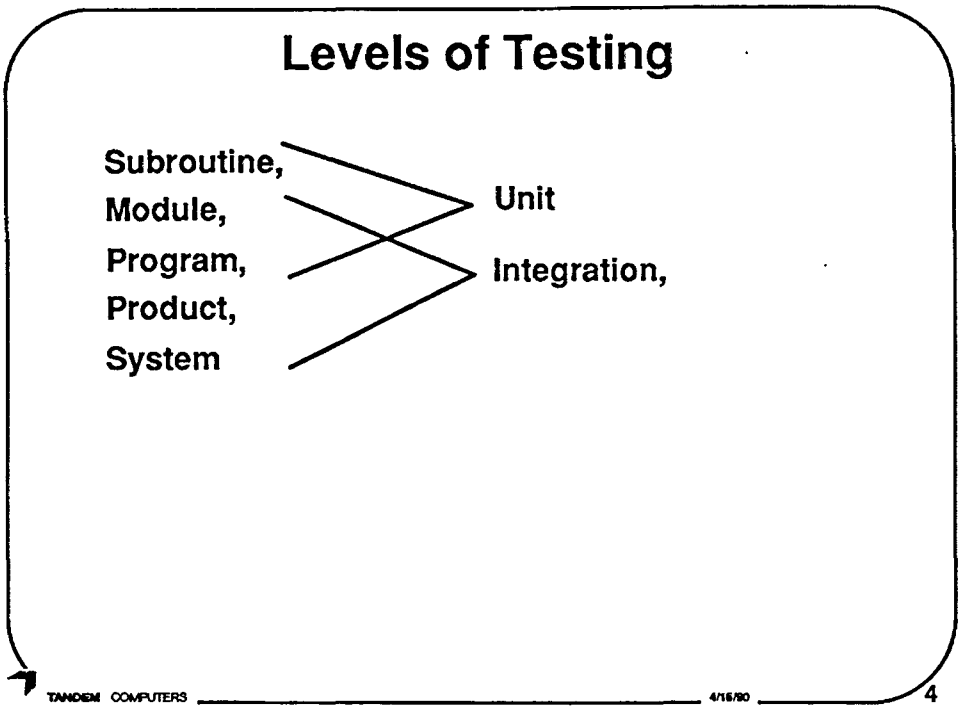
<u>Design Testing</u>

<u>Code Testing</u>

- Level of Testing
  - Developer (Subr, Mod, Prog)
  - Development (Prod, Int), QA (Prod, Int)
  - Gremlin/Alpha/Beta (System)
- 100% Branch Developer <-> 80% Statement QA
- Repeatability is everyone's goal
- QA External view <-> Developer/ment Internal view
- Test Documentation & Defect Tracking
- Stats: Memory, Disc, performance

<u>Documentation Testing</u>

Will not discuss Design & Doc testing unless time permits



"Software unit testing is a process that includes the performance of test planning, and acquisition of a test set, and the measurement of a test unit against its requirements."

TEST UNIT

"A set of one or more computer program modules together with associated control data, (for example, tables), usage procedures and operating procedures that satisfy the following conditions:

1) All modules are from a single computer program
2) At least one of the new or changed modules in the set has not completed unit test
3) The set of modules together with its associated data and procedures are the sole object of a testing process."

"A test unit may occur at any level of the design hierarchy from a single module to a complete program. Therefore, a test unit may be a module, a few modules, or a complete computer program along with associated data and procedures."

"A test unit may contain one or more modules that have already been unit tested."

[IEEE Standard for Software Unit Testing, IEEE87]

"A unit is the work of one programmer." [Biez84]

Subroutine

```
{ index -- find position of a character c in
  string s }
function index( var s: string; c: character ) :
  integer;
var
    i : integer;
begin
    i := 1;
    index := 0;
    while (s[i] <> c) and (s[i] <> ENDSTR) do
        i := i+1;
    if (s[i] <> ENDSTR) then
        index := i
end;
```

Adapted from *Software Tools in Pascal*, P. 48

Subroutines are the smallest parts of a language to test. Other names: Procedures, Functions

Usually tested by a 'driver' that calls the procedure with various parameters.

This example is used in the Coverage Graphs.

Module

Symbol Table

```
table := Create_Table( size );
Insert( Symbol, value )
value := Retrieve( Symbol )
Delete( Symbol )
Destroy( Table )
```

TANDEM COMPUTERS 4/16/90 6

Modules are a basic concept now. Some languages, Ada, Modula, OOPs, e.g. C++, SmallTalk, have syntactic support.

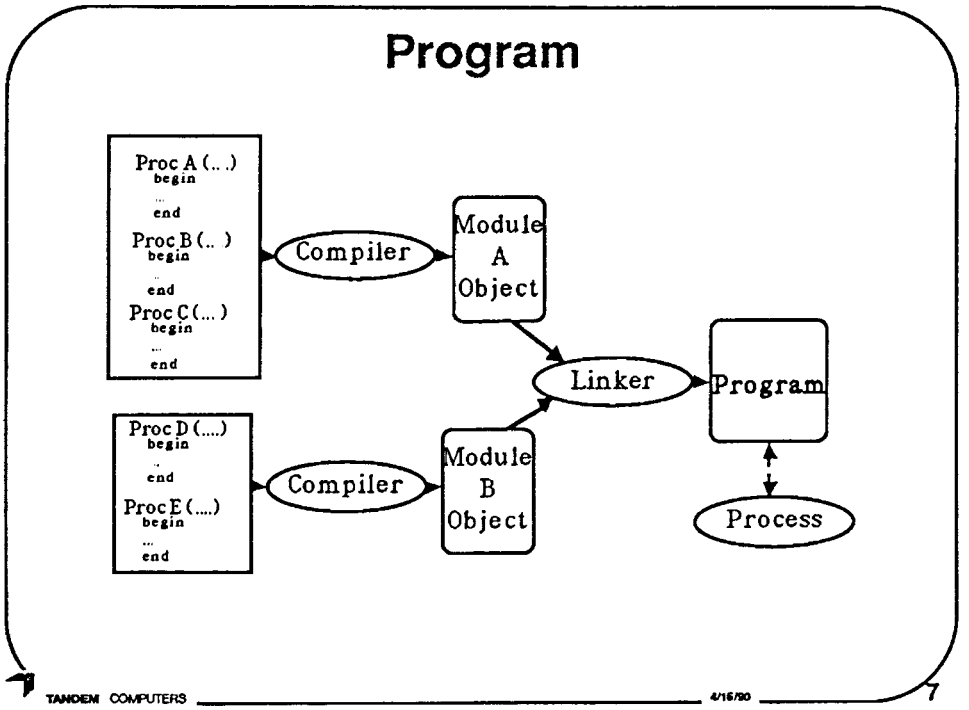
Frequently a Module is separately <u>compilable.</u>

A module is a 'programming work assignment' [Hoff89]. Most common lowest level of testing by programmers.

"(1) A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading; for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine. (ANSI)

(2) A logically separable part of a program."

[IEEE Standard Glossary of Software Engineering Terminology IEEE87]

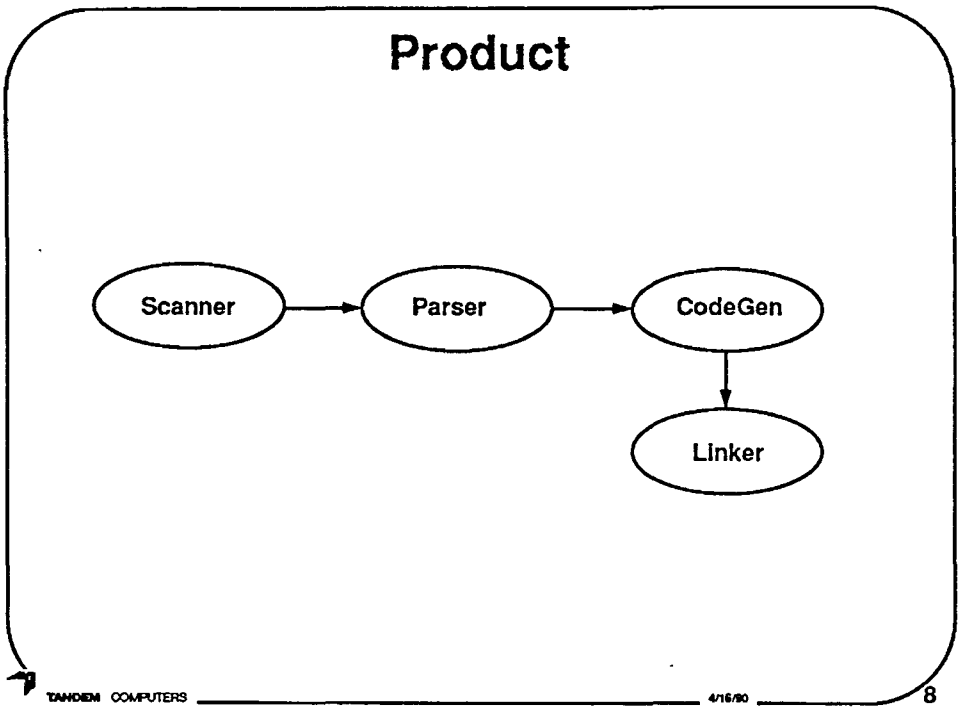


A Computer Program:

A sequence of instructions suitable for processing by a computer. Processing may include the use of an assembler, a compiler, an interpreter, or a translator to prepare the program for execution as well as to execute it. See also (ISO) program.

[IEEE Standard Glossary of Software Engineering Terminology IEEE87]

A Program is instantiated as a Process when it is run.

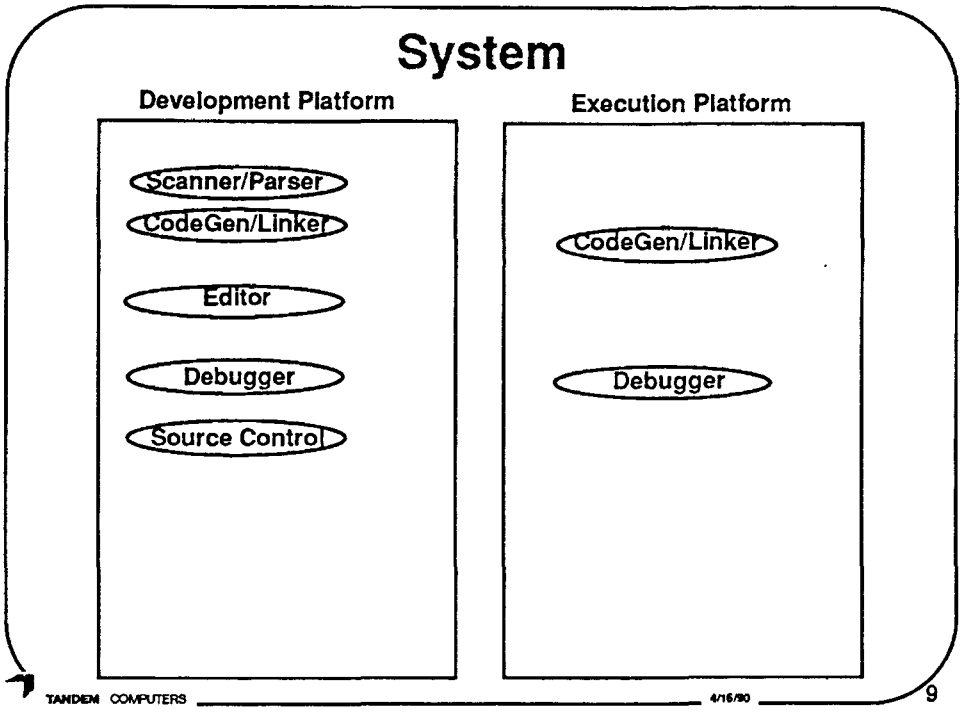


A product is usually something that is sold separately by a company.

In general terms, this also includes the documentation, marketing sheets, etc.

There may not be any distinction between program and product, if the product is only one program.

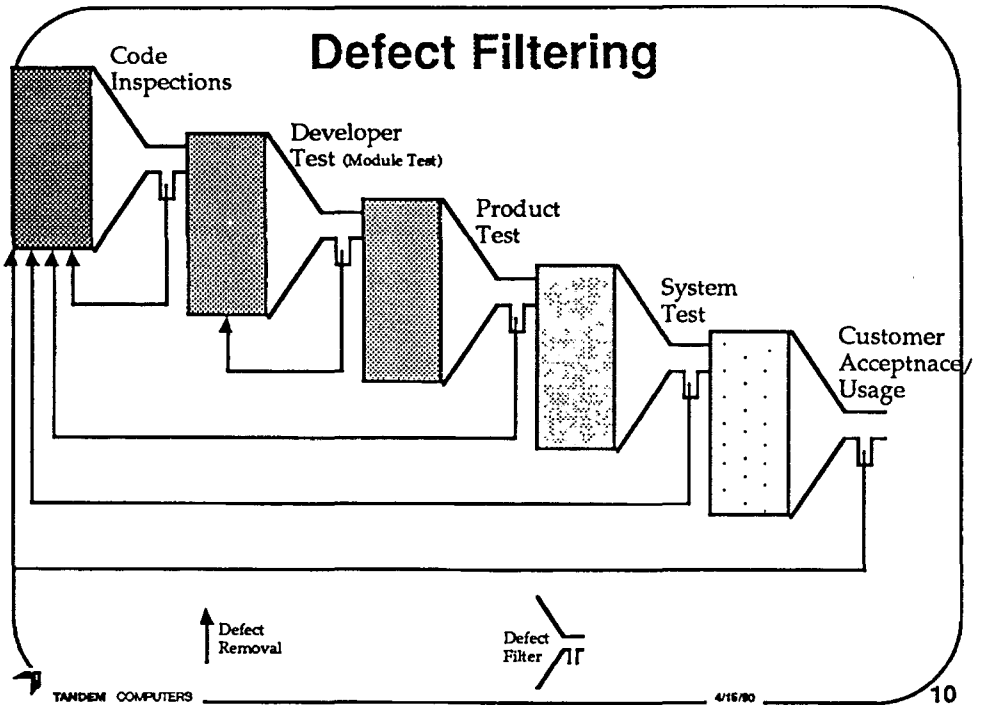
"A **subsystem** is a collection of programs that as a collection accomplish some specified processing." [Beiz84]



Interaction among products

Interaction among different nodes, platforms, etc.

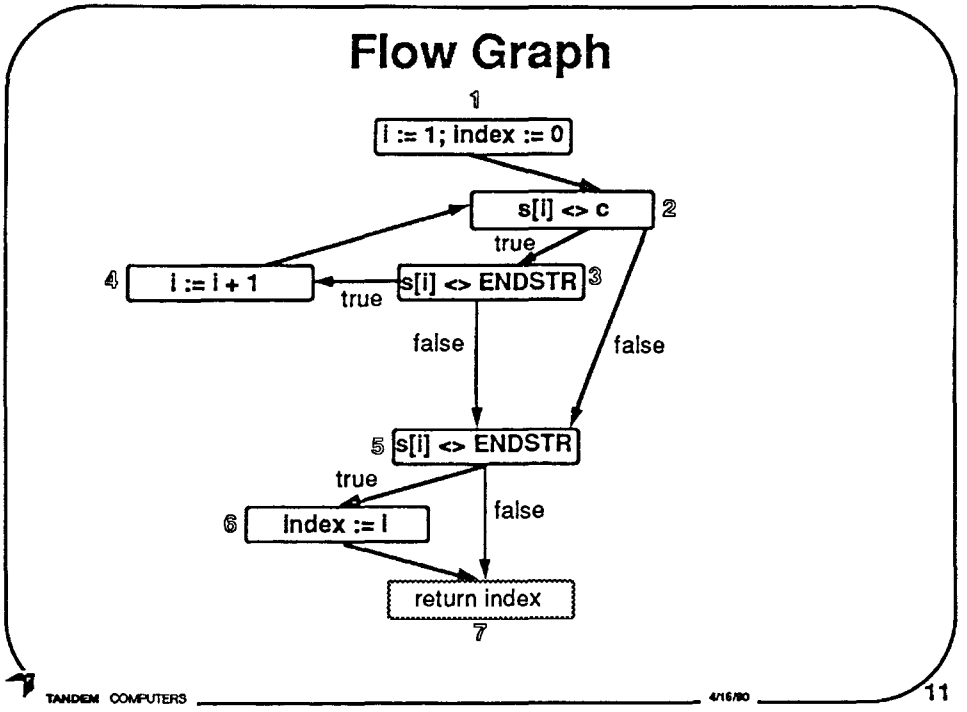
May use customer applications



Picture shows code having fewer and fewer defects as defect removal filters are applied. Number of defects is represented by number of dots (shading)

The amount and type of testing is a cost / risk tradeoff.

You should first select the most effective techniques (e.g. Inspections >60% of defects caught). You should add or tailor other techniques to complement the others for the most cost effective process. For example, Inspections don't find many timing and interface defects. Perhaps Product Test should be geared for these.



100% Statements:

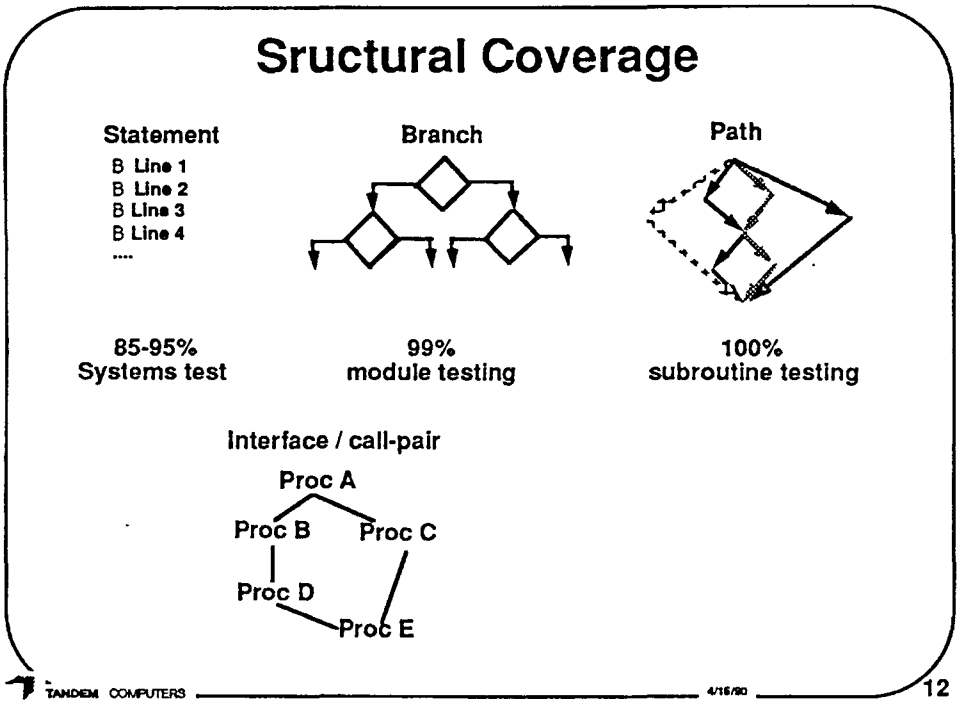
1) s = 'ab' , c= 'b' 1 2 3 4 5 6

100% Branches

1) 2 true, 3 true, 2 false, 5 true
2) s = '', c = 'b' 1 2 3 5
   2 true, 3 false, 5 false

100% Paths

1) 1 ⇨ 2 ⇨ 3 ⇨ 4 ⇨ 2 ⇨ 5 ⇨ 6 ⇨ 7
2) 1 ⇨ 2 ⇨ 3 ⇨ 5 ⇨ 7
3) s = 'a', c='a'
   1 ⇨ 2 ⇨ 5 ⇨ 6 ⇨ 7



Statement is the most basic. Has every statement in the program been executed at least once.

Should show 85-95% coverage in Systems test.

Branch is the most common. In addition to every statement, has every branch been executed in each direction.

Should show 99% in module testing.

Path is generally restricted from full Path coverage. Other examples of coverage might be data-use pairs, etc.

Should show 100% in subroutine testing.

Interface (call-pair) coverage can be used at the system test level (e.g. S-TCAT [Mill86])

Repeatability

- Written procedure
  Level of Detail?
- Automated Script
  Comments?
  Parameterized?

TANDEM COMPUTERS 4/16/90 13

Is the test case repeatable?

1) A written procedure to follow.
   How detailed?
   For expert developer, not very
   For in-expert technician, very precise
2) Automated test script.
   Well commented?
   Parameterized?

Externals vs Internals

- Black Box (at specification time)
- White/Glass Box (knowledge of internals)

- Compatibility across releases

TANDEM COMPUTERS 4/15/90 14

Black box (e.g. Functional) vs White/Glass box (e.g. Structural) (vs Gray box)

Developer testing can be Black Box when done at specification time. Then followed by White/Glass box testing. Similarly for independent test group or QA.

Externals are usually more constant. Externals compatibility.

Internals may be very release specific.

Test Tracking

Documentation
IEEE:

Plan,	Design Spec,
Case Spec,	Procedure Spec,
Log,	Item Transmittal Form,
Incident Report,	Summary Report

Privacy
Public when code goes under CM

Cost

TANDEM COMPUTERS 4/16/90 15

What level of documentation is produced?

IEEE Standard of Software Test Documentation

"Each organization using the standard will need to specify the classes of software to which it applies and the specific documents required for a particular test phase." [IEEE87]

IEEE Standard for Software Unit Testing [IEEE87]

Requires: Design Spec & Summary Report

Beizer's concept of "Public and Private Bugs" [Beiz84]

A possible rule of thumb: all defects are public, i.e. formally tracked, when the code goes under configuration management (CM).

Contrast with Fagan Inspection method - Counts, but not database logs. [Faga76]

Cost of tracking: producing documentation, recording and tracking defects

"the cost of correcting a bug is geometrically related to its degree of public exposure" [Beiz84]

Product Statistics

Disk Space
Memory Size
Speed/Performance

TANDEM COMPUTERS 4/15/90 16

Who determines and records the product statistics?

Development? QA? Test? Performance group?

Which ones?
Why?

Ould & Unwin's Testing in Software Development

4 views and roles

Manager	structure, organization, planning, control
User	Requirements Analysis, System Specification, Acceptance Testing & Plan
Designer	System Design & Specification, Integration and System Testing & Plan
Programmer	Module Design & Specification & Coding, Module Testing & Plan

[Ould86]

An independent testing group participates usually as an internal version of the User Acceptance test, i.e. companies want their independent test groups to find problems, not their user upon acceptance.

Metzger's
Managing Programming People

Manager	
Analyst	
Designer	
Programmer	Unit & Integration test
Tester	Program System test
Support Staff	
Customer	

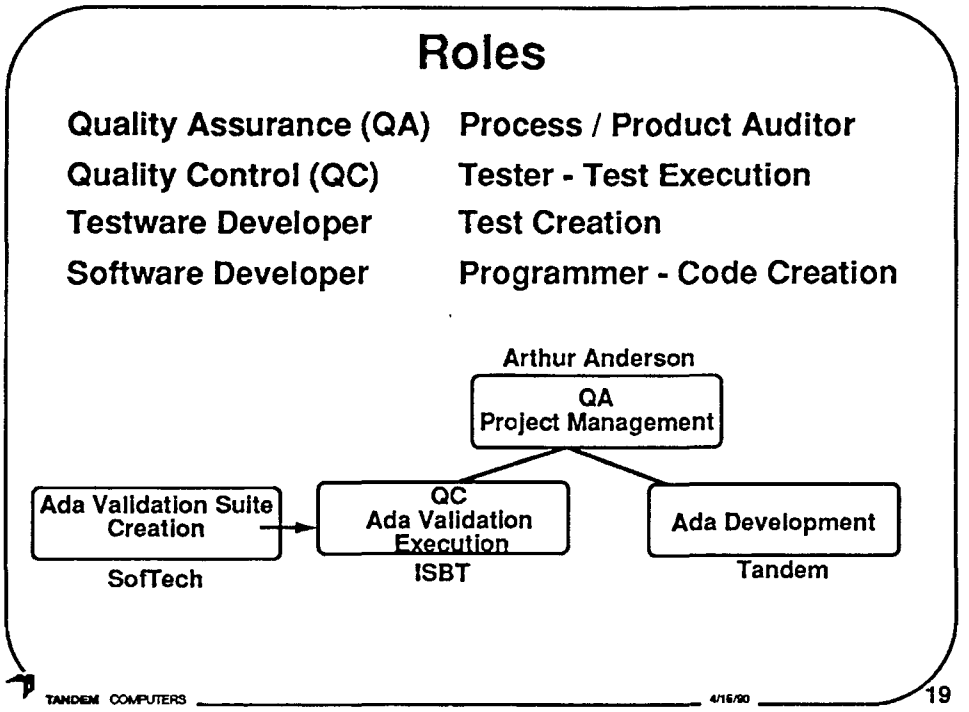
PHASES

Definition	
Design	
Programming	A system of programs is written and tested following the blueprint produced during the Design Phase.
System Test	The program system is ***retested*** by other than the programmers who produced the programs
Acceptance	The program system is tested again, this time as a demonstration for the customer, to show that the system precisely fulfills the customer's requirements agreed upon back in the Definition Phase.
Installation and Operation	

[Metz87]

TANDEM COMPUTERS 4/16/90 18

It is conceivable that the System Test is performed by an independent programming team, i.e. team A tests B's system and team B tests A's system, but an independent test group allows more focused attention and skills.



4 Roles delineated.

Extreme (hypothetical) case is demonstrated where each role is handled by a separate company, the ultimate IV&V.

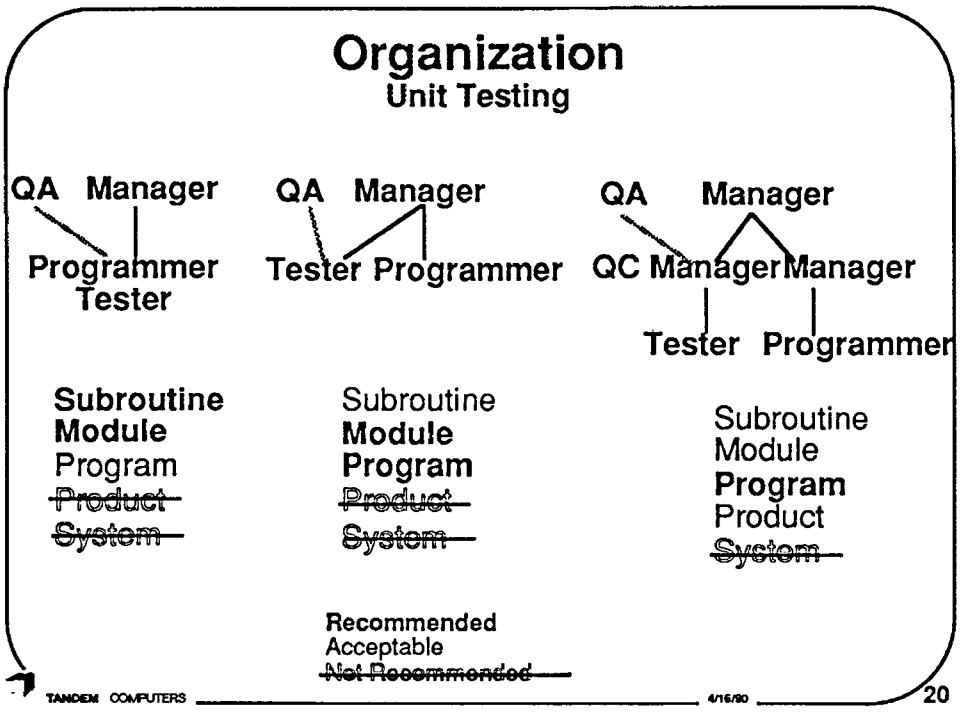
"Quality Assurance

The act of leading, teaching, auditing the process by which the product is produced to provide confidence in the conformance of a specific product to a design or specification.

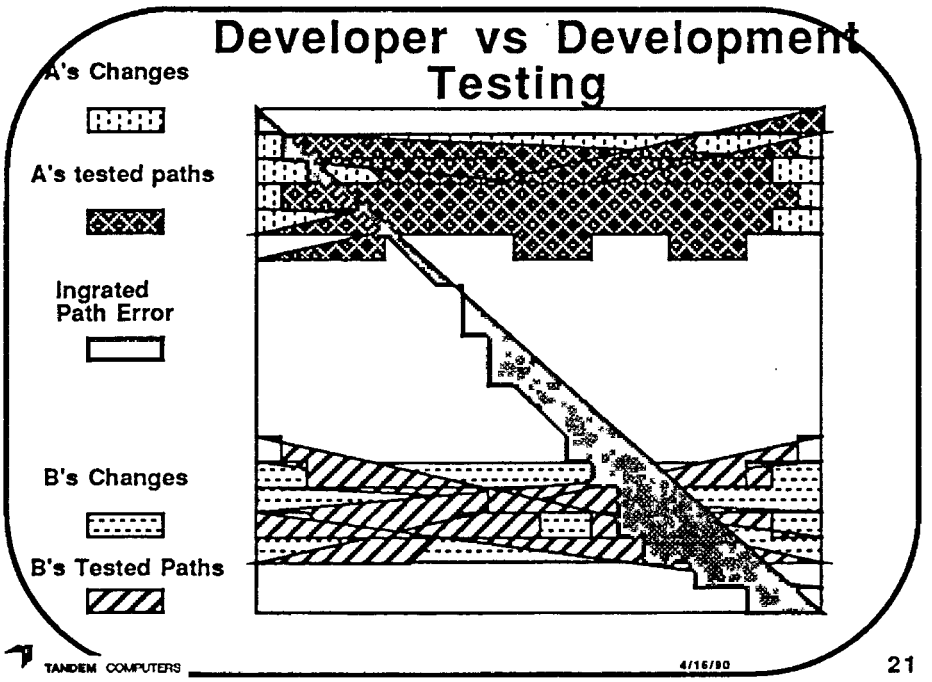
Quality Control

The act of directing, influencing, verifying, and correcting for ensuring the conformance of a specific product to a design or specification."

[Basili - Improving the software process and product in a measurable way]

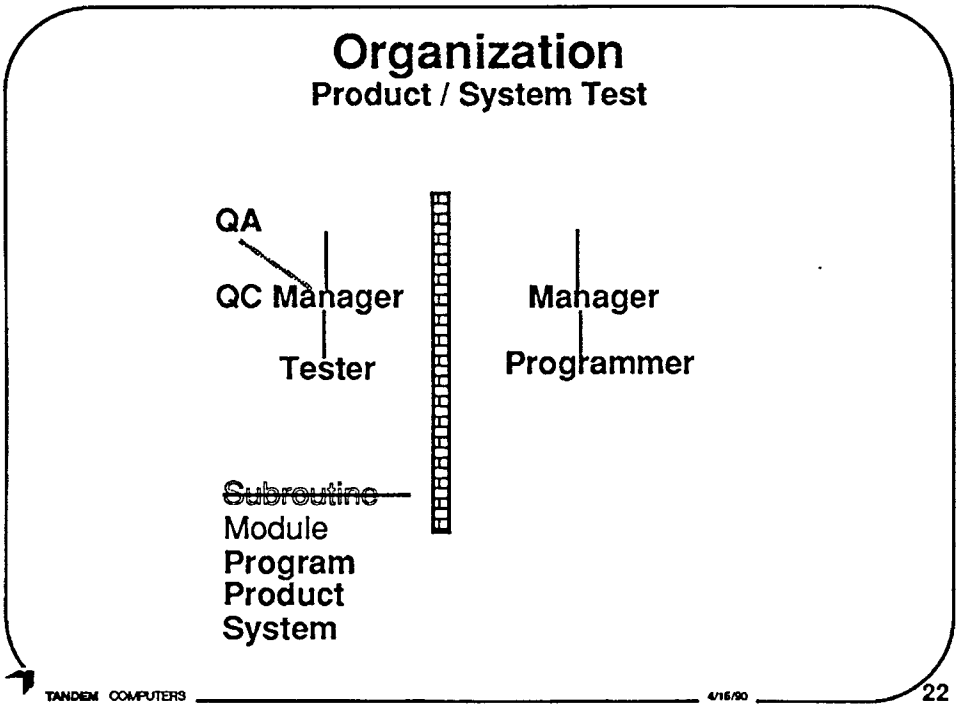


The IEEE Standard for Software Unit Testing "is applicable whether or not the unit tester is also the developer." [IEEE Std]



Who tests the areas not changed by any programmer?

Programmers rarely test all paths that intersect the changes they made.



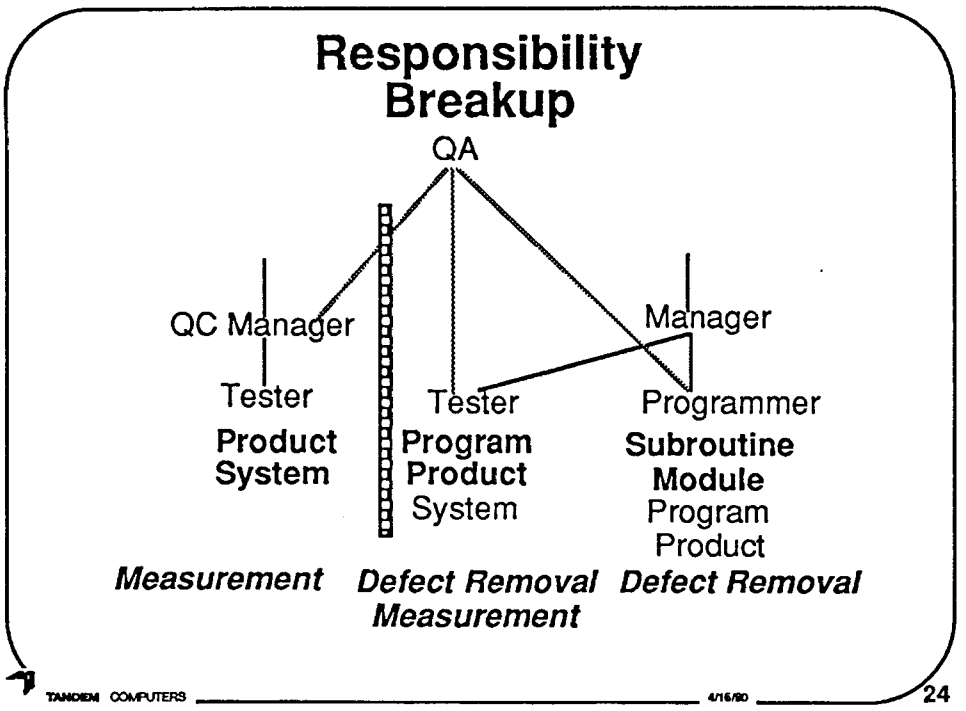
At some point, I believe an independent test organization must exist. Separated to some extent by the legendary 'wall'. This 'wall' gives the test group the independence to be rewarded for their findings regardless of whether it is positive or negative.

If the independent test organization is in the same company (as typical for industrial software), then at some point the testers and programmers report into the same point, e.g. at least the CEO (Chief Executive Officer). At what point they connect is dependent on the size and organization of the company. The higher up, the more independent.

DeMarco

Defect data is collected as a by-product of fault detection by someone other than the person or team responsible for the code.

"Defect data is collected as by-product of fault detection by someone other than the person or team responsible for the code. You can't expect the individual to note and record his own defects and then allow them to be charged against him. Any flaw that is caught and corrected by the person responsible ought not to be thought of as a defect at all." [P.213 Controlling Software Projects [DeMa82]]



This is my recommended model:

Programmers test their own code. Generally this is at the subroutine and module level, but may also include program and product for small products. Defect removal is the primary goal of this testing effort.

A test organization within development offloads developers from testing the program and product. This is not required, but can prove useful. It covers the question of who tests the areas not changed by any programmer. It also allows specialization by the testers. This organization has both measurement and defect removal as its goals.

An independent test organization is essential as already stressed on the previous slides. Measurement should be the primary goal of this testing effort.

Responsibilities

	Documentation	Automation	Coverage	Tracking
QC	TP, TDS TL, IR, Log	Runnable By technician	85% stmt 100% Interface	Formal IR
Tester	TP, Log	Runnable		Formal IR
Programmer	TDS, Summary	Repeatable (by expert)	100% branch	

TP	Test Plan
TDS	Test Design Specification
TP	Test Procedure Specification
TL	Test Library
IR	Incident Report
Log	Test Log
Summary	Test Summary Report

TANDEM COMPUTERS 4/16/90 25

Independent Test, QA, should follow a thorough documented test process. Use statement coverage as a (not the) measure of test library completeness. Can also or alternatively use Interface (call-pair) coverage.

Development's testers goals are not necessarily to get any type of complete coverage, but to complement the Programmers tests by provide testing of overlapped areas that are no single programmers responsibility.

Programmers should adhere to IEEE Unit Test minimums.

Bibliography

[Beiz84] B. Beizer, *Software System Testing and Quality Assurance*, New York: Van Nostrand Reinhold, 1984

[Date74] C.J.Date, M.A. McMorran, and G.C.H. Sharman, "Program Proving and Formal Development: A tutorial introduction," TR-12.127, IBM U.K. Laboratories, Hursley, England, 1974

[DeMa82] T. DeMarco, *Controlling Software Projects*, Englewood Cliffs N.J.: Prentice-Hall, 1982

[Faga76] M.E.Fagan,, "Design and code inspections to reduce errors in program development", *IBM System Journal,* V15#3, 1976

[Faga85] M.E.Fagan, "Advances in Software Inspections", *IEEE Transactions on Software Engineering*, V12#7, July 1986.

[Haml89] R. Hamlet, "Theoretical Comparison of Testing Methods", *Proceedings of ACM SIGSOFT '89: Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, Dec 1989

[Hoff89] D. Hoffman, and C. Brealey, 'Module Test Case Generation', *Proceedings of ACM SIGSOFT '89: Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, Dec 1989

[IEEE87] IEEE, *Software Engineering Standards*, New York: IEEE (Wiley-Interscience), 1987

[Kern81] B. Kernigan, and P.J. Plauger, *Software Tools in Pascal*, Reading MA: Addison-Wesley, 1981

[Metz87] P. W. Metzger, *Managing Programming People - A Personal View*, Englewood Cliffs N.J.: Prentice-Hall, 1987

[Musa87] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, New York: McGraw-Hill, 1987

[Ould86] M.A. Ould, and C. Unwin, *Testing in Software Development,* Cambridge: Cambridge University Press, 1986

[Sèlb87] R.W. Selby, V.R. Basili, and F.T. Baker, "Cleanroom Software Development: An Empirical Evaluation", *IEEE Transactions on Software Engineering*, V13#9, Sept. 1987

[Mill86] E. Miller,*S-TCAT,* Software Research, Inc., UM-1189/2, 1986

Paper 4-T-4

APPROACHES TO SPECIFICATION BASED TESTING

Prof. Debra Richardson
Department of Computer Science
UC/ Irvine

Dr. Debra J. Richardson is currently an Assistant Professor in the Department of Information and Computer Science at the University of California, Irvine. She received teh B.A. degree in Mathematics from Revelle College of the University of California at San Diego in 1976 and the M.S. and Ph.D. degrees in Computer and Information Science from the University of Massachusetts at Amherst in 1978 and 1981, respectively. Dr. Richardson's research interests include software testing and analysis, software development environments, and specification languages. The principal thrusts of her current work are developing testing techniques that utilize specifications and mechanisms for integrating diverse testing and analysis techniques.

Approaches to Specification-Based Testing *

Debra J. Richardson

Information and Computer Science
University of California at Irvine

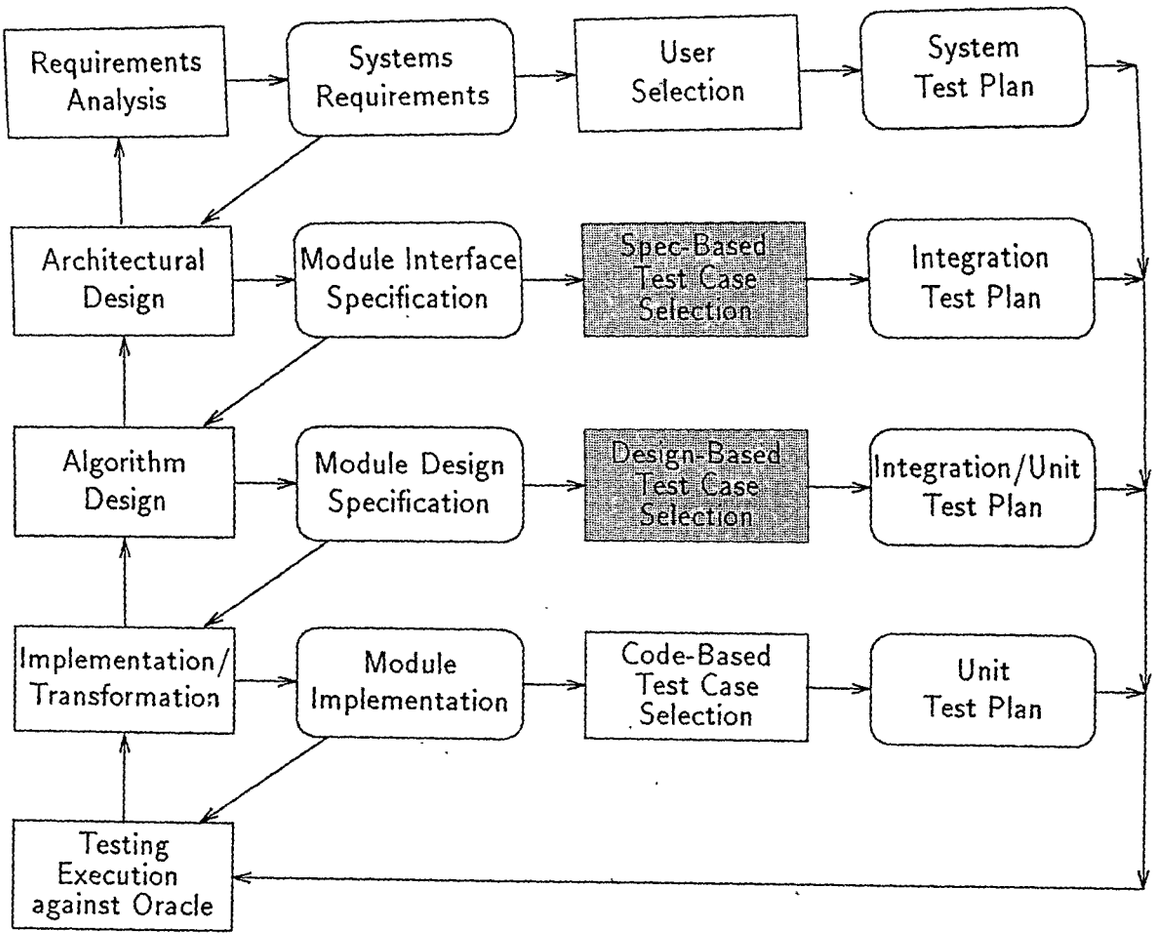
* Presented at TAV3: 3rd Symposium on Testing, Analysis, and Verification sponsored by ACM SIGSoft, Florida, Dec 1989
appears in ACM SIGSoft Software Engineering Notes, December 1989.

This work was supported in part by the National Science Foundation, with cooperation from the Defense Advanced Research Projects Agency, and by Hughes Aircraft Company and the University of California under UC MICRO program.

Specification-Based Testing

- Most testing focuses on actual behavior
- Specification-based testing considers intended behavior
  - augment implementation-based techniques
  - explicit interaction with implementation-based techniques
- Our approach utilizes specifications in selecting test cases
  - formalize traditional functional techniques
  - extend implementation-based techniques

Specification-Based Testing in the Software Lifecycle



Our Approaches

- Error and fault detection criteria
- Active use of specification as oracle
- Test case selection
  - Input criterion
    - condition describing test data (actual data, input domain, output range)
  - Acceptance criterion
    - condition describing correct execution **or** failure detection (expected output, output assertion, false, *ok?*)
- Specifications viewed as pre/post condition pairs

Test Case Templates

structural coverage test case	
input:	path condition
accept:	*ok?*(actual output)

pre/post-condition coverage test case	
input:	pre-condition
accept:	post-condition (data)

pre/post-condition violation test case	
input:	pre-condition $\wedge$ $\neg$post-condition
accept:	*false*

intermediate assertion test case	
input:	$\text{assertion}_{\text{pre}}$
accept:	$\text{assertion}_{\text{post}}$ (data)

Error-Based Testing

- Techniques geared toward detecting specific types of errors
  - i.e., incorrect values produced by execution
  - select test cases to detect potential errors in symbolic representation
- Implementation-based
  - symbolic path representation
  - Domain Testing (White and Cohen)
  - Computation Testing (Howden, Rowland, Richardson and Clark)
- Specification-based
  - symbolic pre/post condition representation
  - Specification/Error-based
  - Oracle/Error-based

Specification/Error-Based

- Apply error-based techniques to specification
  - detect implementation errors that derive from misunderstanding specifications
  - detect specification errors
- Domain (boundary value) testing

spec/domain test case for pre-condition	
input:	*boundary*(pre)
accept:	$(\forall i\ \text{pre}_i\ (\text{data}) \Rightarrow \text{post}_i\ (\text{data})\ ) \wedge ok?$

spec/domain test case for post-condition	
input:	pre $\wedge$ *boundary*(post)
accept:	post (data) $\wedge$ *ok?*

- Computation (special values) testing

spec/computation test case for pre-condition	
input:	*special*(pre)
accept:	$(\forall i\ \text{pre}_i\ (\text{data}) \Rightarrow \text{post}_i\ (\text{data})\ ) \wedge ok?$

spec/computation test case for post-condition	
input:	pre $\wedge$ *special*(post)
accept:	post (data) $\wedge$ *ok?*

Oracle/Error-Based

- Apply error-based techniques to implementation
  - detect implementation errors by violating the specification as an oracle
- Domain (boundary values) testing

oracle/domain test case for PC	
input:	*boundary*(PC) $\wedge$ pre $\wedge \neg$post
accept:	*false*

- Computation (special values) testing

oracle/computation test case for PV	
input:	*special*(PV) $\wedge$ pre $\wedge$ $\neg$post
accept:	*false*

- Incompleteness testing

oracle/incompleteness test case	
input:	$(\forall i \neg \text{pre}_i)$
accept:	*ok?*

Fault-Based Testing

- Techniques geared toward detecting specific types of faults
  - i.e., mistakes in source code
  - select test cases to detect potential faults in source
- Implementation-based
  - distinguish program from source code variants
  - Mutation testing (DeMillo, et. al.)
  - RELAY (Richardson and Thompson)
- Specification-based
  - distinguish specification from pre/post-condition variants
  - specification/fault-based
  - oracle/fault-based

Specification/Fault-Based

- Apply fault-based techniques to specification
  - detect implementation faults that derive from misunderstanding specifications
  - detect specification faults

spec/fault-based test case for pre-condition	
input:	*revealing*(*variant*(pre) $\neq$ pre)
accept:	$(\forall i$ pre_i (data) $\Rightarrow$ post_i (data)) $\wedge$ *ok?*

spec/fault-based test case for post-condition	
input:	pre $\wedge$ *revealing*(*variant*(post) $\neq$ post)
accept:	post (data) $\wedge$ *ok?*

Oracle/Fault-Based

- Apply fault-based techniques to implementation
  - detect implementation faults by revealing violation of specification oracle

oracle/fault-based test case	
input:	(PC and *revealing*(*variant*(source) $\neq$ source) and (*variant*(post) $\neq$ post)
accept:	post (data) and *ok?*

Specification Languages

- Anna
  - annotations: assertions
  - package specifications: axiomatic
  - tied to Ada
- Larch
  - shared language: algebraic
  - interface language: pre/post-conditions
  - not as tied to the implementation
- Future Candidates
  - InaJo, Aslan (Kemmerer): state-based
  - Z (Abrial, Spivey): model-based
  - Refine (Green): set theory and logic
  - RTIL (Razouk and Gorlick): temporal logic

Conclusion

- Use of test cases
  - test adequacy metrics
  - test data selection
  - test oracle
- Other Considerations
  - specification testing
  - test case inheritance
  - test case specifications
  - non-functional specifications

Anna
Specification-Based Testing

- ANNotated Ada
  (assertions in Ada specifications and implementations)
  - Virtual text
  - Quantified expressions
  - Annotations
    - type annotations
    - object annotations
    - statement annotations
    - subprogram annotations
    - exception annotations
  - Package axioms

SquareRoot Example

```
function SquareRoot (N: NATURAL) return NATURAL is
  --| where return S:NATURAL => S**2 <= N < (S+1)**2;

    Low  : NATURAL := 1;          --| Low <= Mid;
    High : NATURAL := N/2+1;   --| High >= Mid;
    Mid  : NATURAL := (Low+High)/2;

  begin
    loop
      --| Low**2 <= N <= High**2;
      exit when (Mid**2 <= N) and ((Mid+1)**2 > N);
      if Mid**2 > N then
        High := Mid;
      elsif Low = Mid then
        Low := Low+1;
      else
        Low := Mid;
      end if;
      Mid := (Low+High)/2;
    end loop;
    return Mid;
  end SquareRoot;
```

Anna Example Specification/Error-Based

- Consider the object annotation Low $\leq$ Mid
- pre-condition = object annotations
  post-condition = object and loop annotations

pre:	[(Low $\leq$ Mid) and (High $\geq$ Mid)]
	$(1 \leq ((N_{pre}/2+1)+1)/2)$ $\wedge$ $(N_{pre}/2+1 \geq ((N_{pre}/2+1)+1)/2)$
post:	[(Low $\leq$ Mid) and (High $\geq$ Mid) and (Mid**2 $\leq$ N $<$ (Mid+1)**2)]
	$(Low_{post} \leq Mid_{post}) \wedge (High_{post} \geq Mid_{post})$ $\wedge$ $(Mid_{post}**2 \leq N < (Mid_{post}+1)**2)$

- Domain test case — *boundary*(pre):

input:	$(1=((N_{pre}/2+1)+1)/2)$ $\wedge$ $(N_{pre}/2+1 \geq ((N_{pre}/2+1)+1)/2)$
accept:	$(Low_{post} \leq Mid_{post}) \wedge (High_{post} \geq Mid_{post})$ $\wedge$ $(Mid_{post}**2 \leq N < (Mid_{post}+1)**2)$ $\wedge$ *ok?*

- Test data and fault detection
  - N = 0 causes violation of loop annotation
  - Low should be initialized to 0

Anna Example
Oracle/Error-Based

- Choose a path through SquareRoot
  to Low := Low + 1;
  (8,9,10,11,12,14,15)

- Symbolic evaluation results

PC:	[not ((Mid**2 $\leq$ N) and ((Mid+1)**2 > N)) and not (Mid**2 > N) and (Low = Mid)]
	$((((N_{in}/2+1)/2)**2 > N_{in}) \vee$ $(((N_{in}/2+1)/2+1)**2 \leq N_{in}))$ $\wedge (((N_{in}/2+1)/2+1)**2 \leq N_{in})$ $\wedge (0=(N_{in}/2+1)/2)$
PV:	$N=N_{in}$, Low=1, High = $N_{in}/2+1$, Mid = $(N_{in}/2+1)/2$

- Test case to violate annotation on Low

input:	[PC $\wedge$ not (Low $\leq$ Mid)]
	PC $\wedge$ ($N_{in} > (N_{in}/2+1)/2$)
accept:	*false*

- Test data and fault detection
  - N = 1 reveals inconsistency between incrementing Low and Low's object annotation
  - Low should be annotated by Low $\leq$ Mid + 1

Anna Example Specification/Fault-Based

- Hypothesize that High's object annotation
  High $\geq$ Mid
  should be:
  High $>$ Mid

- Symbolic evaluation and RELAY results

PC:	*true*
OC:	[High = Mid]
	$N_{in}/2 + 1 = (N_{in}/2 + 1)/2$

- Test data and fault detection
  - Revealing condition is infeasible, High = Mid is not satisfiable
  - hence not a fault
  - High should be annotated by High $>$ Mid

Anna Example
Oracle/Fault-Based

- Hypothesize that
  High := N/2+1
  should be:
  High := (N+1)/2

- Symbolic evaluation and RELAY results

PC:	*true*
OC:	$N_{in}/2+1 \neq (N_{in}+1)/2$
TC:	*true*

- Test case

input:	PC ∧ (OC ∧ TC) ∧ $((N_{in} \leq (N_{in}/2+1)**2) \wedge$ $(N_{in} > ((N_{in}+1)/2)**2))$ $\vee ((N_{in} > (N_{in}/2+1)**2) \wedge$ $(N_{in} \leq ((N_{in}+1)/2)**2))$
accept:	$((Low**2 \leq N_{in}) \wedge (N_{in} \leq High**2))$ ∧ *ok?*

- Test data and fault detection
  - N = 2 causes violation of loop annotation for variant but not for original, although both versions run correctly
  - loop should be annotated by
    $Low**2 \leq N \leq (High+1)**2$

Paper 4-M-2

ESTABLISHING THE QA FUNCTION ON A LIMITED BUDGET AND SCHEDULE

Mr. Daniel R. Sullivan
The Computer Company (Richmond, VA)

Mr. Daniel R. Sullivan is employed by The Computer Company in Richmond, VA. He is currently serving as Systems Development Manager. He has held numerous positions with the firm over the last eleven years, including Product Development Manager and Technical Consultant. He developed the testing standards for the company's design and development division, and has served as Testing Manager. Mr. Sullivan was previously employed by McDonnell-Douglas Automation Company (MCAUTO). He received a B.S. in Mathematics-Computer Science from Bradley University, and an M.S. in Computer Science from the University of Missouri-Rolla. He is a Certified Systems Professional (CSP).

Establishing the QA Function

on a Limited

Budget and Schedule

Presented By:
Daniel R. Sullivan
The Computer Company
Richmond, Virginia

Synopsis

The purpose of this presentation is to provide a case study that depicts the establishment of a Quality Assurance program within an organization. The QA program was developed in a short time frame to support a specific project, which had contractual requirements for defining a project Test Plan and delivering Test Results at various phases of the project.

As will be seen, the time frame and constraints were such that the program needed to be established quickly and at minimal cost.

This presentation will discuss the steps taken to establish the QA program, the results of those efforts, and some of the lessons learned from the experience.

AGENDA

1. Background Information
   - The Company
   - The Project
   - The Rules

2. Approach Taken

3. Results

4. Lessons Learned

The Computer Company

- Established in 1969

- Primary business is Medicaid

- 1989 Revenues in Excess of $50 Million

- A Subsidiary of First Financial Management Corp. (FFMC)

Alaska Medicaid Project

- Contract awarded December, 1986
- Scope of Work to be Performed
- Requirements Analysis Initiated December, 1986
- Test Manager Identified March, 1987
- Test Plan Presented May, 1987
- Systems Design & Development Began June, 1987
- System Testing Began October, 1987
- Acceptance Testing Began January, 1988
- Production Processing Started April, 1988

The Testing Charter

- Satisfy Contractual Obligations
- Ensure System Functions Properly
- Develop Standards for Future Projects
- Perform Work Within Budget (Hours) Established

Approach Taken

- Studied Available Literature
- Defined a QA Plan and Testing Methodology
- Developed a Test Management System
- Conducted Internal Training in Methodology
- Established an Independent Test Team
- Adjusted Approach As Necessary

SYSTEM/ACCEPTANCE TEST CASE SPECIFICATIONS

PROJECT: DATE:

SUBSYSTEM:

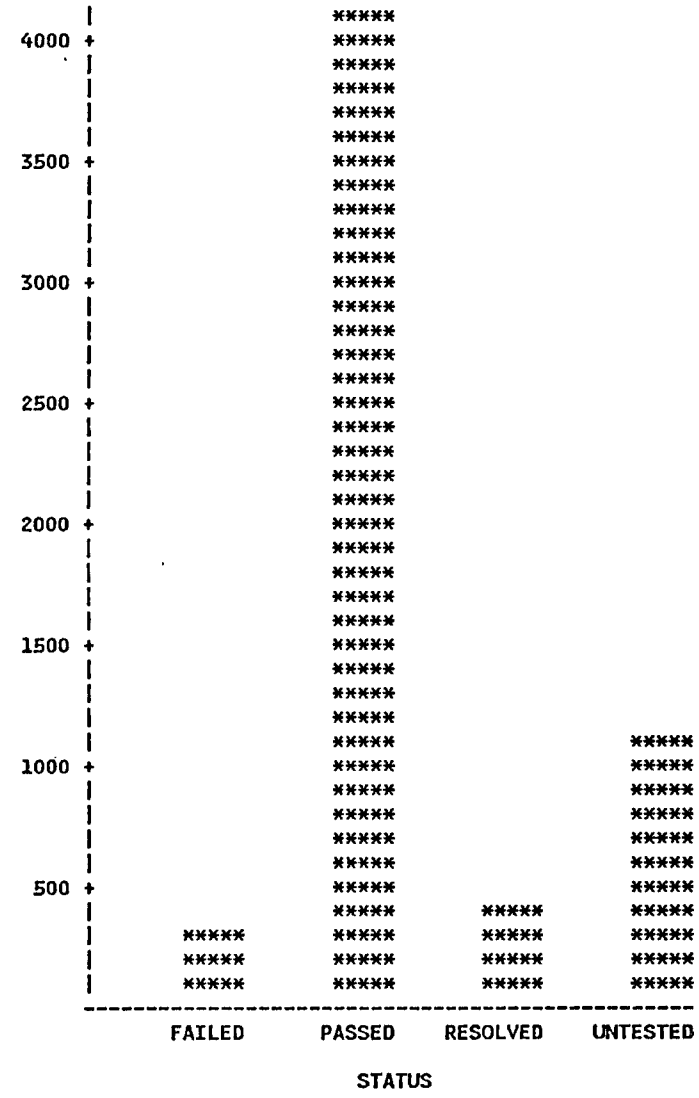
MODULE: SITUATION NO:

TEST CASE NUMBER: DESCRIPTION: OBJECTIVE(S):
INPUT SPECS:
EXPECTED RESULTS:
ACTUAL RESULTS/LOCATION:
COMMENTS:

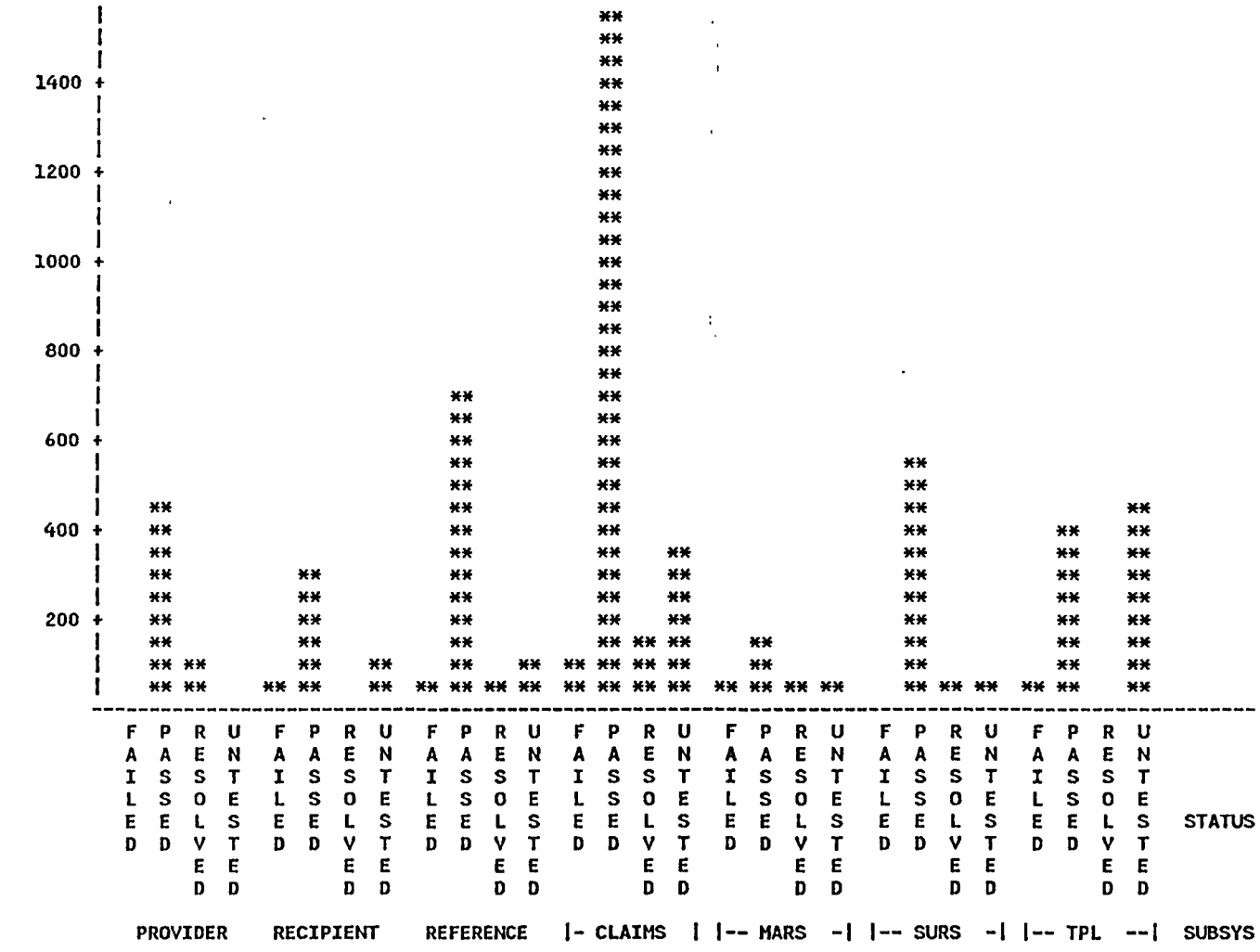
PREPARED BY: _____ APPROVED BY: _____

TEST SITUATION	DESCRIPTION	TEST CASE	OBJECTIVE(S)
04 5.1.1	EDIT INQUIRY REQUEST	01	TEST INVALID FCN (INVALID FOR VARIOUS REASONS ACCORDING TO DOCUMENTATION IN IMPUT SPECS).
	EDIT INQUIRY REQUEST	02	TEST INVALID FCN - (BECAUSE OF DUPLICATION).
04 5.1.2	DISPLAY INQUIRY DATA	01	VERIFY SCREEN ELEMENTS
04 5.2.1	PROCESS REFUND/RETURN REASON S	01	VERIFY THAT THERE ARE PROCEEDURES IN PLACE THAT ENSURE PROPER COMPLETION OF REFUND/RETURN REASON SHEET CP-I-06.
04 5.2.2	PROCESS ADD PAY/RECOVERY TRANS	01	TEST TO DETERMIN PROCEEDURES ARE IN PLACE FOR PROCESSING ADD PAY/ RECOVERY TRANSACTION REQUEST CP-I-06
04 5.2.3	EDIT ADD TRANSACTION CP-I-06-1	01	TEST INVALID FCN.
	EDIT ADD TRANSACTION CP-I-06-1	02	VERIFY ALL FIELDS OF EDIT ADD TRANSACTION SCREEN ** ALSO VERIFY VALID LEAP YEAR FCN.
	EDIT ADD TRANSACTION CP-I-06-1	03	VERIFY FIELD ACCEPTANCE FOR REASON CODE #01
		04	VERIFY FIELD ACCEPTANCE FOR REASON CODE #02
		05	VERIFY FIELD ACCEPTANCE FOR REASON CODE #08
		06	VERIFY FIELD ACCEPTANCE FOR REASON CODE #09
		07	VERIFY FIELD ACCEPTANCE FOR REASON CODE #10
		08	VERIFY FIELD ACCEPTANCE FOR REASON CODES 11 THROUGH 14
		09	VERIFY FIELD ACCEPTANCE FOR REASON CODES 16
		10	VERIFY FIELD ACCEPTANCE FOR REASON CODES 17 AND 18
		11	VERIFY FIELD ACCEPTANCE FOR REASON CODES 19
		12	VERIFY FIELD ACCEPTANCE FOR REASON CODES 20
		13	VERIFY FIELD ACCEPTANCE FOR REASON CODES 21 THROUGH 25, 31, 37.
		14	VERIFY FIELD ACCEPTANCE FOR REASON CODES 26 THROUGH 30 & 90.
		15	VERIFY FIELD ACCEPTANCE FOR REASON CODES 32 THROUGH 36 & 38.
		16	VERIFY FIELD ACCEPTANCE FOR REASON CODES 39.
		17	VERIFY FIELD ACCEPTANCE FOR REASON CODES 40 THROUGH 44, 49, ALSO 60 THROUGH 67, 69, 73 THROUGH 75.
		18	VERIFY FIELD ACCEPTANCE FOR REASON CODES 45 THROUGH 48, 68.
		19	VERIFY FIELD ACCEPTANCE FOR REASON CODES 50 THROUGH 55, 20.
04 5.2.4	PROCESS ADD PAY/RECOVERY TRANS	01	TEST TO DETERMIN PROCEEDURES ARE IN PLACE FOR PROCESSING ADD PAY/ RECOVERY TRANSACTION REQUEST CP-I-06

FREQUENCY BAR CHART
FREQUENCY



FREQUENCY BAR CHART
FREQUENCY



Results

- Methodolgy Used Worked Well
- Customer Was Pleased
- Testing Effort Suffered Due to Development Overruns
- Project Test Plan Converted to Standards
- Test Cases Defined Applicable To Future Projects

Lessons Learned

- Better Use of Risk Analysis
- Developers' Attitude Towards Testers
- The State of Testing
- Capturing Useful Metrics
- The Effects of Testing
- Regression Testing
- The Effectiveness of Reviews
- Other Resources

References

- The Complete Guide to Software Testing
  William Hetzel
  QED Information Sciences, Inc. 1984

- The Art of Software Testing
  Glenford J. Myers
  John Wiley & Sons, Inc. 1979

- IEEE Guide for Software Quality Assurance Planning
  Standard 983-1986
  IEEE, Inc. 1986

- IEEE Standard for Software Test Documentation
  Standard 829-1983
  IEEE, Inc. 1983

- "Packaging your testing tool box"
  Bob Stahl
  Computerworld
  October 9, 1989

- "Systematic software testing requires formal training"
  Mike Feuche
  Computerworld
  March 19, 1990

Paper 4-M-3

LINKING PRODUCTIVITY AND QUALITY

Mr. Leonard White
Productivity Management Group

Mr. Leonard R. White is president and founder of Productivity Management Group, Inc., a consulting firm which specializes in resolving data processing productivity issues. The Productivity Management Group creates productivity measurement programs addressing hardware, software, and personnel issues for companies with large data processing staffs. Len is a member of the International Function Point Users Group, and currently serves as editor for the Management Reporting Manual. He has spoken at industry conferences on productivity issues. Len received a Bachelor's degree in Business Administration from Virginia Polytechnic Institute in Blacksburg, Virginia. He received his Master's degree in Computer Science from Rensselaer Polytechnic Institute in Troy, New York.

PRODUCTIVITY MANAGEMENT GROUP, INC.

LINKING PRODUCTIVITY AND QUALITY

SOFTWARE RESEARCH, INC.

QUALITY WEEK 1990

"QUALITY & RISK MANAGEMENT"

PRODUCTIVITY MANAGEMENT GROUP, INC.

OBJECTIVES

- TO INTRODUCE PRODUCTIVITY AS PART OF QUALITY
- TO DEMONSTRATE EVALUATION OF PRODUCTIVITY & QUALITY

PRODUCTIVITY MANAGEMENT GROUP, INC.

DATA PROCESSING

AND

DOMINOS PIZZA

PRODUCTIVITY MANAGEMENT GROUP, INC.

QUALITY IS

DIFFICULT TO DEFINE

BECAUSE

IT IS BEAUTIFUL

Leonard White, President

(716) 689-7724

178 Foxhunt Lane

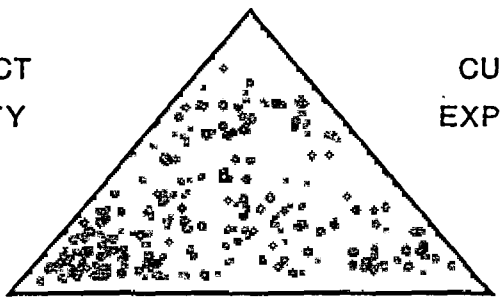
East Amherst, NY 14051

PRODUCTIVITY MANAGEMENT GROUP, INC.

QUALITY

PRODUCT QUALITY

CUSTOMER EXPECTATION



PRODUCT USE

PRODUCTIVITY MANAGEMENT GROUP, INC.

STRICT DEFINITIONS OF FACTS

ENHANCE THE MEASUREMENT PROCESS

Leonard White, President

(716) 689-7724

178 Foxhunt Lane

East Amherst, NY 14051

PRODUCTIVITY MANAGEMENT GROUP, INC.

FACTS SHORTEN

THE DISTANCE BETWEEN

PERCEPTION AND REALITY

PRODUCTIVITY MANAGEMENT GROUP, INC.

STRICT DEFINITIONS OF QUALITY

& PRODUCTIVITY NARROW THE SCOPE

OF QUALITY & PRODUCTIVITY

PROGRAMS AND INHIBIT IMPROVEMENT

Leonard White, President

(716) 689-7724

178 Foxhunt Lane

East Amherst, NY 14051

PRODUCTIVITY MANAGEMENT GROUP, INC.

MEASURING QUALITY & PRODUCTIVITY

- DEVELOPMENT RATE $\frac{\text{FUNCTION POINTS}}{\text{WORK MONTH}}$
- SUPPORT RATE $\frac{\text{WORK HOURS}}{\text{FUNCTION POINTS}}$
- QUALITY INDEX $\frac{\text{HOURS TO REPAIR DEFECTS}}{\text{FUNCTION POINTS}}$

PRODUCTIVITY MANAGEMENT GROUP, INC.

MATRIX FOR CLASSIFYING EXTERNAL INPUTS

FILES REFERENCED	DATA ELEMENTS		
	1-4	5-15	GT 15
0 or 1	L	L	A
2	L	A	H
OVER 2	A	H	H

Leonard White, President

(716) 689-7724

178 Foxhunt Lane

East Amherst, NY 14051

PRODUCTIVITY MANAGEMENT GROUP, INC.

FUNCTION POINT EXAMPLE

RAW DATA		WEIGHTS			
10	INPUTS	X	4	=	40
10	OUTPUTS	X	5	=	85
10	INQUIRIES	X	4	=	40
1	DATA FILES	X	10	=	10
1	INTERFACES	X	7	=	7
32					182 FUNCTION POINTS

UNADJUSTED FP X COMPLEXITY FACTOR = TOTAL FP.
182 X 1.10 = 200

PRODUCTIVITY MANAGEMENT GROUP INC.

SYSTEM COMPLEXITY FACTORS

CHARACTERISTIC	RATING	CHARACTERISTIC	RATING
DATA COMMUNICATIONS	5	ON-LINE UPDATE	4
DISTRIBUTED FUNCTIONS	0	COMPLEX PROCESSING	2
PERFORMANCE	3	REUSABILITY	0
HEAVILY USED CONFIGURATION	2	INSTALLATION EASE	4
TRANSACTION RATE	3	OPERATIONAL EASE	5
ON-LINE DATA ENTRY	5	MULTIPLE SITES	2
END USER EFFICIENCY	5	FACILITATE CHANGE	5
		TOTAL RATING	45

SYSTEM COMPLEXITY = TOTAL RATING X .01 + .65 1.10

NOT PRESENT = 0 AVERAGE INFLUENCE = 3

MINOR INFLUENCE = 1 SIGNIFICANT INFLUENCE = 4

MODERATE INFLUENCE = 2 STRONG INFLUENCE = 5

Leonard White, President

(716) 689-7724

178 Foxhunt Lane

East Amherst, NY 14051

PRODUCTIVITY MANAGEMENT GROUP, INC.

MEASURE SPEED OF DELIVERY WITH DELIVERY RATE

MEASURE LONG TERM VALUE WITH SUPPORT RATE

MEASURE ERRORS WITH A QUALITY INDEX

PRODUCTIVITY MANAGEMENT GROUP, INC.

IMPROVE CONSTANTLY AND FOREVER
THE SYSTEM OF
PRODUCTION AND SERVICE

- ONLY Management Can Initiate Changes in Quality and Productivity
- People Don't Adopt Change - They Adapt To It
- Significant Improvement Requires Significant Change
- Can't Manage What You Can't Measure

PRODUCTIVITY MANAGEMENT GROUP, INC.

CREATE CONSTANCY OF PURPOSE FOR IMPROVEMENT OF PRODUCT & SERVICE

- Understand The Process - Which One?
- Invest In The Future - Of What?
- Mobility Of Management - Stop It!
- Emphasis On The Short Term - Reduce It!

Leonard White, President

178 Foxhunt Lane

(716) 689-7724

East Amherst, NY 14051

Paper 4-M-4

APPLICATIONS OF SOURCE CODE ANALYZERS

Dr. Stephen Carr
IPT Corporation

Dr. C. Stephen Carr graduated from the University of California at Berkeley with a bachelors and masters degree in Electrical Engineering in 1965 and was awarded a PhD in Computer Science from the University of Utah at Salt Lake City in 1969. Dr. Carr founded Information Processing Techniques (IPT) in 1974 and today runs the organization of forty professionals. IPT performs programming and product design services for major US manufactures including Computervision, EG&G, Smith-Kline, GE, 3M and many more. IPT develops and markets source code analyzers in support of software development.

Application Of Source Code Analyzers

by

C. Stephen Carr

Compiling the following three C source files with

cc ticket check insert

will not produce any errors. Yet, when running lint-PLUS™ over the same three source files, numerous problems are immediately detected. The following is a source listing of the three source files and the two included header files; ticket.c, check.c, insert.c, limit1.h and limit2.h:

```
/******************** LIMIT1.H *******************/
#define SPEED_LIMIT 55

/******************* LIMIT2.H *******************/
#define SPEED_LIMIT 65
struct s1 {
       long id;
       int num;
       struct s1 *llink, *rlink;
       struct s2 *namelist;
       } *tab1;
struct s2 { char name[10]; struct s2 *link; } *tab2;

/******************* CHECK.C *******************/
#include "limit1.h"
#include "limit2.h"
check_id(speed)
int speed;
{
    return(speed > SPEED_LIMIT);
}

/******************* INSERT.C *******************/
#include <stdio.h>
#include "limit2.h"
struct s1 *install();
struct s1 *insert(name,id)
char *name;
long id;
{
     struct s1 *p1, *q1;
     struct s2 *p2;
     p1 = tab2;
     while(p1 != NULL && p1 != id) {
           q1 = p1;
           if(p1->id > id) p1 = p1->rlink;
           else p1 = p1->link;
      }
      if (p1 == NULL) p1 = install(id,q1);
      p2 = (struct s2 *)calloc(sizeof(*p2));
      p1->namelist = p2;
      strcpy(p2->link,id);
}
```

```
/******************* TICKET.C *******************/
#include "limit2.h"
struct s1 *insert();
ticket(speed,name,id)
char speed, *name;
int id;
{
     if (check_id(speed,name))
     printf ("speed violation >>> citation no %d\n",  \
              insert(name,id)-> num);
}
```

The following is the output from lint-PLUS™ using the following command:

lintplus /all/xref/stat/tree=ticket/func ticket insert check

```
/******************** lint-PLUS OUTPUT ********************/

        lintPLUS rev 2.92          CHECK.C;2   28-Sep-1989, 10:30

#define SPEED_LIMIT 65
^
limit2.h line 1: warning # 45: Redefined macro: "SPEED_LIMIT"
>        If you want to redefine it, use #undef to undefine it first.

    1 warning(s)

        lintPLUS rev 2.92         INSERT.C;2   28-Sep-1989, 10:30

     p1 = tab2;
            ^
INSERT.C line 11: warning # 76: Dissimilar pointer types.
>        When programs are being moved from one CPU to another,
>        addressing differences can cause havoc. Consider the effect
>        of the operation on all potential target machines.

     while(p1 != NULL && p1 != id) {
                                ^
INSERT.C line 12: warning # 66: Improper mixture of integers and
                                pointers.
>        Perhaps a function should have been declared as returning a
>        pointer. If it is necessary to mix integers and pointers, use
>        a typecast.

          else p1 = p1->link;
                       ^
INSERT.C line 15: error # 53: No such element in this structure: "link"
>        This implementation  requires  that  structure elements be
>        used only with their own type of structure.

          else p1 = p1->link;
                       ^
INSERT.C line 15: warning # 66: Improper mixture of integers and
pointers.
```

```
}
^
INSERT.C line 21: hint # 88: Possible missing return statement.
>        The current function is supposed to return something, but it
>        may be possible for it to fall off the end without doing so.

}
^
INSERT.C line 21: hint # 69: Unused identifier: "name"
>        An identifier declared outside of a header file was not
>        referenced within the scope in which it was declared. It
>        should probably be deleted or moved to a header file.

    2 hint(s)

    3 warning(s)

    1 error(s)

                Beginning global checking...

       INSERT.C line  18: not enough args: calloc

       INSERT.C line  20: pointer mismatch: strcpy, arg #1

            - Defined Arg Of Type > Character Pointer
            - Called  Arg Of Type > Structure Pointer

       INSERT.C line  20: pointer expected: strcpy, arg #2

            - Defined Arg Of Type > Character Pointer
            - Called  Arg Of Type > Integer/Long

       TICKET.C line   7: mix of char and int: check_id, arg #1

            - Defined Arg Of Type > Integer
            - Called  Arg Of Type > Character

       TICKET.C line   7: too many args: check_id, arg # 2

       TICKET.C line   8: long/int confusion: insert, arg #2

            - Defined Arg Of Type > Integer/Long
            - Called  Arg Of Type > Integer

/********** lint-PLUS™'s cross reference table **********/

                ---- Cross-reference ----

    calloc defined VMS.C line 284
           referenced: INSERT.C line 18
  check_id defined CHECK.C line 3
           referenced: TICKET.C line 7
   freopen defined VMS.C line 20
           referenced: stdio.h line 57
    insert defined INSERT.C line 5
           referenced: TICKET.C line 2
```

```
            referenced: TICKET.C line 8
   install is undefined
            referenced: INSERT.C line 4
            referenced: INSERT.C line 17
    printf defined VMS.C line 84
            referenced: TICKET.C line 8
    strcpy defined VMS.C line 160
            referenced: INSERT.C line 20
      tab1 defined limit2.h line 7
      tab2 defined limit2.h line 8
    ticket defined TICKET.C line 3

              ***File & Function List***

               CHECK.C:            check_id

               INSERT.C:           insert

               TICKET.C:           ticket

*** This is a primary tree (ticket) ***

   [] -- root of a tree       <> -- recursive call
   {} -- undefined            () -- duplicate subtree

[ticket:0]-+-check_id
           |
           +-{printf}
           |
           +-insert-+-{install}
                    |
                    +-{calloc}
                    |
                    +-{strcpy}

              ***** STATISTICS *****

     Source files:       3
     Source lines:       37
     Include files:      5
     Include lines:      82
     Functions:          3

RUNTIME STATISTICS:
-------------------

    LOCAL MODE  :

       ELAPSED TIME = 00:00:26.51
           CPU TIME = 0:00:18.40

    GLOBAL MODE :

       ELAPSED TIME = 00:00:38.28
           CPU TIME = 0:00:21.46
```

Application Of Source Code Analyzers

by
C. Stephen Carr

Compiling the following Fortran source file with

FORT DEMO.FOR

produces no errors. Yet, when running FORTRAN-lint™ over the same source file, 14 potential problems are discovered. The following is a source listing of "DEMO.INC" and "DEMO.FOR":

```
C******************* DEMO.FOR *******************
C  'PROCDAT'
       PROGRAM PROCDAT
       INTEGER IUNIT, PUNIT
       INCLUDE 'DEMO.INC'
       DO 100 I = 1, 5
 50      CALL GETUNIT( I+5, IUNIT, PUNIT)
         CALL READNAME( CURITEM.NAME, CURITEM.DIMENSIONS)
         CALL SETTYPE( CURITEM)
         CALL PRINT( CURITEM, IUNIT)
100    CONTINUE
       IF (IUNIT .EQ. 23) GO TO 50
       END
C 'GETUNIT'
       SUBROUTINE GETUNIT( UNIT, UNIT1)
       INTEGER UNIT, UNIT1
       READ (UNIT1,*) UNIT
       END
C 'READNAME'
       SUBROUTINE READNAME( NAME, DIMS)
       CHARACTER*(*) NAME
       INTEGER INUSE, STATUS
       COMMON /BLOCK/ INUSE, STATUS
       REAL*8 DIMS(3)
       READ (5, *) NAME, DIMS
       END
C 'SETTYPE'
       SUBROUTINE SETTYPE( CURITEM)
       INCLUDE 'DEMO.INC'
       CURITEM.TYPE = CURITEM.DIMENSIONS(2)
       IF (CURITEM.TYPE .GT. 5) CALL PRINT( CURITEM)
       END
C 'PRINT'
       SUBROUTINE PRINT( CURITEM, IUNIT)
       INCLUDE 'DEMO.INC'
       IF (CURITEM.TYPE .NE. COUNT) CALL PRINTIT( IUNIT, CURITEM)
       END
C 'PRINTIT'
       SUBROUTINE PRINTIT( IUNIT, CURITEM)
       INCLUDE 'DEMO.INC'
       IF (IUNIT .EQ. INUSE) THEN
         STATUS = 2
         CALL DIPSTAT( 4, CURITEM)
         CALL GETUNIT( INUIT, 3)
       END IF
```

```
      WRITE (IUNIT,*) CURITEM.TYPE
      END
C 'DIPSTAT'
      SUBROUTINE DIPSTAT( ISTAT, CURITEM)
      ISTAT = PRINT( CURITEM, 1)
      END

C******************** DEMO.INC *******************
      STRUCTURE /ITEM/
        CHARACTER*10 NAME
        INTEGER TYPE
        REAL DIMENSIONS(3)
      END STRUCTURE
      RECORD /ITEM/ CURITEM

      INTEGER INUSE*2, STATUS, COUNT, TIME
      COMMON /BLOCK/ INUSE, STATUS
      COMMON /BK2/ COUNT, TIME

C******************* END OF SOURCE *******************
```

The following output from FORTRAN-lint™ was generated using the following command:

FLINT /FYI /GLOBAL /XREF /STAT /TREE DEMO.FOR

```
FORTRAN-lint    Rev 2.61                    30-Apr-90  10:46:12

Options: /GLOBAL/FYI/TREE/STATISTICS/OUTPUT=demo.out

*********************************************************************
      Program PROCDAT                    File DEMO                 Line 2

> 50      CALL GETUNIT( I+5, IUNIT, PUNIT)
>                       ^
DEMO:PROCDAT line 6: INTERFACE WARNING #63- expression is changed
                     by subprogram.

> 50      CALL GETUNIT( I+5, IUNIT, PUNIT)
>                                   ^
DEMO:PROCDAT line 6: INTERFACE ERROR #57- too many arguments.

>         CALL READNAME( CURITEM.NAME, CURITEM.DIMENSIONS)
>                                      ^
DEMO:PROCDAT line 7: INTERFACE ERROR #252- R*4 array passed to
                     dummy arg which is a R*8 array.

>       IF (IUNIT .EQ. 23) GO TO 50
>                                ^
DEMO:PROCDAT line 11: SYNTAX WARNING #47- branch into do loop
                      via label 50.

USAGE ERROR #126- local variables referenced but never set:
                  IUNIT (Line 6)
*************************************************************
```

```
Subroutine READNAME                    File DEMO              Line 19

DEMO:READNAME line 22: INTERFACE WARNING #185- common block
                       /BLOCK/  length mismatch (compared  to
                       initial use in routine PROCDAT).

DEMO:READNAME line 22: INTERFACE WARNING #122- common block
                       /BLOCK/ organization differs at member
                       INUSE (compared to initial use in  routine
                       PROCDAT).

*****************************************************************
Subroutine SETTYPE                File DEMO                 Line 27

>       IF (CURITEM.TYPE .GT. 5) CALL PRINT( CURITEM)
>                                                   ^
DEMO:SETTYPE line 30: INTERFACE ERROR #56- not enough arguments.

******************************************************************
Subroutine PRINTIT                 File DEMO                Line 38

>         CALL DIPSTAT( 4, CURITEM)
>                       ^
DEMO:PRINTIT line 42: INTERFACE ERROR #59- constant is changed by
                      subprogram.

>         CALL DIPSTAT( 4, CURITEM)
>                          ^
DEMO:PRINTIT line 42: INTERFACE ERROR #248- struct ITEM passed to
                      a R*4 dummy arg.

USAGE WARNING #127- local variables set but never referenced:
                    INUIT (Line 43)

******************************************************************
Subroutine DIPSTAT       File DEMO                        Line 48

>       ISTAT = PRINT( CURITEM, 1)
>               ^
DEMO:DIPSTAT line 49: INTERFACE ERROR #95- this name is used as a
                      subroutine.

*****************************************************************

FORTRAN-lint    (global checking)            30-Apr-90  10:46:12

Global checking:

*** Inconsistent organization of common /BLOCK/,
      ref/set checking suppressed for this common block

USAGE ERROR #133- common block members referenced but not set:
                  /BK2/COUNT

USAGE FYI #135- unused common block members:  /BK2/TIME
```

```
FORTRAN-lint    (call tree)                    30-Apr-90  10:46:12

Call tree:

This is a primary tree starting at the program 'PROCDAT'

PROCDAT-+-GETUNIT
        |
        +-READNAME
        |
        +-SETTYPE--PRINT--PRINTIT-+-DIPSTAT--*PRINT*
        |                         |
        |                         +-GETUNIT
        |
        +-PRINT--PRINTIT-+-DIPSTAT--*PRINT*
                         |
                         +-GETUNIT


FORTRAN-lint    (statistics)           30-Apr-90  10:46:12

>>> Statistics:

 Number of source files:     1

 Source files:   50 lines,  1271 bytes (  6% comments, 94% code )
 Include files:  44 lines,  1052 bytes ( 14% comments, 86% code )
 Total parsed:   94 lines,  2323 bytes ( 10% comments, 90% code )

 Total subprograms:      7
   Subroutines:          6
   Functions:            0
   Program:              1
   Block Data:           0

 Total messages: 14

                 Errors  Warnings   FYIs
                -------- -------- --------
    Syntax:           0        1        0
    Interface:        6        3        0
    Data usage:       2        1        1

    Implicit typing:       <supp>
    Portability issues:    <supp>
    ANSI extensions:       <supp>

 Elapsed time:       0 00:00:06.19
 Elapsed CPU time:   0 00:00:03.28
 Buffered I/O count:        34
 Direct I/O count:          90
 Page faults:              199
```

Paper 4-T-2

PRACTICAL MODULE REGRESSION TESTING TOOLS & TECHNIQUES

Prof. Daniel Hoffman
Dept. of Computer Science
University of Victoria

Dr. Daniel Hoffman received the Ph.D. degree in computer science in 1984, from the University of North Carolina, Chapel Hill, and is currently an Assistant Professor of Computer Science at the University of Victoria in B.C., Canada. His research area is software engineering, emphasizing software specification and testing.

Practical Module Regression Testing Tools and Techniques

Dr. Daniel Hoffman

University of Victoria
Department of Computer Science
Victoria, B.C. Canada

Talk Overview

- Modules and interfaces
- Testing strategy
- Test case description
- Test harness generation
- Design for testability
- Experience
- Conclusions

Modules and interfaces

- Module

  A programming work assignment

- Module interface

  The set of assumptions users are permitted to make about the module

- Call-based interface

  Communication through *access programs*

- Exception handling

  Exceptions explicitly specified, detected, signaled

- Module traces

  Sequences of calls on the module

Symbol Table Example

Acc. pgms.	Inputs	Outputs	Exceptions
s_init			
s_addsym	string		maxlen tblfull
s_delsym	integer		
g_cnt		integer	
g_legsym	string	boolean	
g_legid	integer	boolean	
g_sym	integer	string	notlegid
g_id	string	integer	notlegsym

- Two traces

```
s_init().s_addsym("cat").g_legsym("cat")
s_init().s_addsym("cat").g_id("dog")
```

Testing Tasks

(1) Build test harness

(2) Select inputs

- Structural coverage, dataflow coverage
- Functional testing

(3) Determine expected outputs

- The *test oracle* problem

(4) Execute tests

(5) Compare actual outputs against expected outputs

(6) Evaluate results

○ Costs of all 6 steps must be considered

Testing Principles

- Systematic module testing
  - For both development and maintenance
- Design for testability
  - Module designer responsible for ensuring testability
  - Controllability and observability
- Cost-effective automation
  - Test harness construction
  - Test execution
  - Actual output checking
- Practical constraints
  - Widely applicable now or in near future
  - Minimal special training required
  - Not dependent on compiler modifications

Test Case Description

- Test case syntax

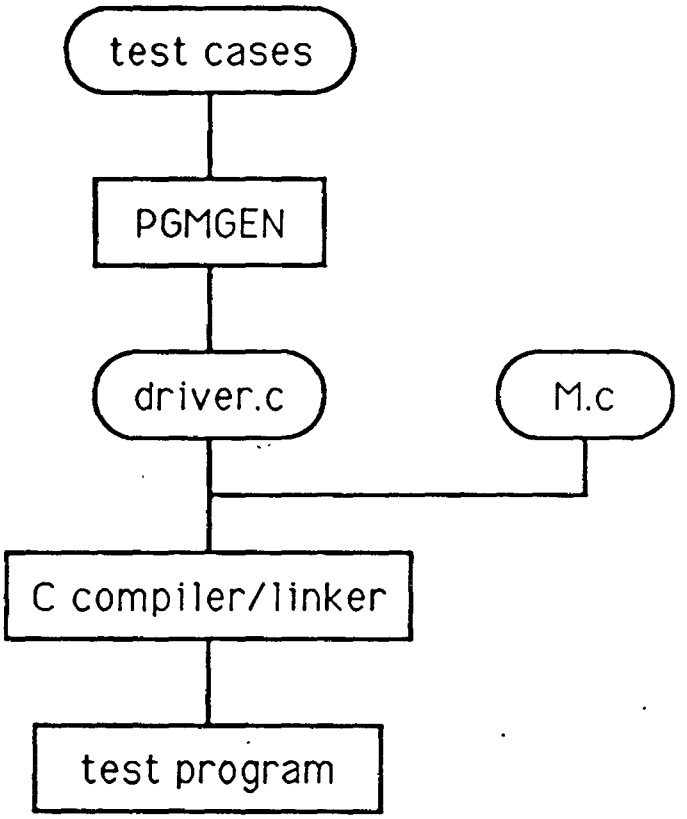
  $< trace, expexc, actval, expval >$

  - *trace*
    trace used to exercise the module
  - *expexc*
    exception that *trace* is expected to generate
  - *actval*
    expression evaluated after *trace*
    — the "actual value" of the trace
  - *expval*
    the value that *actval* is expected to have

- Two test cases

```
<s_init().s_addsym("cat"),
     noexc, g_legsym("cat"), 1, boolean>

<s_init().s_addsym("cat").g_id("dog"),
     notlegsym, dc, dc, dc>
```

Test Harness Generation



Test Harness Generation

Generate code to record exception occurrences

For each test case of the form:

$$< c_1.c_2.....c_N, expexc, actval, expval, type >$$

generate code to:

```
invoke c1, c2, ..., cN
compare actual exception occurrences against expexc
if there are any differences
        print a message
else
        if actval ≠ expval
                print a message
        if exceptions occurred since cN was invoked
                print a message
update summary statistics
```

Generate code to print summary statistics

Design for testability — case study

- The GRADES system
  - Maintain assignment, student, score information
  - Spreadsheet-style screen editor
  - Detail and summary reports
  - Implemented in C; 19 modules; 10K lines source

- Applying design for testability
  - Initialization calls
  - Controllability problems

    Keyboard input
  - Observability problems

    Screen, file output

Design for testability — case study

- Call-based input/output — 9 modules
  - Few controllability or observability problems
  - Many test cases required to handle input combinations
- Keyboard input — 2 modules
  - Controllability: keystrokes supplied manually
  - Observability: echoing checked manually
  - Approach: isolate; test manually
- Screen output — 5 modules
  - Observability: screen contents checked manually
  - Approach: isolate; display patterns
- Report output — 3 modules
  - Observability: output difficult to check from driver
  - Approach: isolate; use diff

Experience

- Modules tested
  - Over 50 modules of various kinds
  - Cost
- Structural coverage
  - Statement coverage necessary, *not* sufficient
  - With system testing —
    100% coverage difficult to achieve
  - With module testing —
    100% coverage easily attained
- Design for testability
  - Identify C & O problems early
  - Eliminate where possible
  - Isolate problems that cannot be eliminated
- Teaching testing
  - Scripts written by instructor
  - Scripts written by students

Conclusions

- Standardized module interfaces
  - For reusable test scaffolding
  - As the basis for automated support
- Systematic module testing
  - Trace-based test case language
  - Automatic driver generation
- Design for testability
  - Designer responsible for ensuring testability
  - Importance of controllability and observability
- Work underway
  - Industrial applications
  - Executable test oracles

REFERENCES

Scaffolding construction

[1] D.J. Panzl. Automatic software test drivers. *Computer*, 11(4):44–50, April 1978.

[2] M.M. Gorlick, C.D. Kesselman, D.A. Marotta, and D.S. Parker. Mockingbird: a logical methodology for testing. *Journal of Logic Programming (to appear)*, 1989.

[3] A. Jagota and V. Rao. TCL and TCI: a powerful language and an interpreter for writing and executing black box tests. In *Proc. Pacific Northwest Software Quality Conf.*, pages 147–166, IEEE Computer Society, 1986.

Input selection

[1] W.E. Howden. Functional program testing. *IEEE Trans. Soft. Eng.*, SE-6(2):162–169, March 1980.

[2] J.C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, September 1975.

[3] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Soft. Eng.*, SE-11(4):367–375, April 1985.

Test oracles

[1] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, July 1981.

[2] W.T. Tsai, D. Volovik, and T.F. Keefe. Automated test case generation for programs specified by relational algebra queries. *IEEE Trans. Soft. Eng.*, SE-16(3):316–324, March 1990.

Available from the speaker on request

[1] D.M. Hoffman. A CASE study in module testing. In *Proc. Conf. Software Maintenance*, pages 100–105, IEEE Computer Society, October 1989.

[2] P. Brown and D.M. Hoffman. Application of module regression testing at TRIUMF. In *Proc. Intl. Conf. Accelerators and Large Experimental Physics Control Systems*, November 1989.

[3] D.M. Hoffman and C. Brealey. Module test case generation. In *Proc. 3rd Symp. on Software Testing, Analysis, and Verification*, ACM SIGSOFT, 1989.

[4] D.M. Hoffman. On criteria for module interfaces (to appear 5/90). *IEEE Trans. Soft. Eng.*, 1990.

Paper 4-A-1

SOFTWARE QUALITY ASSURANCE OF DataEase 4.2

Mr. Timothy A. Nichol
DataEase International, Inc.

Timothy A. Nichol is the Manager of Software Quality Assurance at DataEase International. He has been with DataEase for two years. His expertise is in Integration and Structural Testing. Before joining DataEase he worked for three years at Ashton-Tate in their Developiment and SQA departments.

The DataEase Quest For Excellence

Timothy A. Nichol, Manager,
Software Quality Assurance

DataEase International

DataEase International, Trumbull Connecticut, develops and markets a full line of micro computer software products. These software products include:

- Database Management packages, Business Graphics, Image Database Management and Tools for Applications Development.

DataEase 4.2 won the 1989 PC Magazine's Editor's Choice Award for multiuser relational databases.

Reprinted from the July 1989 issue of DataBased Advisor Magazine:

> *Like most things in life, database management systems aren't static. They're continually poked, caressed, and coddled by vendors to lure more users into the fold. Some giants of the trade have stumbled trying to keep up with, or ahead of, their challengers. Other companies like DataEase and Borland, seem to forge relentlessly ahead, finding new ways to excite their followers. DataEase 4.0 is a prime example of what can be achieved, when motivated by the quest for excellence, as well as the instinct for surviving in a highly competitive industry.*

The DataEase "Quest For Excellence"

- The Team
- The Product
- Overview
- High Level Project Test Plan

Functional Testing

- Pre-Alpha
- Alpha
- Automated Testing
  - Definition
  - Capture/Playback Tools
    - AutoTester — Software Based
    - Atron — Host/Target Based
  - "Front-line" Scripts
- Lessons that we've learned:
  - Don't underestimate the script development time
  - Design them carefully
  - Don't use them too early in the cycle.
- Alpha Test Loop
- Beta Test Loop

Manual Testing

- Known Problem Areas
- Performance Tests
- Network Functionality
- Fix Confirmations
- Individual Functional Areas
- Auxiliary Software
- Network Certification (6 network minimum)

Beta Testing

- Site Selection
- Site Management
- Preliminary Sites

Structural Testing

- Critical Module Identification
- Coverage Analysis

Problem Tracking System

- Design/Implementation
- DataBase for Each Product
- Complete History of Problems Reported
- Demo of Problem Tracking System as Corporate Resource

Paper 4-A-2

VALIDATION OF PHARMACEUTICAL SOFTWARE

Mr. Don Simmons
Beckman Instruments, Inc.

Mr. Steve Herrick
Beckman Instruments, Inc.

Mr. Steven S. Herrick currently is the Director of Laboratory Automation Operations for Beckman Instruments. His division primarily produces software intensive systems supporting the fine chemicals industry including pharmaceuticals. These systems employ very high degrees of software integration. Mr. Herrick holds Master of Science degrees in both Electrical Engineering and in Computer Science. He has been developing a wide variety of software systems and software products for over 20 years.

Validation of Pharmaceutical Software

The pharmaceutical industry is regulated by the Food and Drug Administration in a number of significant ways. These regulatory requirements place special burden on software intensive system products used in the manufacture, quality assurance, and release of pharmaceutical products. Recently, these requirements have been interpreted to include explicit validation of systems by the using organization. This talk discusses major requirements and an approach to end user system validation and revalidation.

Don Simmons
Beckman Instruments, Inc.
Fullerton, CA

Steven S. Herrick
Beckman Instruments, Inc.
Allendale, NJ

VALIDATION OF AUTOMATED SYSTEMS

IN THE PHARMACEUTICAL INDUSTRY

- Regulated by the Food and Drug Administration
- Similar requirements by US EPA
- Similar requirements in Europe and Japan
- Two basic categories of legislated systems validation rigor:
  - Medical devices (clinical application)
  - Manufacturing (GMP) and research (GLP)

GOOD MANUFACTURING PRACTICES

(GMP)

- ## THE LAW

"...Assure that finished pharmaceuticals meet requirements as to safety, have appropriate identity and strength, and meet the quality and purity characteristics the drugs are represented to possess..."

-Food, Drug and Cosmetic Act
Sections 501 and 701

- ## THE REGULATION (Blue Book)

"Validation seeks to provide documented evidence that a system does what is purports to do."

- ## THE PRACTICE

"The goal of validation should be to obtain <u>high confidence</u> that the system will perform as purported."

THE PROBLEM

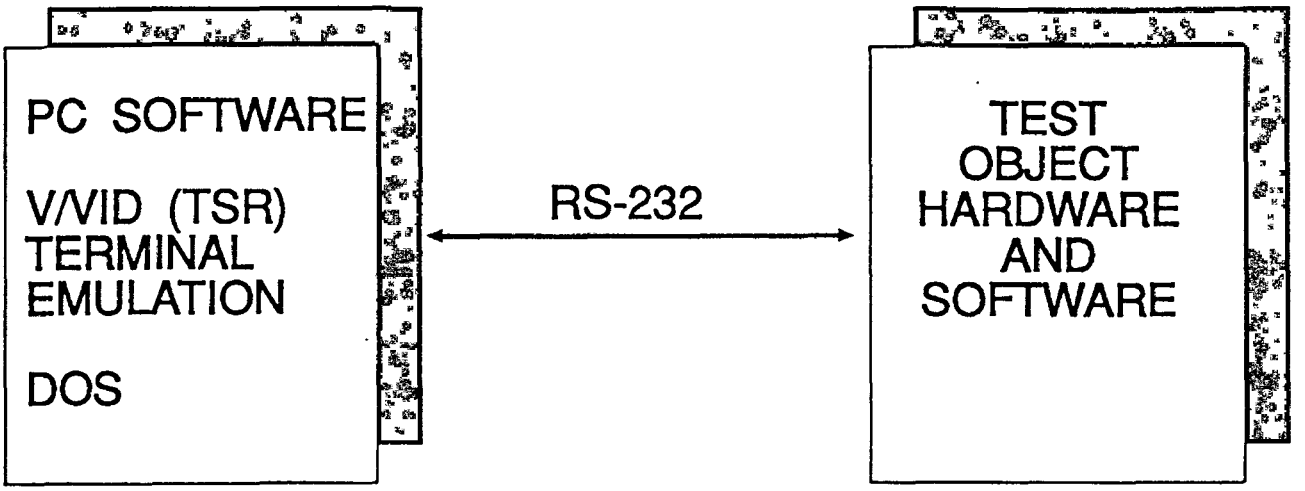
- A systems supplier to the pharmaceutical industry must provide ...
  - A sound product meeting customer needs, AND
  - A means to demonstrate and document that the product performs as required
- Sound products ...
  - Developed with product life-cycle paradigm
  - Subject to voluntary customer review
  - User training often included in "product"
- A means to demonstrate ...
  - Must be executable by user
  - Must adapt to customer's data and methods
  - Must provide minimal burden on system and user

VALIDATION AND VERIFICATION

INTERACTION DIRECTOR (V/VID)

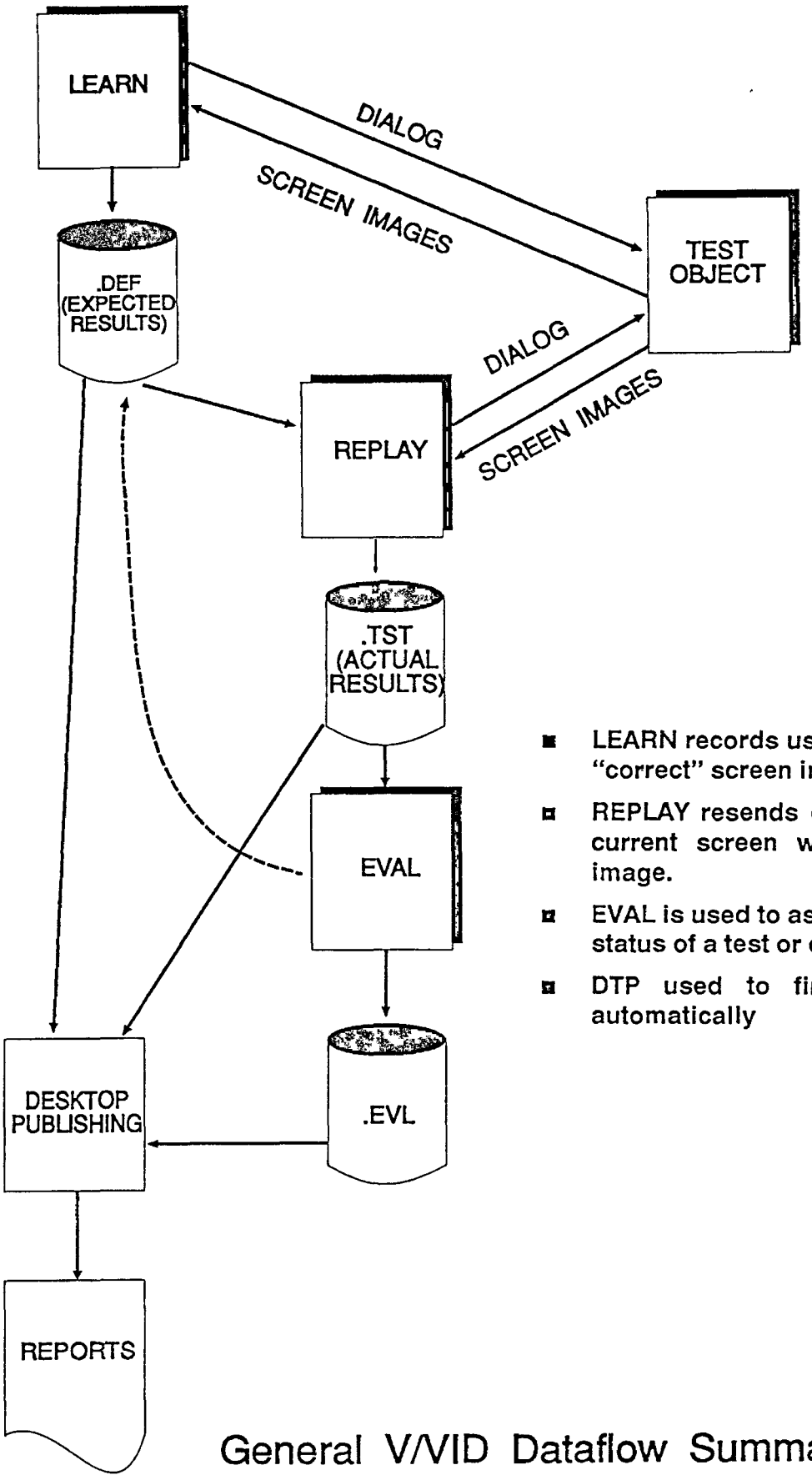
- **Learns customer task**
  - Captures keystrokes and "correct screen output"
  - Records documenting comments
  - Allows fields to be eliminated from subsequent comparison
- **Replays**
  - Keystrokes and user delays as recorded
  - Compares current screen to recorded screen (pass/fail)

- **Evaluates**
  - Each failure
  - Records comments regarding expected failure
  - Allows redefinition of new screen as master
- **Reports**
  - Validation test definitions
  - Validation run results
  - Evaluation results
  - Validation incident reports
  - Validation progress reports



General Test Bench Setup

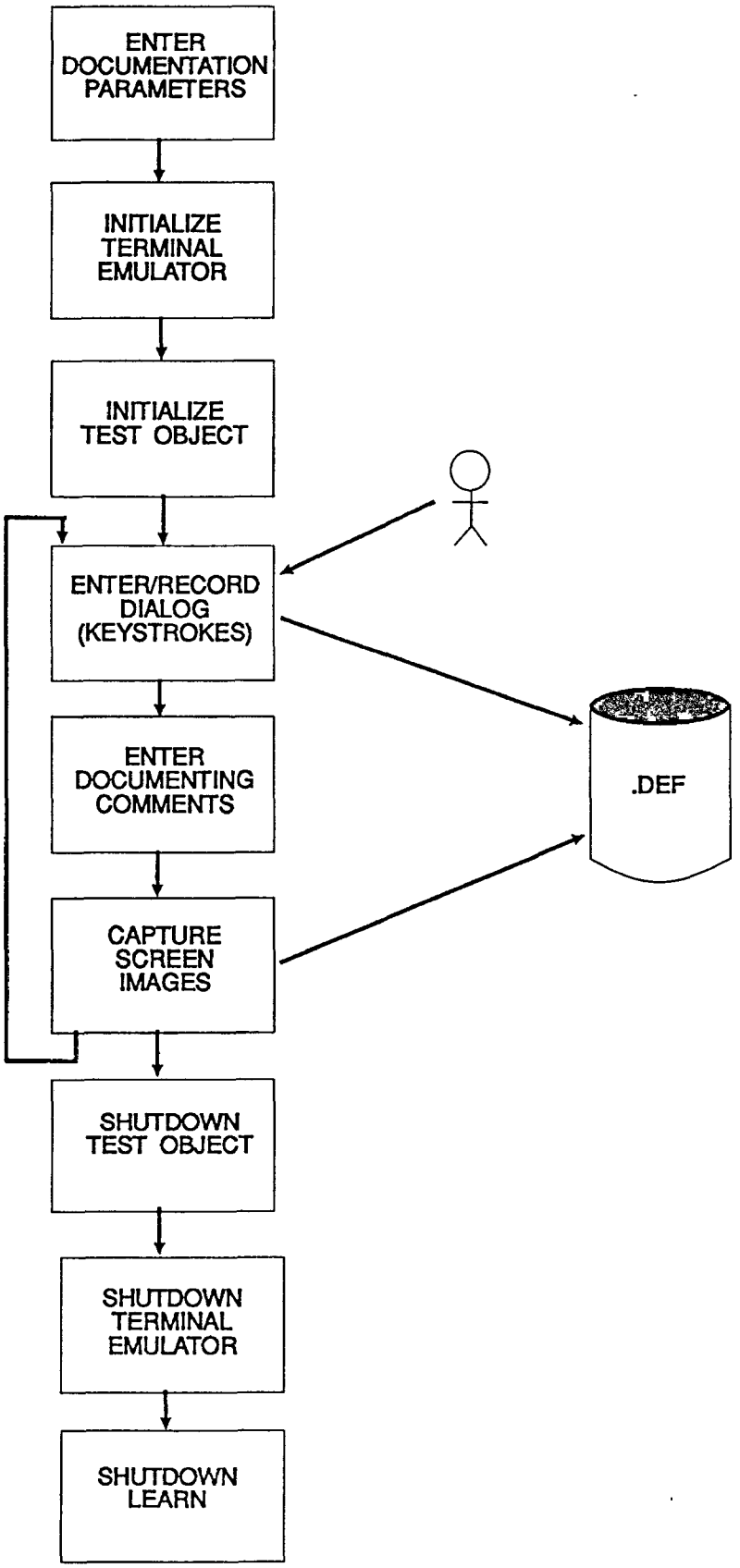
- Uses standard PC hardware and software
- Commercially available terminal emulation package (Reflection 7™ or Kermit, etc.)
- Test object is undisturbed - operates as usual without modifications for testing



- LEARN records user entered dialog and "correct" screen images.
- REPLAY resends dialog and compares current screen with recorded screen image.
- EVAL is used to assess overall pass/fail status of a test or collection
- DTP used to finish the paperwork automatically

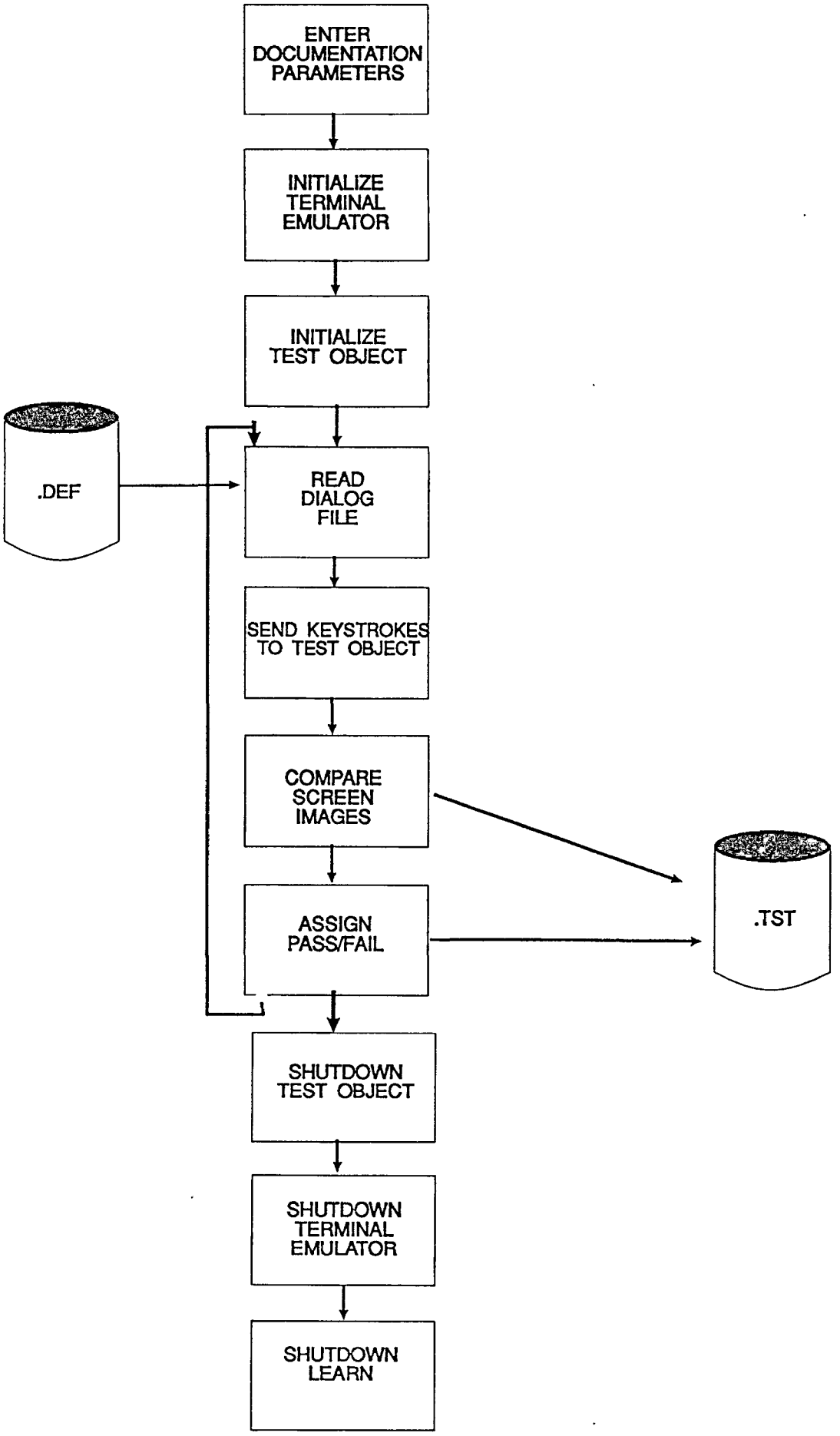
General V/VID Dataflow Summary

DONS2.cdr



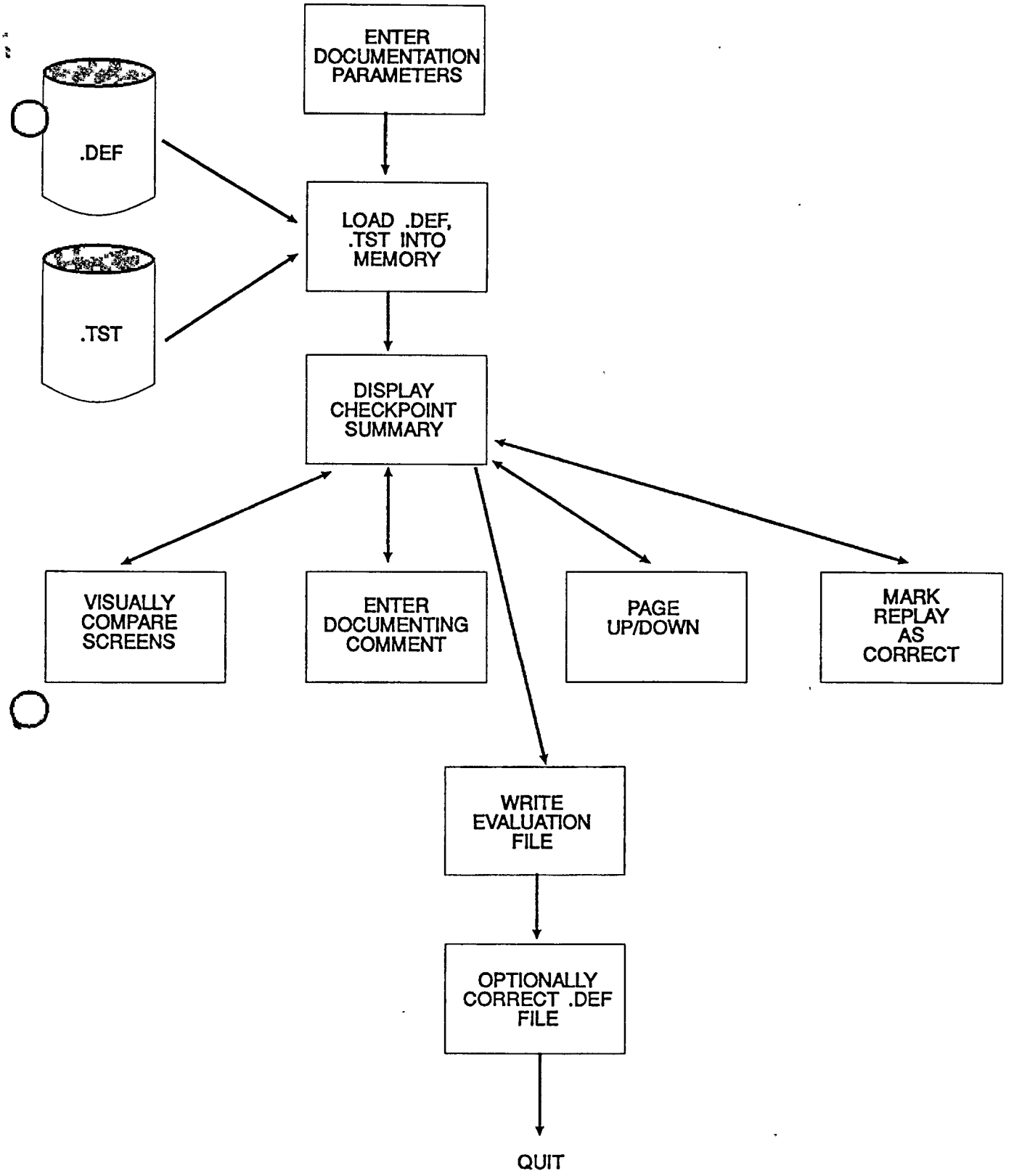
General V/VID LEARN Operations

DONS3.cdr



General V/VID REPLAY Operations

DONS4.cdr



General V/VID EVAL Operations

DONS5.cdr

SUMMARY

- V/VID has proven to be a worthwhile development tool
  - With frequent fixes and updates, automated regression testing counts
  - Testing effort can be accumulated and reused
- V/VID has proven to be usable by laboratory end users
  - Organizes the original validation and documentation task
  - Rapid, low cost revalidation
- We recommend the approach
  - With commercial products such as Software Research's CapBak™
  - Augmented for report generation and user comments.

- Medical devices are validated by manufacturer
- Manufacturing and research systems validated by user

Paper 4-A-3

PREDICTIVE RELIABILITY METRICS

Mr. Paul E. Janusz
Product Assurance Engineer
U.S. Army AMCCOM

Mr. Paul E. Janusz is an engineer in the Software Quality Assurance group of the Product Assurance Directorate, involved with research and development activities to develop tools and techniques which will assist the Independent Verification and Validation (IV & V) effort. These activities involve the development and implementation of automated tools for software development, testing, and maintenance, the incorporation of metrics to assess and quantify software preformance, and the application of artificial intelligence technology to solve common SQA problems. He recieved a B.S. in Civil Engineering fom SUNY at Buffalo in 1981 and a B.S. in Mathematics from SUC at Oneonta in 1980.

Paper 4-A-4

THE MUSA-OKOMOTO SOFTWARE RELIABILITY MODEL AS IMPLEMENTED AT TIBURON SYSTEMS

Ms. Sophia Shieh
Tiburon Systems, Inc.

Ms. Sophia Shieh has been a software quality engineer at Tiburon Systems Inc. in San Jose, California since 1987. Prior to joining Tiburon, she was a software engineer at Schlumberger Inc. where she contributed to the design, development and integration of the Schlumberger Sentry Test System for nearly ten years. She attained a B.S. Degree from National Taiwan Normal University in 1973 and a M.S.C.S. Degree from the University of Santa Clara in 1978.

Paper 4-A-4

THE MUSA-OKOMOTO SOFTWARE RELIABILITY MODEL AS IMPLEMENTED AT TIBURON SYSTEMS

Ms. Sophia Shieh
Tiburon Systems, Inc.

Ms. Sophia Shieh has been a software quality engineer at Tiburon Systems Inc. in San Jose, California since 1987. Prior to joining Tiburon, she was a software engineer at Schlumberger Inc. where she contributed to the design, development and integration of the Schlumberger Sentry Test System for nearly ten years. She attained a B.S. Degree from National Taiwan Normal University in 1973 and a M.S.C.S. Degree from the University of Santa Clara in 1978.

AN APPLICATION OF THE SOFTWARE RELIABILITY MODEL AS IMPLEMEMENTED IN INDUSTRY

May 1, 1990

Sophia Shieh
TIBURON SYSTEMS, INC.
2085 Hamilton Ave.
San Jose, CA 95125

SECTION 1

BACKGROUND

How can we predict when software is ready for release? Does the reliability of the software meet customer expectations or quality objectives? Is there a quantitative way to measure software reliability?

In addressing these issues at TIBURON SYSTEMS, INC., we researched tools and models which predict how soon in the development cycle a software program can meet its reliability requirements. From the models researched, those described by John D. Musa in IEEE Spectrum [1] were considered most suitable for our project needs. The IEEE paper [1] describes two reliability execution time models - Basic and Logarithmic Poisson (also called Musa-Okumoto Logarithmic Poisson) models.

The Musa-Okumoto Logarithmic Poisson execution-time model is based on a non-homogeneous Poisson process. Its "failure experienced" function is logarithmic with respect to execution time, rather than exponential. These reliability models have been successfully applied to projects at AT&T, Hewlett-Packard, and other military and commercial companies. AT&T has been experimenting and improving these models over the last ten years.

At TIBURON, we applied the reliability models to a communication processor which allows multiple Tactical Data Processors to share common communication equipment. This communication processor performs message formatting and reformatting, determines and controls routing of data to appropriate systems, and also allows Tactical Data Processors and communication systems with different interface protocols to exchange data.

Figures 1 illustrates the communication processor operating on different types of platforms.

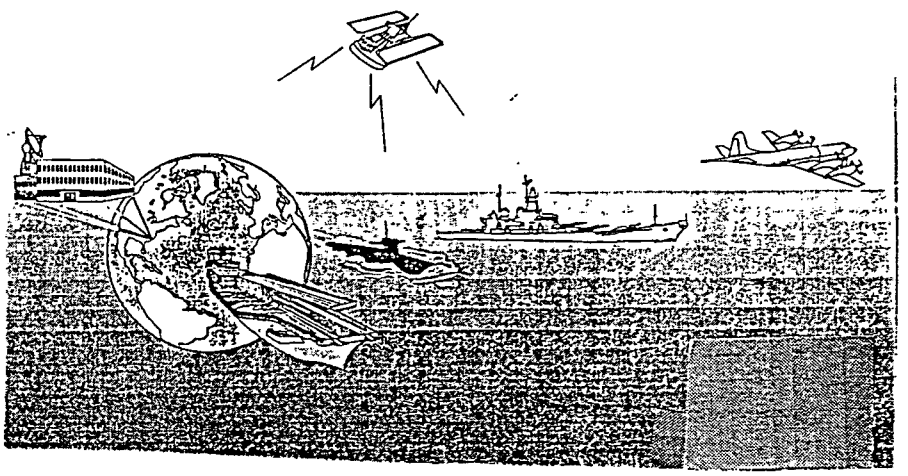


Figure 1

Figure 2 illustrates the commmunication processor operating with a satellite interface.

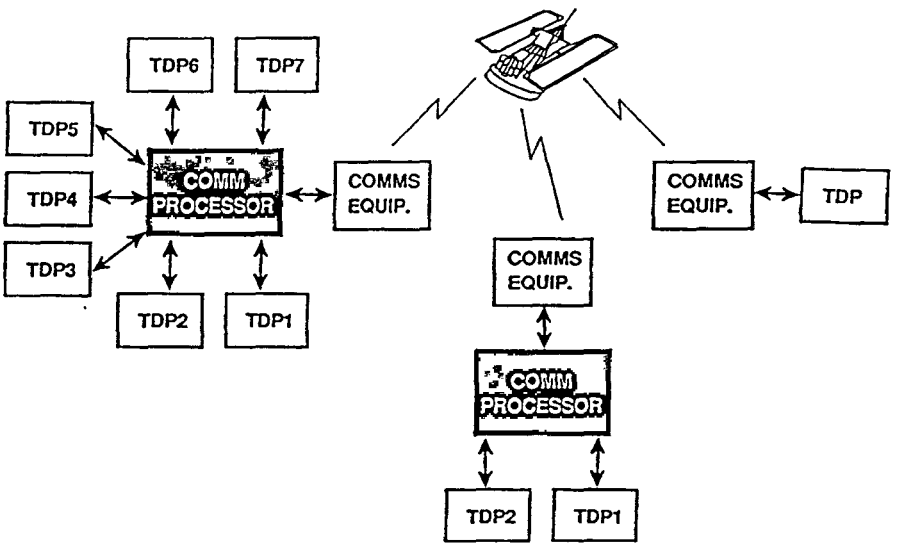


Figure 2

SECTION 2

APPLICATION

MODEL [2]

The following explains the basic terms or concepts used in discussing the software reliability models:

Failure: A departure in a computer program's operations from the user's requirement. For example, a failure may be a "crash" in the system or may be an incorrect character displayed on the monitor.

Failure intensity: The number of failures occurring in a given time period such as 1 failure per 1000 hours.

Operational profile: The set of functions a computer program is required to perform, along with their associated probabilities of occurence in field operation.

Software reliability: The probability of failure-free operation of a computer program for a specified time. For example, a failure intensity of 3 failures per 1000 hours translates into a reliability of 0.985 over five hours of execution. As far as the program is concerned, the failure intensity will be considered as constant for the duration of the release to the field with no changes in code and no repairs being made. The failure process will be considered as a homogeneous Poisson process. This implies that the failure intervals are exponentially distributed and that the number of failures in a given time period follows a Poisson distribution. If the failure intensity is λ and the period of execution time is τ, then the number of failures during the period is a Poisson distribution with parameter $\lambda\tau$.

The software reliability and constant failure intensity λ are related as follows:

$$R(\tau) = \exp(-\lambda\tau)$$

where

$R(\tau)$ = reliability
λ = failure intensity
τ = execution time

The reliability is dependent not only on the failure intensity but also on the period of execution time. The reliability (probability of no failures in a period of execution time τ) is lower for longer time periods as shown in Figure 3.

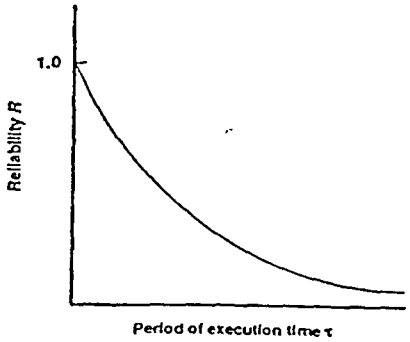


Figure 3
Reliability Versus Period
of Execution Time

The Logarithmic Poisson model assumes that failures occur as a random process. A random process whose probability distribution varies with time is called nonhomogeneous. Most of the failure characteristics during the testing process fit into this situation.

The Musa-Okumoto Logarithmic Poisson model is based on a nonhomogeneous Poisson process. The model is called logarithmic because the expected number of failures is a logarithmic function of time shown at equation (A).

The expected number of failures for the Logarithmic Poisson model is always infinite at infinite time as shown in Figure 4. This can be represented as:

$$\mu(\tau) = \frac{1}{\theta} \ln (\lambda_0 \theta \tau + 1) \qquad \text{(A)}$$

Where

μ = failures experienced (expected)
θ = failure intensity decay parameter
λ_0 = initial failure intensity
τ = execution time

Figure 5 shows the failure intensity $\lambda(\tau)$ as a function of execution time.

$$\lambda(\tau) = \frac{\lambda_0}{\lambda_0 \theta \tau + 1} \qquad \text{(B)}$$

Where

θ = failure intensity decay parameter
λ_0 = initial failure intensity
τ = execution time

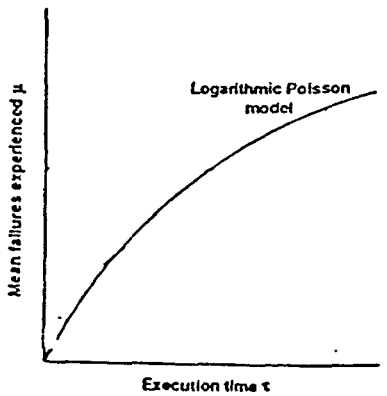


Figure 4
Mean Failures Experienced Versus Execution Time

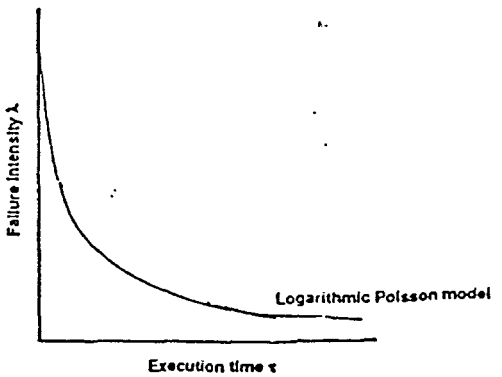


Figure 5
Failure Intensity Versus Execution Time

A highly nonuniform operational profile yields a failure intensity history that tends to be more suitably modeled by the Logarithmic Poisson execution time model. The communication project under study has highly nonuniform operational profiles, where some functions are executed much more frequently than others.

To solve the equation (B), we need to estimate:

θ = failure intensity decay parameter
λ_0 = initial failure intensity

using the Maximum Likelihood Estimation. The Newton-Raphson root-finding procedure [3] is also applied to do the point estimation of the parameters.

Additional execution time to failure intensity objective is shown as:

$$\Delta\tau = \frac{1}{\theta}\left(\frac{1}{\lambda_F} - \frac{1}{\lambda_P}\right) \qquad \text{(C)}$$

Where
$\Delta\tau$ = additional execution time
θ = failure intensity decay parameter
λ_P = present failure intensity
λ_F = failure intensity objective

Additional failures to failure intensity objective is shown as:

$$\Delta\mu = \frac{1}{\theta}\ln\left(\frac{\lambda_P}{\lambda_F}\right) \qquad \text{(D)}$$

Where
$\Delta\mu$ = additional failures (expected)
θ = failure intensity decay parameter
λ_P = present failure intensity
λ_F = failure intensity objective

From equations (C) and (D), we are able to predict the additional hours required to reach the objective and predict additional failures expected.

In addition to the Musa-Okumoto Logarithmic Poisson Model, we also apply the Reliability Demonstration approach [4] as shown in Figure 6, which is based on sequential sample theory to determine if the failure intensity objective is met with high confidence or not.

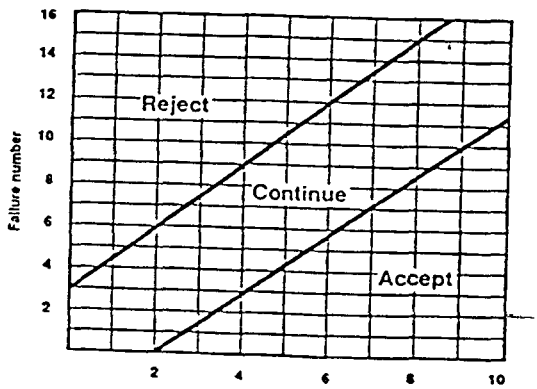


Figure 6
Reliability Demonstration
Chart

DATA AND RESULTS

In this paper, Sample A refers to an early phase of the project and Sample B refers to a subsequent phase during which time the operating system was changed.

The sample results of the application of the software reliability model to the TIBURON projects are shown in the following figures.

Figure 7 illustrates the failure intensity versus execution time in Sample A. The two peaks of high failure intensity are caused by stress test and real operational tests (site testing). The figure also shows both Basic execution time model and Logarithmic Poisson execution time model for comparison.

From Figure 7, we are able to objectively and visibly determine the current quality status of the software product under development. We also conclude that stress testing and site testing should be completed as early as possible so that the failure intensity objective can be reached sooner.

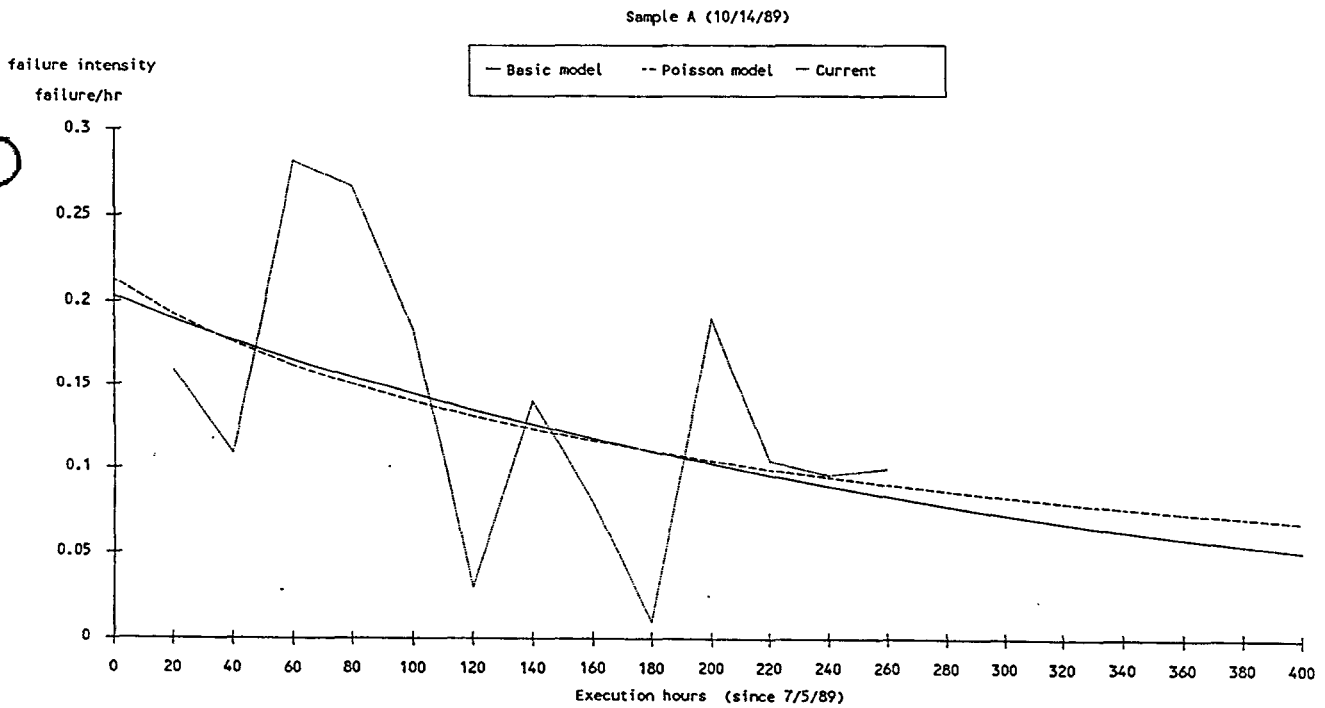


Figure 7
Failure Intensity Versus Execution Time
with Model Prediction

From the collected failure data, we are able to estimate the parameters for both Logarithmic Poisson execution time model and Basic execution time model, and get the additional failures expected and hours required for testing to reach the reliability intensity objective.

Figures 8 and 9 illustrate status report printouts from the software reliability program implemented in our company for internal use.

The sample printouts provide four different self-selected objectives and associated reliabilities for a given execution time of five hours. For reference, additional failures expected and additional hours required for testing are also included.

SOFTWARE RELIABILITY PREDICTION
BASIC EXECUTION TIME MODEL

SAMPLE A

Based on sample of 35 FAILURES
Execution time is 260.0 hours
Initial failure intensities 0.2034 failures/CPU hour
Total failures expected 59.31272
Present failure intensities 0.100 failures/CPU hour
Present Date : 10/14/89

Failure intensity objective failure / CPU hour	0.100	0.071	0.045	0.021
Additional failures will be experienced to reach failure intensity objective	0	8	16	23
The additional execution time required to reach the failure intensity objective	0.0	99.8	232.8	455.0
The Realibility for execution 5.0 hours (the probability of failure-free operation over 5.0 hours)	0.60	0.70	0.80	0.90

. Realibility=1.0 .. best
Realibility=0.0 .. worst

Figure 8

SOFTWARE RELIABILITY PREDICTION
LOGARITHMIC POISSON EXECUTION TIME MODEL

SAMPLE A

Based on sample of 35 FAILURES
Execution time is 260.0 hours
Initial failure intensities 0.2128 failures/CPU hour
Failure intensity decay parameter 0.02443
Present failure intensities 0.100 failures/CPU hour
Present Date : 10/14/89

Failure intensity objective failure / CPU hour	0.100	0.071	0.045	0.021
Additional failures will be experienced to reach failure intensity objective	0	14	33	64
The additional execution time required to reach the failure intensity objective	0.0	167.2	500.2	1539.6
The Realibility for execution 5.0 hours (the probability of failure-free operation over 5.0 hours) . Realibility=1.0 .. best---------- Realibility=0.0 .. worst	0.60	0.70	0.80	0.90

Figure 9

Based on the information shown in Figure 9, we are able to create the table shown in Figure 10 indicating possible release dates according to available resources. This table may be used to suggest increasing testing resources to reach failure intensity objective sooner or to meet the target date.

Figures 9 and 10 provide visibility into the software release process, thereby facilitating control over release.

SAMPLE A

failure intensity objective	additional execution hours expected	additional failures expected	days to reach objective **	
0.100	0	0	0 day	10/15/89
0.071	167.2	14	41 days	12/11/89
0.045	500.2	33	108 days	3/14/90

** Note : "days to reach objective" is calculated from the Calendar Time Model [2] and our company's resource usage data.

Figure 10

Figures 11, 12, 13, and 14 show representations of the failure data collected from Sample B. Figure 11 shows the failure intensity versus execution hours with goal. It provides great visibility of current quality status by comparing the current failure intensity with the goal (goal is set to 0.02 failure/hour for this sample output). Switching the software under test from one operating system to another is denoted as Period I and Period II. The model prediction is not applied to this output because of the switch. Also, the output is to be used only for comparing the quality status between the two periods.

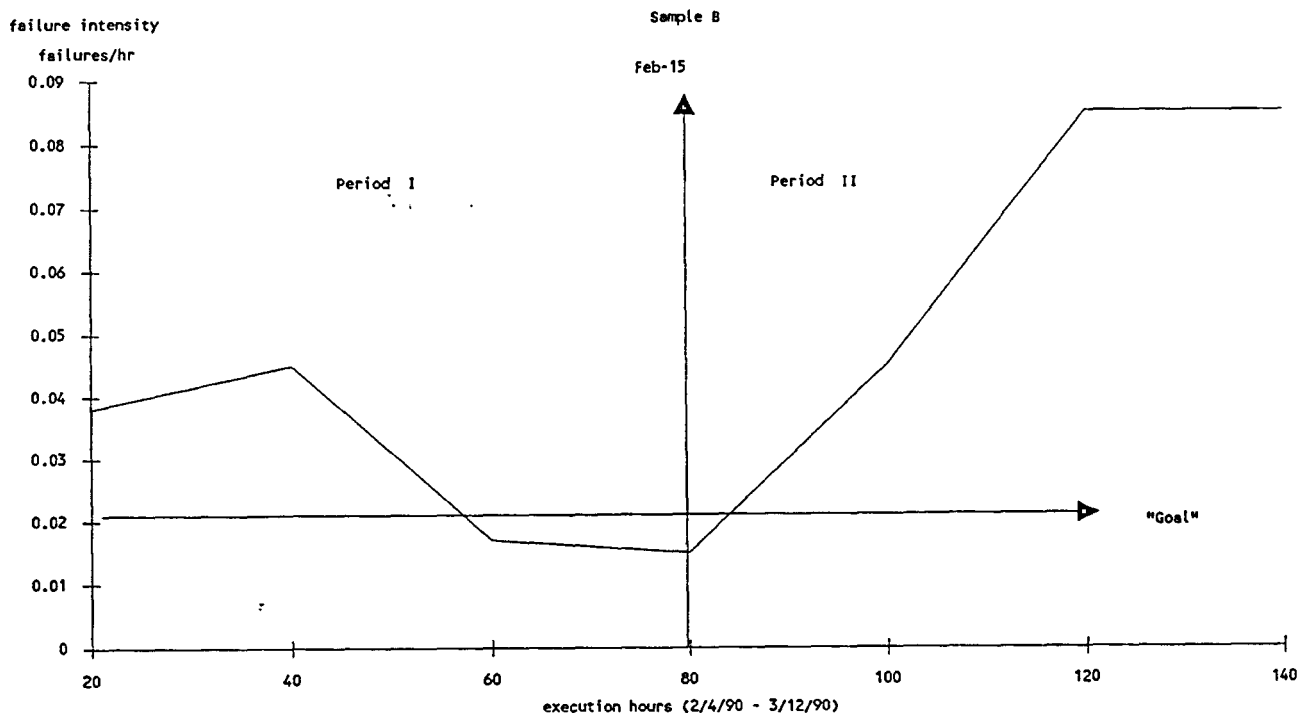


Figure 11
Failure Intensity Versus Execution Time
with Goal

Figures 12 and 13 show the testing time usage including machine setup time, system execution hours, and failure investigation hours. Figure 12 is for Period I (2/4 - 2/14) and Figure 13 is for Period II (2/15 - 3/12).

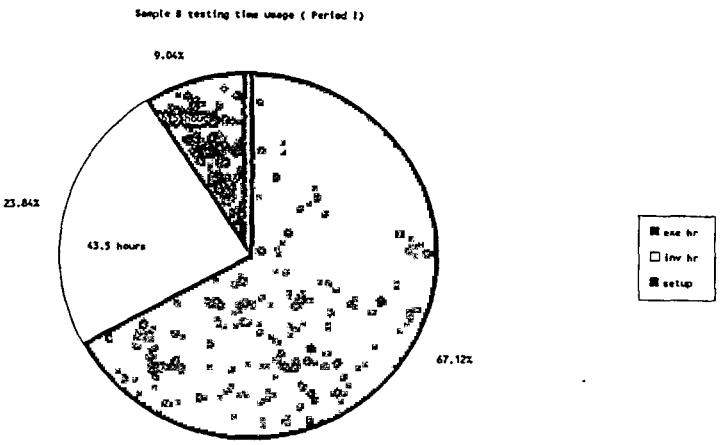


Figure 12
Testing Time Usage Percentage
for Period I

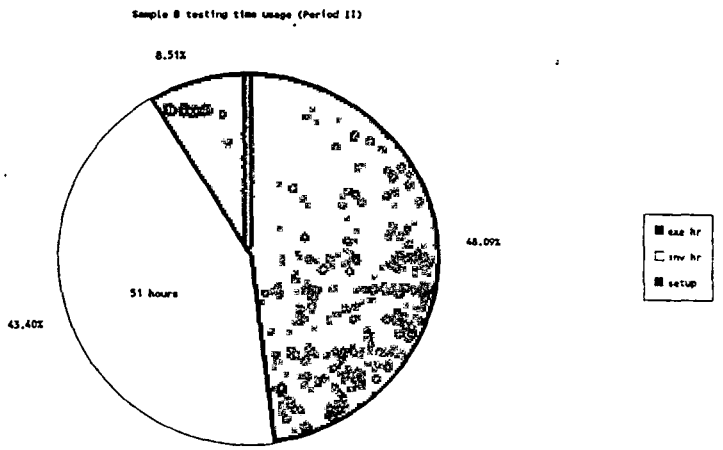


Figure 13
Testing Time Usage Percentage
for Period II

We found that by collecting the failure data, we were able to have greater visibility of the testing process. Figure 14 shows the comparison table for sample test Periods I and II.

COMPARISON:

	SAMPLE B	
	PERIOD I (2/4-2/14)	PERIOD II (2/15-3/12)
1.failure intensity	HIGH --> LOW	LOW --> HIGH
2.working days	8	18
3.testing time usage:		
a. system execution time	67.12%	48.09%
b. failure investigation time	23.84%	43.40%
c. machine setup time	9.04%	8.51%
4. efficient system execution hours	122.5 hours	56.5 hours

Figure 14
Testing Process Comparison Table for Period I and Period II

As shown in the table, more time was spent in Period II (18 working days), but less efficient system execution hours (56.5 hours) were achieved. The percentage of testing time usage of failure investigation hours in Period II is approximately twice that in Period I. From this data, we were able to make suggestions for improvements such as the need to increase resources (efficient test tools, machine availability) for testing to achieve target release dates or quality objective.

SECTION 3

CONCLUSION

With experience in measuring software reliability during the project development cycle, we are able to apply the same process to other projects in the company and as a service to outside customers.

We have found the following benefits from applying the software reliability measurement and prediction models:

- Objectively and visibly determine the current status of the quality of a software product, and help management determine the tradeoff between level of quality and delivery date (marketing windows).

- Improve field maintenance operations by predicting the expected field failures after software release. This information is useful in estimating the software product warranty cost. In addition, this information can also be used to establish software product liability risks.

- Maintain product quality standards while providing new features.

- Provide greater visibility for the software testing process and greater control over the software release process by reviewing the collected failure data and reliability prediction and making recommendations based on the results obtained.

After working with the software reliability models, we have found that the software failure intensity is affected by test engineer experience and the adequacy/availability of test tools. In addition, the customer service failure report tracking system must be adequate to ensure the accuracy of field failure prediction.

In order to use the software reliability models successfully, it is necessary to collect correct failure data. Both management and engineering must be motivated to support data collection efforts. This can only be accomplished if management and engineering are educated in the software reliability measurement concept.

At this point, we are still working on this process to validate the predictive capabilities of these models. However, we feel that having quantitative information of software reliability has significant benefits in managing the software release process.

SECTION 4

REFERENCES

1. John D. Musa, "Tools for Measuring Software Reliability", IEEE SPECTRUM, February 1989

2. John D. Musa, Anthony Iannino, and Kazuhira Okumoto, "Software Reliability: Measurement, Prediction, Application" published by McGraw-Hill, 1987

3. Carnahan, B., and J. O. Wilkes, 1973. "Digital Computing and Numerical Methods", Wiley, New York

4. Grant, E. L., and R. S. Leavenworth, 1980. "Statistical Quality Control", McGraw-Hill, New York

Paper 5-T-1

DESIGN AND CODE TRACEABILITY IN PDL: QUALITY CONTROL ASPECTS

Dr. Paul Oman
University of Idaho

Dr. Paul W. Oman is an assistant professor of computer science with the college of engineering at the University of Idaho. His research interests include software complexity metrics and human factors analysis of programmer behavior. Oman is the Software Test Lab Editor for IEEE Software magazine and has published several articles on software tools and programming. Oman recieved the Ph.D. in computer science from Oregon State University in 1989. He is a member of ACM, IEEE, IEEE Computer Society, and Phi Kappa Phi.

DESIGN AND CODE TRACEABILITY IN PDL: QUALITY CONTROL ASPECTS

Dr. Paul W. Oman
Computer Science Department
College of Engineering
University of Idaho
Moscow, ID 83843
208-885-6589
oman@ted.cs.uidaho.edu

Abstract

The importance of early defect removal and the use of software complexity metrics have been established in many empirical studies. It is many times more costly to remove an error late in the software life cycle than early in the software life cycle. Software complexity metrics have been shown to be useful in identifying error-prone or difficult to test modules. However, the metrics are usually derived from the actual program code -- late in the software life cycle. In this paper we demonstrate the ability to extract complexity metrics from Program Design Language (PDL), as contained in design specification documents, and show the utility of these metrics in providing a mechanism for identifying trouble spots, measuring consistency within and across modules of a software system, and for assessing traceability between PDL and source code.

INTRODUCTION

The importance of early defect removal and the utilization of software complexity metrics have long been established in numerous software studies [Basi84, Boeh76, Boeh84, Jone79, Parn79, Pfle87]. It is many times more costly to remove an error late in the software life cycle than early in the software life cycle. Software complexity metrics have been shown to be useful in identifying error-prone or difficult to test modules [Grem84, Shen85, Taka85]. However, these metrics are derived from the actual program code -- late in the software life cycle.

Several extensions of complexity metrics for software designs have been proposed and recent studies have described the use of design-time metrics to aid software development and maintenance:

- Szulewski, et. al., [Szul81] defined Halstead's metrics for the parts of a graphical design representation.
- Troy and Zweben [Troy81] defined 21 metrics measuring the complexity of structure charts.
- Hall and Preiser [Hall84] applied McCabe's cyclomatic complexity to hierarchically structured design graphs.
- McCabe and Butler [McCa89] extended the definition of cyclomatic complexity by defining three levels of complexity: (1) module complexity, the cyclomatic complexity of a reduced graph, (2) design complexity, the sum of the module complexities, and (3) integration complexity, the design complexity minus the number of modules plus one.
- Brandl [Bran90] described applying structure chart metrics to Hughes Aircraft Company designs.
- Henry and Selig [Henr90] found correlations between the structural complexity of Ada-like design descriptions and the corresponding source code complexity.
- Rombach [Romb90] found correlations between structural complexity measures of high level designs and the cost of changes during maintenance.

All of these applications of complexity metrics to software designs show promise. But further empirical tests are necessary to determine how the development process can be influenced through the collection and assessment of complexity measures.

The difficulty in assessing design complexity is partially due to the creativity of the design process. Software designs range from informal English descriptions to formal graphic notations. Hence, design metrics must be tailored to a specific design method and notation. In this paper we have restricted

ourselves to extending complexity metrics to pseudocode design descriptions. We show the utility of these metrics in providing a mechanism for measuring complexity and consistency within and across modules and for assessing traceability between PDL and source code. Design complexity metrics give the software developer a mechanism for targeting unusually complex parts of a system design. When used in conjuction with code metrics, they provide a way to identify inconsistencies between design specification and code implementation.

We implemented a prototype PDL metric extractor that analyzes pseudocode from design specification documents and computes simple measures of PDL complexity. Specifically, Halstead's software science metrics [Hals77], the number of lines of pseudocode, and the number of tokens are calculated on a module by module basis. When used in conjunction with a software tool for identifying metric "outliers," these measures provide a basis for:

1. Rank ordering the complexity of module designs.

2. Identifying inadequate PDL descriptions of modules.

3. Identifying inconsistencies within the design document.

4. Targeting potential problems prior to coding.

When used in conjunction with source code complexity metrics obtained after coding, the comparison of PDL metrics to code metrics provides a verification mechanism to check for consistency and traceability.

USING COMPLEXITY METRICS

Complexity metrics are objective measures of how complex a piece of code is and how difficult it may be for a programmer to test, maintain, or understand [Cook84]. Software complexity metrics do not measure the complexity itself, but instead measure the degree to which those characteristics thought to contribute to complexity exist within the software.

Numerous studies have demonstrated the utility of complexity metrics. Experiments have shown a strong relation between programs with high complexity metric values and the difficulty of performing programming tasks such as program comprehension, debugging, and maintenance [Bern84, Elsh84, Gord79, Kafu85]. Complexity metrics have been used to identify error-prone program modules and have provided reasonable predictions of the number of errors in modules [Grem84, Shen85, Taka85].

Complexity metrics can aid in the allocation of resources for testing and maintenance [Harr86]. Modules with high metric values are likely to be more difficult to test and maintain. They contain most of the errors and, hence, should be allocated

more resources. Program errors are the major cost associated with software development and the cost of finding and correcting errors is related to the software life cycle phase in which the error is found. It is over ten times as expensive to find and fix an error discovered during the test phase as during the design phase. Although errors can be introduced in any phase of the life cycle, most (well over 50 percent) are introduced during the design stage [Basi84, Zelk79].

Since complexity metrics have been shown to identify modules likely to be error-prone, it is appropriate to extend complexity metrics to program design documents for two reasons: (1) most errors are introduced in the analysis and design phases, and (2) the cost of finding and correcting an error increases throughout the software life-cycle. The objective is to identify sections of a program's design that are likely to be hard to test and to contain errors.

In this paper we demonstrate the ability to extract complexity metrics from Program Design Language (PDL), as contained in design specification documents. The PDL metrics tool analyzes pseudocode and computes simple token-count metrics such as Halstead's Software Science measures and the number of lines of pseudocode. The tool allows the user to define operands and operators or to use the program's defined values. Reynolds [Reyn87] developed a similar pseudocode complexity metrics analyzer but his system uses a database and reasoning system to classify operators and operands. Ours uses external state tables and binary search trees to classify operators and operands.

A PROTOTYPE PDL METRICS EXTRACTOR

Our prototype PDL Metrics Extractor is an experimental tool to assist us in determining what metrics can be calculated from pseudocode descriptions, and how those metrics can be used to influence software development. The primary design consideration in building the tool was that it had to be able to process any and all pseudocode descriptions. This demand for flexibility excluded the use of syntax directed parsing and restricted the tool to simple token scanning and analysis.

The metrics extractor is a window oriented program for PC compatible microcomputers. It uses an external state table to drive the scanning of the input file containing pseudocode. Tokens extracted from the pseudocode are identified as being either Operators, Operands, Skipped (for extraneous prepositions), or control tokens for directing the calculation of the metrics. Two modes of token identification are possible:

1. Automatic: Automatically classifies all tokens.

2. Query: Asks the user for direction whenever the current token has not been previously classified.

External default lists of operators, operands, and tokens to be skipped are used to initialize binary search trees which guide the token classification. These binary search trees are updated with every token and token counts are then used to compute the metrics. The external state table and external token lists permit the metric extractor to be tailored to specific applications and programming languages. The default lists are based on Salt's counting strategies [Salt82], which were later adopted by Conte, Dunsmore, and Shen [Cont86]. Thus, when using its default tables, the metric extractor is capable of processing any text file, but is most accurate with Pascal-like pseudocode.

Figure 1(a) shows the metric extractor in Query mode. The tool is querying the user for instructions on how to process the "4.15" token. The input pseudocode scrolls through the upper window while classified tokens are added to the three scrolling windows for operators, operands, and skipped tokens. Queries to the user appear in the middle and bottom lines of the display. Figure 1(b) shows the metrics extractor upon reaching the end of a module. The binary search trees are scanned and dumped into the three scrolling windows at the bottom of the display. The frequency of occurrence is listed next to each token.

Output from the metrics extractor includes: (1) a screen display of the total number of lines of pseudocode processed (LOC), the total number of tokens processed (TOK), and totals for Halstead's four baseline metrics (n1, N1, n2 and N2), and (2) an output file containing these metrics on a line by line basis for each module processed by the tool. This format is consistent with the output from our source code metrics extractor (designed and implemented for earlier studies). Table 1 shows an example PDL metrics output file; Table 2 shows the corresponding source code metrics as calculated by our source code metrics extractor. Both files are compatible with our metric outlier identification program (described in the next section) and commonly used statistical packages.

USING PDL METRICS FOR QUALITY CONTROL

Initially, we tested the PDL metrics extractor on small pseudocode samples taken from published sources [Nann85, Rohl83, Shel86]. These tests showed the metrics extractor was reliable and useful for calculating token-based metrics such as Halstead's Software Science measures. But harder to calculate structure metrics, like Henry and Kufura's information flow [Kafu85], could not be approximated reliably. For the token-based metrics, we found a high correlation between those extracted from the pseudocode and those calculated from the corresponding source code. Figure 2(a) shows the metric extractor processing pseudocode from [Rohl83], while Figure 2(b) shows it processing the corresponding source code. But in these tests the input was limited to the "polished" pseudocode typically found in academic texts. We wanted to test the relationship between real-world pseudocode designs and the code generated from them.

For a more rigorous and practical test, we applied the metrics extractor to ongoing software engineering projects at the University of Idaho [Oman86]. The projects used a specifying approach to software development using abridged versions of the IEEE standards for requirements specifications (IEEE Std. 830-1984) and design specifications (IEEE Std. 1016-1987) [IEEE87].

We extracted PDL metrics from the pseudocode taken from the design documents of three group projects: (1) A Bezier curve fitting program, (2) A materials and resource planning system, and (3) An interactive DFA simulator. In all cases, the metric analysis was conducted unbeknown to the students working on the project. This was done deliberately to avoid interfering with the ongoing software development and to exclude Hawthorne effects (where the subjects' behavior changes because they know they're being studied). All three projects were implemented successfully (each exceeding 3000 lines of code) and passed the customer's implementation sign-off. The data shown in Tables 1 and 2 are from the DFA simulator project.

The PDL metrics were analyzed statistically to determine intrasystem characteristics, and they were compared to metrics calculated from the final source code. Several characteristics relative to software quality control were discovered:

1. PDL metrics follow many of the same statistical patterns exhibited in code metrics. For instance, there are high correlations ($r>.75$) between size related metrics such as LOC, Halstead's Vocabulary (V), Halstead's Volume (N) and estimated Volume (N^).

2. Metrics for detailed pseudocode modules correlated highly with corresponding source code metrics ($r>.8$), while metrics for loosely written pseudocode showed almost no correlation to the corresponding code ($r<.3$).

3. Linear regression models predicting source code complexity from PDL metrics could not be reliably constructed if intrasystem variation was high due to inconsistently written pseudocode. (However, subsequent tests have shown that it is possible to construct predictive models when pseudocode design standards are enforced.)

4. PDL metrics could be used to rank order modules by design complexity. However, this rank ordering did not always follow the complexity ordering of source code modules. Differences can be attributed to inconsistencies in the individual team members design and coding styles, and the incomplete mapping between specified design and actual code.

5. Unusual PDL module descriptions could be targeted by identifying metric outliers (extreme values) in precisely the same manner as is done for source code metrics. These PDL outliers frequently corresponded to source code outliers, so that modules flagged as having unusual

pseudocode descriptions were also unusual in their source code characteristics.

All of these findings have implications on software quality, but the last result is especially promising because it provides a mechanism for identifying unusual, possibly error-prone designs. Outliers are extreme values of some characteristic under study. They point to anomalies in the data that require further investigation. For instance, a module with an extremely high cyclomatic complexity, V(g) > 50, may be indicative of a complicated multiway branching structure or a simple CASE statement. Its true complexity (relative to program comprehension) cannot be determined without actually viewing the suspect code. The outlier value is the flag indicating that additional analysis is warranted. Outliers are usually computed as being one or two standard deviations away from the average (mean) value of a normal data distribution.

Table 3 shows the output from our metric-outlier program as it processed the data from Table 1. Likewise, Table 4 shows the metric-outliers for the source code metrics in Table 2. As shown in these tables, the metric-outlier program calculates average metric values for each column of metrics and then prints, on a module-by-module basis, flags indicating where metric outliers exist. For instance, the value "+2+" indicates the value for a metric (column) of a module (row) is greater than two, but less than three, standard deviations above the average.

Using the calculation of metric outliers we were able to identify anomalies in the PDL. Outliers with extremely low values pointed to pseudocode that was either too vague or described small simple modules. On the other hand, outliers with extremely high values pointed to pseudocode that was either too detailed or highly complex. In this fashion we were able to identify "trouble spots" in the PDL. For example, in the Bezier curve fitting project (data not shown) we found:

- PDL outliers: Load, Profile, and Sort
- Code outliers: Change, Profile, and Sort

Further examination showed:

- Profile and Sort were overly complex in both design and implementations and should have been simplified at design time.
- Load and Change had been written by the same programmer and in both cases the pseudocode was vaque and inadequate.
- The implementation of Load proceeded within normal bounds (i.e., the implementation overcame the design inadequacy).
- The implementation of Change contained many instances of redundant code causing abnormally high code metric values.

We obtained similar results from our analyses of the other two projects. For instance, on the DFA simulator project (data shown in Tables 3 and 4) we found:

- PDL outliers: MoveCursor, DrawEdge, LoadData, and BezierCurve
- Code outliers: MoveCursor, DrawEdge, PrintStateTable, and Simulate

Subsequent analysis showed:

- MoveCursor and DrawEdge are overly complex modules in both pseudocode and code.
- The pseudocode complexity for Loaddata was overstated (its implementation proceeded within normal bounds).
- The coding process dispersed the complexity of BezierCurve into other code modules.
- PrintStateTable is not overly complex, but its printing conditions inflate the V(g) metric.
- The code for Simulate is quite complex and the pseudocode description does not reflect the true nature of the module.

For our predictive system, MoveCursor and DrawEdge are examples of true hits: Overly complex modules in both pseudocode and code. LoadData and BezierCurve are examples of false hits: Targeting trouble spots that don't exist. Their pseudocode showed excessive complexity, but the source code showed no unusual characteristics. PrintStateTable is a false miss: It appeared as though it should have been targeted, but subsequent analysis shows that not to be true. Simulate, on the other hand, is an example of a clear miss: Not identifying a trouble spot. These categories are typical of predictive systems such as ours.

CONCLUSIONS

We have used a prototype PDL metrics extractor to gauge the adequacy and consistency of PDL pseudocode designs. We tested the metrics extractor on design documents from published sources and from systems being developed in a software engineering lab at the University of Idaho. We extracted the PDL complexity metrics and then conducted comparative analysis with respect to intramodule and intermodule relationships and with respect to metrics derived from the corresponding source code.

Intramodule analysis suggests that PDL metrics follow the known patterns of code metrics; i.e., the relationships observed and documented across code metrics from a single module are also found in PDL metrics from a single module. Deviations from these

known patterns can be used to identify internal inconsistency.

Intermodule relationships are useful in isolating PDL descriptions that are inconsistent with respect to the other modules within the design specification; i.e., identification of metric values that are statistical outliers with respect to the average metric value for all modules within the system. In this study we were able to:

1. Identify PDL descriptions that were inadequate because of lack of detail.

2. Identify PDL descriptions that were too detailed.

3. Identify inconsistent PDL descriptions.

4. Demonstrate a high degree of traceability between detailed PDL descriptions and corresponding source code.

5. Demonstrate a low degree of traceability between inadequate PDL descriptions and corresponding source code.

The metrics extractor is flexible, but inherently primative because of its context-free classification of tokens. Further, the metrics extractor cannot process entire design specifications documents (such as those outlined in the IEEE 1016-1987 Standard); the pseudocode must be extracted from the design specification prior to processing. In order to achieve the robustness and strength necessary for industrial applications, design metrics extractors should incorporate:

1. Context sensitive classification of tokens.

2. Guided processing of whole specification documents.

3. Automatic post-processing identification of metric outliers and module rank-orders.

With these improvements we believe that PDL metrics can be utilized in manners similar to code complexity metrics; namely, identifying error-prone designs and rank ordering module complexity for purposes of resource allocation. Studies are presently underway to determine how PDL metrics can be used for early fault identification and defect removal.

```
                    Pseudo  Code  Metrics  Extractor
Pseudo Code
                 set xtemp, ytemp, and ztemp = translated and scaled x,y,
                      and z values
                 Bezsurf (limitvector, xtemp, ytemp, ztemp, nview)
                 interrupt #5
                 Screen_setup
                 Bezsurf (limitvector,x, y, z, nview)
            End
        4.15 Description of Module SORT

New Token -->4.15<--.......ACTION REQUIRED (see below).
Extracted Operators        Extracted Operands        Skipped Tokens
interrupt                  xtemp                     the
Screen_setup               ytemp                     updated
Bezsurf                    ztemp                     of
(                          nview                     the
,                          5                         Procedure
,                          limitvector               of
,                          x                         var
,                          y                         of
)                          z                         #
End                        nview

Menu: 1 Dtor, 2 Dand, 3 Skip, 4 Skip eoln, 5 End Module, x skip to "x" ?
```

Figure 1(a). Query Mode.

```
                    Pseudo  Code  Metrics  Extractor
Pseudo Code
                         x[j] = x[j+1]
                         y[j] = y[j+1]
                         x[j+1] = hold
                         y[j+1] = hold2
                       end
                     pass = pass + 1
                   end
                 end.

Extracted Operators        Extracted Operands        Skipped Tokens
UNTIL 2                    TWO 1                     TYPE 0
WHILE 1                    VALUES 1                  UPDATED 1
WITH 0                     X 15                      USE 0
WRITE 0                    XTEMP 2                   USED 0
WRITELN 0                  Y 13                      USER 0
XOR 0                      YES 0                     VAR 3
[ 10                       YTEMP 2                   WHAT 0
] 10                       Z 9                       WHICH 0
^ 0                        ZTEMP 2                   WILL 0
-----------------------   -----------------------   -----------------------

Enter module name --> done
```

Figure 1(b). At the End of a Module.

Figure 1. Screen Displays of the PDL Metrics Extractor.

```
                    Pseudo  Code  Metrics  Extractor
Pseudo Code
   else
      begin
      calculate the date of Easter;
      write out the date
      end
   end;
write out the title
end.

Extracted Operators        Extracted Operands        Skipped Tokens
write                      a
else                       suitable
begin                      message
;                          calculate
write                      date
end                        Easter                    the
end                        out                       the
;                          date                      of
write                      out                       the
end                        title                     the

Enter module name --> easter
```

Figure 2(a). Processing Pseudocode.

```
                    Pseudo  Code  Metrics  Extractor
Pseudo Code
         writeln(' ':6,Easter:2,'April':6)
       else
         writeln(' ':6,Easter+31:2,'March':6)
       end
   end;
writeln(' ':4,'Some Easter Dates');
writeln(' ':4,'================')
end.

Extracted Operators        Extracted Operands        Skipped Tokens
:                          31
,                          2
)                          'March'
;                          6
writeln                    ''
(                          4
:                          'SomeEasterDates'
,                          ''
)                          4                         var
end                        '================'        integer

Enter module name --> easter
```

Figure 2(b). Processing Source Code.

Figure 2. Processing Pseudocode and Code.

Table 1. PDL Metrics Extractor Output File.

	name	tok	loc	n1	N1	n2	N2
1	Init	21	10	3	7	11	14
2	DrawMenu	89	22	3	15	22	74
3	Control	43	16	7	16	24	26
4	MoveCursor	124	25	7	24	33	99
5	LoadData	190	39	17	96	33	92
6	SaveData	79	17	13	40	21	38
7	PrintData	16	10	6	6	7	9
8	Simulate	37	13	9	17	11	19
9	DrawEdge	131	45	16	63	31	67
10	ClearScree	35	15	9	14	12	20
11	DrawNode	67	22	15	29	21	38
12	DeleteNode	81	22	18	38	23	43
13	DeleteEdge	37	15	10	17	14	20
14	QuitRGS	32	14	9	16	8	16
15	PrintTable	95	26	8	60	12	34
16	PrintSimul	47	9	12	17	18	24
17	GetNextCha	6	4	3	3	3	3
18	CalcPath	24	8	6	11	11	12
19	SimPath	52	10	9	18	12	32
20	BuildNodeT	15	4	4	7	4	8
21	UpdateNode	40	7	10	21	11	19
22	LabelEdge	14	10	4	5	7	9
23	BuildEdgeT	118	16	15	50	29	64
24	UpdateEdge	65	14	13	30	21	33
25	BezierCurv	153	29	15	76	28	77
26	UpdateStat	25	5	9	10	10	13
27	CalcFact	20	6	7	10	5	10
28	DrawCursor	5	4	1	1	3	3
29	CheckCurso	20	5	6	8	7	12
30	StatusLine	6	4	1	1	5	5
31	BuildState	26	4	8	9	13	15

Table 2. Code Metrics Extractor Output File.

	name	dsl	loc	tok	com	Vg	Nst	n1	N1	n2	N2
1	CHECK_CU	6	5	47	18	1	0	7	16	8	14
2	DRAW_CUR	45	42	390	19	16	5	14	176	22	151
3	MOVE_CUR	276	273	1553	46	37	28	21	781	74	531
4	CALC_FAC	10	6	43	16	2	1	6	12	4	9
5	BEZIER_C	48	41	396	23	5	2	19	156	37	116
6	LABEL_ED	131	122	981	46	16	10	26	433	64	349
7	BUILD_ED	22	19	112	24	2	1	5	34	30	30
8	BUILD_ST	23	18	231	19	8	5	19	92	21	60
9	ATAN2	24	19	133	15	16	7	16	55	6	40
10	DRAWARRO	49	38	360	37	12	3	23	146	35	108
11	DRAW_EDG	309	291	2165	86	58	16	37	1043	82	720
12	UPDATE_E	20	19	76	16	2	1	5	34	18	30
13	UPDATE_S	10	7	65	16	3	2	10	21	11	19
14	DELETE_E	112	104	806	39	20	8	30	361	67	282
15	BUILD_NO	5	4	29	14	1	0	5	10	4	8
16	DRAW_NOD	82	75	539	35	15	8	33	261	40	161
17	UPDATE_N	5	4	29	15	1	0	5	10	5	8
18	DELETE_N	77	70	534	35	18	8	34	258	44	160
19	PRINT_ST	152	140	1014	18	52	13	18	474	36	294
20	PRINT_SI	26	24	211	15	8	4	23	95	23	71
21	PRINT_DA	23	21	143	15	6	2	22	70	17	39
22	STATUSLI	9	8	71	18	1	0	4	24	17	22
23	INITIALI	47	42	231	23	6	2	12	101	53	92
24	LOAD_DAT	150	137	1014	39	28	10	38	444	76	360
25	SAVE_DAT	58	51	388	30	13	6	29	172	50	134
26	GET_NEXT	16	15	97	16	2	1	15	47	19	31
27	CALCULAT	9	6	53	18	3	1	11	16	8	11
28	SIM_PATH	17	16	106	19	4	1	9	41	12	33
29	SIMULATE	237	221	1501	65	53	11	36	738	68	471
30	QUIT_RGS	34	31	194	20	8	3	23	102	26	54
31	CLEAR_SC	35	32	203	22	8	3	24	108	25	56
32	DRAW_MEN	87	45	850	35	1	0	5	142	41	140
33	CONTROL_	22	19	83	26	3	2	20	47	17	21
34	RGS	73	12	344	38	2	1	11	27	11	15

Table 3. PDL Metrics Outlier Table.

	name	tok	loc	n1	N1	n2	N2
1	Init	.	.	-1-	.	.	.
2	DrawMenu	.	.	-1-	.	.	+1+
3	Control	.	.	.	.	.	.
4	MoveCursor	+1+	+1+	.	.	+1+	+2+
5	LoadData	+2+	+2+	+1+	+3+	+1+	+2+
6	SaveData	.	.	.	.	.	.
7	PrintData	.	.	.	.	.	.
8	Simulate	.	.	.	.	.	.
9	DrawEdge	+1+	+2+	+1+	+1+	+1+	+1+
10	ClearScree	.	.	.	.	.	.
11	DrawNode	.	.	+1+	.	.	.
12	DeleteNode	.	.	+1+	.	.	.
13	DeleteEdge	.	.	.	.	.	.
14	QuitRGS	.	.	.	.	.	.
15	PrintTable	.	+1+	.	+1+	.	.
16	PrintSimul	.	.	.	.	.	.
17	GetNextCha	-1-	-1-	-1-	.	-1-	-1-
18	CalcPath	.	.	.	.	.	.
19	SimPath	.	.	.	.	.	.
20	BuildNodeT	.	-1-	-1-	.	-1-	.
21	UpdateNode	.	.	.	.	.	.
22	LabelEdge	.	.	-1-	.	.	.
23	BuildEdgeT	+1+	.	+1+	+1+	+1+	+1+
24	UpdateEdge	.	.	.	.	.	.
25	BezierCurv	+2+	+1+	+1+	+2+	+1+	+1+
26	UpdateStat	.	.	.	.	.	.
27	CalcFact	.	.	.	.	-1-	.
28	DrawCursor	-1-	-1-	-1-	.	-1-	-1-
29	CheckCurso	.	.	.	.	.	.
30	StatusLine	-1-	-1-	-1-	.	-1-	.
31	BuildState	.	-1-	.	.	.	.

Table 4. Code Metrics Outlier Table.

	name	dsl	loc	tok	com	Vg	Nst	n1	N1	n2	N2
1	CHECK_CU	.	.	.	.	.	.	-1-	.	-1-	.
2	DRAW_CUR	.	.	.	.	.	.	.	.	.	.
3	MOVE_CUR	+2+	+2+	+2+	+1+	+1+	+3+	.	+2+	+1+	+2+
4	CALC_FAC	.	.	.	.	.	.	-1-	.	-1-	.
5	BEZIER_C	.	.	.	.	.	.	.	.	.	.
6	LABEL_ED	.	.	+1+	+1+	.	.	.	.	+1+	+1+
7	BUILD_ED	.	.	.	.	.	.	-1-	.	.	.
8	BUILD_ST	.	.	.	.	.	.	.	.	.	.
9	ATAN2	.	.	.	.	.	.	.	.	-1-	.
10	DRAWARRO	.	.	.	.	.	.	.	.	.	.
11	DRAW_EDG	+3+	+3+	+3+	+3+	+2+	+1+	+1+	+3+	+2+	+3+
12	UPDATE_E	.	.	.	.	.	.	-1-	.	.	.
13	UPDATE_S	.	.	.	.	.	.	.	.	.	.
14	DELETE_E	.	.	.	.	.	.	+1+	.	+1+	.
15	BUILD_NO	.	.	.	.	.	.	-1-	.	-1-	.
16	DRAW_NOD	.	.	.	.	.	.	+1+	.	.	.
17	UPDATE_N	.	.	.	.	.	.	-1-	.	-1-	.
18	DELETE_N	.	.	.	.	.	.	+1+	.	.	.
19	PRINT_ST	+1+	+1+	+1+	.	+2+	+1+	.	+1+	.	.
20	PRINT_SI	.	.	.	.	.	.	.	.	.	.
21	PRINT_DA	.	.	.	.	.	.	.	.	.	.
22	STATUSLI	.	.	.	.	.	.	-1-	.	.	.
23	INITIALI	.	.	.	.	.	.	.	.	.	.
24	LOAD_DAT	+1+	+1+	+1+	.	.	.	+1+	+1+	+1+	+1+
25	SAVE_DAT	.	.	.	.	.	.	+1+	.	.	.
26	GET_NEXT	.	.	.	.	.	.	.	.	.	.
27	CALCULAT	.	.	.	.	.	.	.	.	-1-	.
28	SIM_PATH	.	.	.	.	.	.	.	.	.	.
29	SIMULATE	+2+	+2+	+2+	+2+	+2+	+1+	+1+	+2+	+1+	+1+
30	QUIT_RGS	.	.	.	.	.	.	.	.	.	.
31	CLEAR_SC	.	.	.	.	.	.	.	.	.	.
32	DRAW_MEN	.	.	.	.	.	.	-1-	.	.	.
33	CONTROL_	.	.	.	.	.	.	.	.	.	.
34	RGS	.	.	.	.	.	.	.	.	.	.

References

[Basi84] V. Basili & B. Perricone, "Software Errors and Complexity: An Empirical Investigation", Communications of the ACM, vol. 27(1), Jan. 1984, pp. 42-52.

[Bern84] G. Berns, "Assessing Software Maintainability", Communications of the ACM, vol. 27(1), Jan. 1984, pp. 14-23.

[Boeh76] B. Boehm, "Software Engineering," IEEE Transactions on Computers, vol. C-25(12), Dec. 1976, pp. 1226-1241.

[Boeh84] B. Boehm, "Verifying and Validating Software Requirements and Design Specifications", IEEE Software, vol. 1(1), Jan. 1984, pp. 75-88.

[Bran90] D. Brandl, "Quality Measures in Design," ACM SIGSOFT Software Engineering Notes, vol. 15(1), Jan. 1990, pp. 68-72.

[Cont86] S. Conte, H. Dunsmore, & V. Shen, Software Engineering Metrics and Models, Benjamin/Cummings, Menlo Park, Ca., 1986.

[Cook84] C. Cook, "Software Complexity Measures," Proceedings of the 1984 Pacific Northwest Software Quality Conference, Portland, Oregon, pp. 343-363, 1984.

[Elsh84] J. Elshoff, "Characteristic Program Complexity Measures," Proceedings of the Seventh International Conference on Software Engineering, IEEE, Florida, pp. 288-293, 1984.

[Gord79] R. Gordon, "Measuring Improvements in Program Clarity", IEEE Transactions on Software Engineering, vol. SE-5(2), Mar. 1979, pp. 79-90.

[Grem84] L. Gremillion, "Determinants of Program Repair Maintenance Requirements", Communications of the ACM, vol. 27(8), Aug. 1984, pp. 826-832.

[Hall84] N. Hall & S. Preiser, "Combined network complexity measures," IBM Journal of Research and Development, Jan. 1984, pp. 15-27.

[Hals77] M. Halstead, Elements of Software Science, Elsevier, New York, N/Y., 1977.

[Harr86] W. Harrison & C. Cook, "A Micro/Macro Measure of Software Complexity," Journal of Software Systems, vol. 7(3), Aug. 1987, pp. 213-219

[Henr90] S. Henry & C. Selig, "Predicting Source Code Complexity at the Design Stage," IEEE Software, vol. 7(2), Mar. 1990, pp. 37-44.

[IEEE87] IEEE, Software Engineering Standards, John Wiley, New York, N.Y., 1987.

[Jone79] C. Jones, "A Survey of Programming Design and Specification Techniques", Proceedings, Specifications of Reliable Software, Apr. 1979, pp. 91-103.

[Kafu85] D. Kafura, "A Survey of Software Metrics," Proceedings of the 1985 ACM Annual Conference, Oct. 1985, pp. 502-506.

[McCa89] T. McCabe & C. Butler, "Design Complexity Measurement and Testing," Communications of the ACM, vol. 32(12), Dec. 1989, pp. 1415-1425.

[Nann85] T. Nanney, Computing and Problem-Solving with Pascal, Prentice Hall, Englewood Cliffs, N.J., 1985.

[Oman86] P. Oman, "Software Engineering Practicums: Case Study of a Senior Capstone Sequence," ACM SIGCSE Bulletin, vol. 18(2), June 1986, pp.53-57.

[Parn79] D. Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, vol. SE-5(2), Mar. 1979, pp. 128-138.

[Pfle87] S. Pfleeger, Software Engineering: The Production of Quality Software, Macmillan, New York, N.Y., 1987.

[Reyn87] R. Reynolds, "The Partial Metrics System: Modeling the stepwise refinement process using partial metrics," Communications of the ACM, vol. 30(11), Nov. 1987, pp. 956-963.

[Rohl83] J. Rohl, Writing Pascal Programs, Cambridge University Press, London, England, 1983.

[Romb90] D. Rombach, "Design Measurement: Some Lessons Learned," IEEE Software, vol. 7(2), Mar. 1990, pp. 17-25.

[Salt82] N. Salt, "Defining software science counting strategies," ACM Sigplan Notices, Mar. 1982, pp. 17-26.

[Shel86] G. Shelly, T. Cashman, & S. Forsythe, Turbo Pascal Programming, Boyd & Fraser, Boston, MA., 1986.

[Shen85] V. Shen, T. Yu, S. Thebaut, & L. Paulsen, "Identifying Error Prone Software - An Empirical Study," IEEE Transactions on Software Engineering, vol. SE-11(4), pp. 317-324, April, 1985.

[Szul81] P. Szulewski, P. Bucher, S. DeWolf, & M. Whiteworth, "The measurement of Software Science parameters in software design," Performance Evaluation Review (ACM Sigmetrics), vol. 10(1), 1981, pp. 89-94.

[Taka85] M. Takahashi & Y. Kamayachi, "An Empirical Study of a Model for Error Prediction", Proceedings of the Eight International Conference on Software Engineering, 1985, pp. 330-336.

[Troy81] D. Troy & S. Zweben, "Measuring the quality of structured designs," Journal of Software and Systems, vol. 2, 1981, pp. 113-120.

[Zelk79] M. Zelkowitz, A. Shaw, & J. Gannon, Principles of Software Engineering and Design, Prentice-Hall, Englewood Cliffs, N.J., 1979.

Paper 5-T-2

THE INSTALLATION OF AN INTEGRATED CAD COMPONENT DATABASE LIBRARY

Mr. Darl Patrick
SANDIA National Laboratories

Mr. Darl Patrick has been a member of the Sandia Laboratory technical staff for nine years. His current assignment is the definition, design, installation, and maintenance of an engineering data base for an Engineering Design and Manufacturing Automation (EDMA) CAD/CAM system. Prior to his current assignment, Mr. Patrick certified high risk automated test equipment and software. He has developed and implemented several CASE tools within the Department of Energy Nuclear Weapons complex. He is a member of the IEEE standards committee for data exchange between software tools and has been a principle speaker at national and international software conferences.

SAND90-0792c
Unlimited Release
Printed May 1990

The Installation of an Integrated CAD Component Database Library

Darl P. Patrick
CAD Applications Software Development Division
Sandia National Laboratories
Albuquerque, NM 87185

Abstract

The installation of a new computer aided design (CAD) system involves more than the purchase of new hardware and software. The installed electronic component database contains models which are used to support the laboratories assigned mission. The transfer and integration of that data base information to a new database hardware and software environment is a complex and delicate task. This paper describes the transition process Sandia National Laboratories conducted to move electronic component models off of a centralized CAD system to an integrated distributed system.

Acknowledgment

The author would like to thank R. E. Thompson of the CAD Applications Software Development Division for his assistance in developing the centralized component database concept at Sandia, and T. B. Linnerooth of the Microelectronics CAD Development Division for his help in establishing the integrated component library requirements.

Contents

Figures

SAND90-0792c
Unlimited Release
Printed May 1990

The Installation of an Integrated CAD Component Data Base Library

Darl P. Patrick
CAD Applications Software Development Division
Sandia National Laboratories
Albuquerque, NM 87185

Introduction

Computer Aided Design (CAD) software provides design engineers automated engineering design assistance. Poor data transportability between data bases limits the CAD usefulness. The lack of data transportability causes severe problems in the CAD environment forcing many engineers to use translators or filters to move information between systems. The translators become operating system and language dependent, further limiting the engineer's capability to move information from one CAD package to another. Furthermore, different CAD data bases compound the problem of moving CAD information from one engineer to another. Establishing a corporate linked component data base is one solution to this problem.

The purpose of this paper is to describe a linked engineering component data base design. The database design includes conversion from a multiple data base to a single data base environment, installation, management, maintenance, and the cost of a corporate linked database.

Background

Data migration from one computer system to another or from one version to a newer version occurs for several reasons.

o A company decision to

the United States Department of Energy supports this work under contract DE-AC04-76DP00789.

install new hardware and software capabili-- ties.
- o The vendor has released a new version of the software and will eventually withdraw support for the existing version.
- o A company desire to move to a new computer aided design (CAD) system.
- o The vendor has dis- continued support of the existing system.

When a corporation decides to move from one database system to another, plans must be made to move the component database. The component database information is a major asset of the company. Efficient migration of the component database is a critical element of the company's ability to function competitively. Decisions made when moving a component database within or from outside the company are the same.

Objectives

The component database technology should be chosen based on specific strategic objectives. A thorough understanding of company needs, requirements, and the effect on future expansion of the data system is necessary.

Two ideas for managing a component database are a centralized database and distributed database. The older centralized design uses a mainframe computer to hold all user files. Distributed systems using stand alone workstations have large amounts of local memory and hard-disk space. Users of stand alone workstations have enough local software to complete their jobs without accessing the server. Workstations connected together can pass data files over a local area network (LAN).

Both systems use servers and a network. The local area networks may connect to a larger network via a gateway.

Characteristics of a centralized data system:

<u>Advantages</u>
- o Brings applications to a single location.
- o Centralized maintenance of data files.
- o All users have access to the same data files.
- o No redundant data entry.
- o No data inconsistencies.
- o Central point of soft- ware revision control.
- o Training cost reduced.
- o Operating system admin- istration reduced.
- o Application system administration reduced.
- o Long term system costs reduced.

<u>Disadvantages</u>
- o A failure of the host can disrupt the entire network.
- o A failure of the net- work can isolate net nodes with no service.

The centralized database system does not fit all company structures. Some sites are moving to a distributed database. The

distributed database uses workstations with local hard disk drives and installed memory.

Characteristics of a distributed data system:

Advantages
- o Independent data operation.
- o Flexibility in hardware and software selection.
- o Flexibility in third party vendor support.
- o Lower short term acquisition cost.
- o High reliability.
- o Network independent.

Disadvantages
- o Increased systems administration effort.
- o Difficult to maintain all databases consistent on a large system.
- o High equipment maintenance costs.

Priorities

Moving from one CAD system to another does not mean creating an entirely new data system. The company has invested a large amount of time and money developing the current data structure. The component database migration to the new CAD system must be undertaken with caution. System librarians, administrators, and managers should complete an analysis of the benefits and risks of moving to a new system before deciding to purchase a new CAD package. The component database librarians must rank the identified migration goals and tasks before attempting to move a database between systems. Some of the goals and tasks identified are:

Migration Goals

- o Define, design, and implement a migration path.
- o Minimize disturbance to users.
- o Use theoretical data, old data, and experience to configure the new database to meet requirements.
- o Ensure that a complete component database backup and recovery strategy is in place.
- o Support customers' needs as soon as production starts.
- o Determine old, new, or modified standards usable with the new database.
- o Complete all librarian training before start of production.

Migration tasks

- o Migration of current applications and shell scripts. Move or rewrite?
- o Estimate ease of tuning the new database.
- o Applications migrated should be stable prior to full user access. New features can hide old and new bugs.
- o Establish process flow charts and determine skills and training needed to support the new database.
- o Understand what jobs must be done and what jobs are nice to do.
- o Set a time for withdrawing support for the old system.
- o Back up (archive) the

old system (twice).
- o Ensure that correct access protection in files and tasks are in place.
- o Change all user log-in environments.

System Configuration

The centralized and distributed data base both have strong and weak points. Each represents an extreme version. Sandia National Laboratories, Division 2854, realized that a blend of the systems would be necessary. A successful system migration of the component database would serve the greatest number of customers, and be able to draw on skills already established. The degree of blend would determine the systems characteristics. The potential customer base is 200 seats. These seats encompass the following disciplines:

- o Hybrid (HMC) Microcircuit design
- o Analog Circuit design
- o Digital Circuit design
- o Printed Wiring Board (PWB) design
- o ASIC design
- o Custom VLSI design
- o Component Library Support
- o PWB and HMC Manufacturing

The component library will have to serve each of these disciplines as both a common source and a unique resource.

None of the disciplines listed are in a common building. Indeed, the network at the production agency is one-thousand miles from the Sandia location. Each user group will establish a local network. The establishment of an inter-linking network will occur after local users have equipment on line and production use has started.

The centralized component data base design allows linking of major nodes. Each node has a local librarian trained in the skills needed to support the node users. There are four parts of each local library.

- o Local User Library
- o Released Library
- o Commercial Library
- o Import/Export Temporary holding file area

The local librarians also maintain metrics and tools unique to their function, and component data base administration skills, Figure 1.

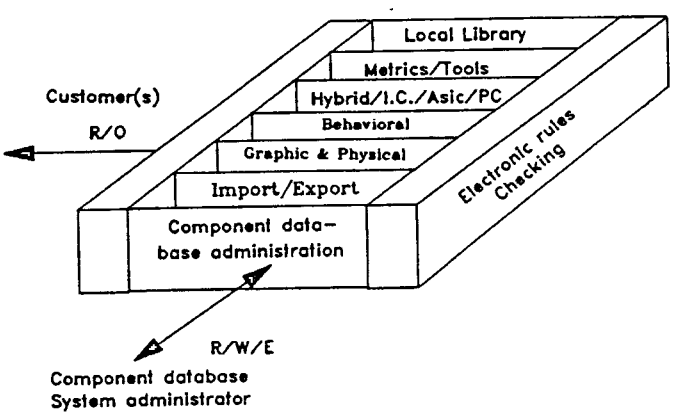


figure 1
Local Component
Database Structure

For each of the four library parts, the local librarians maintain graphic, physical, and simulation, models for their node. Administration and metric software is common to each of the library types. A sneaker-net, under control of the system librarian, maintains a "master copy" of the local nodes. Each of the Department of Energy (DOE) production agencies receives a copy of the master component library. Biweekly, the changes from each local librarian are up-loaded to the master component library. The system librarians validate all changes before their release. Each of the production agencies receives copies of certified changes entered into the master library. Sandia National Laboratories, Division 2854, reviews changes made by any of the production agencies. A system buffer holds all changes until the system librarian certifies them. The system librarian certifies the changes and adds them to the master library. Applicable local librarians receive changes upon completion of certification. The system library, figure 2, is the only complete copy of the corporate component library. Customers have access to component data "outside" their area only by

Integrated Component Database

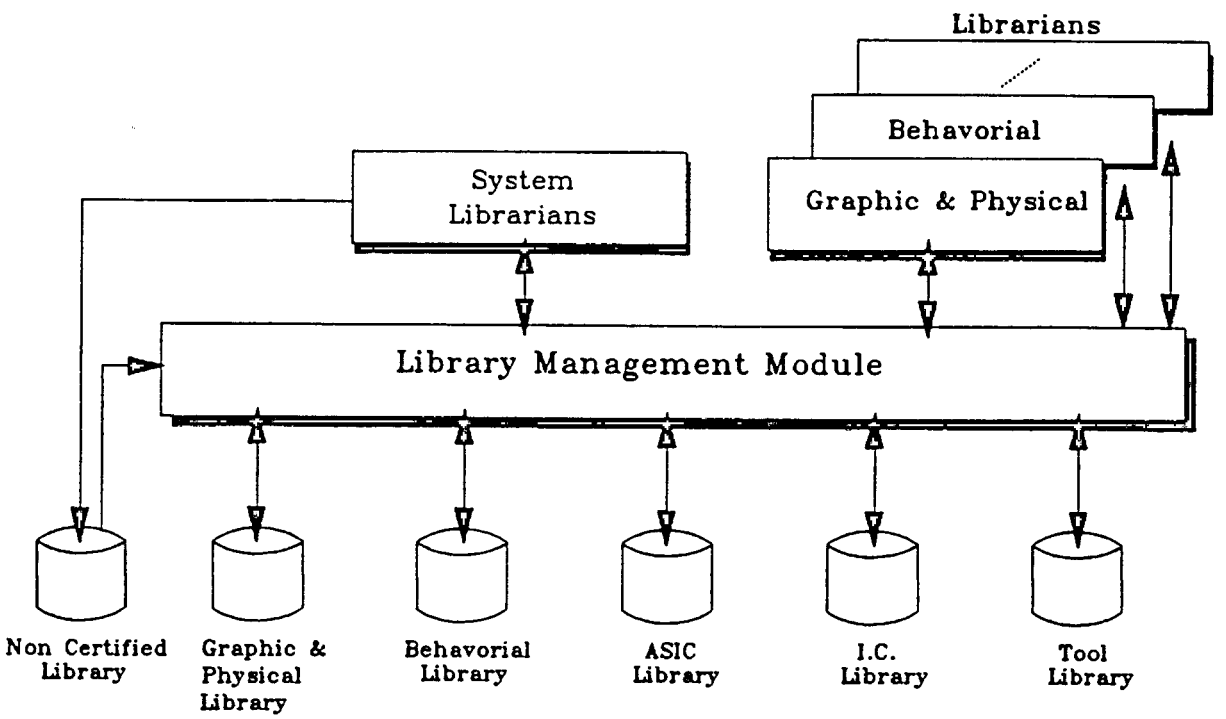


figure 2

requesting a down-load from the system librarian. The down-load is done by tape using the "sneaker-net," figure 3.

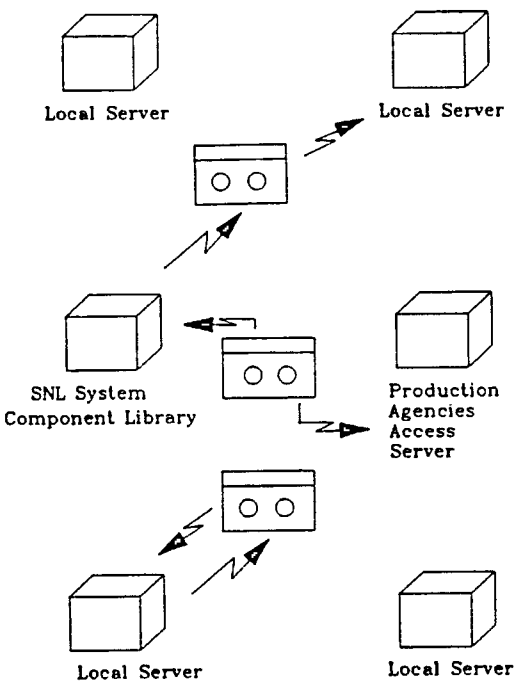


figure 3
Node Sneaker-Net

The "sneaker-net" depicted in figure 3, is an interim system installation. Each local server will eventually connect to the system component library server.

The CAD component database library system contains elements of both the centralized and distributed database systems. The database system is different in concept from the computing system. A distributed computing system may contain either a centralized or a distributed database. Each user network or node, uses a server, has an administrator, a librarian, and a common local component database. The system component database library is a distributed network in the initial phase. Installation of a communications network will allow a gradual migration from a distributed system to an integrated system for all users.

The design of an Integrated CAD Component Database Library has the following attributes:

- o Local librarian component control.
- o Local component database.
- o System component database availability.
- o Revision control.
- o System and Local administration control.
- o Metrics and tool libraries.
- o Common Interagency component database maintenance.
- o Released and "Working Libraries" at user and system levels.
- o Release Procedures.

Completion of the communication network will move the systems component library from a distributed system to an integrated system. In a fully integrated system the system database will be accessible to all users. Local librarians will maintain only those libraries which their customers need daily.

System librarians will no longer need the sneaker net to maintain the component database.

models may not be usable in the delivered form. Local librarians and system librarians must convert heuristic and tabulated data

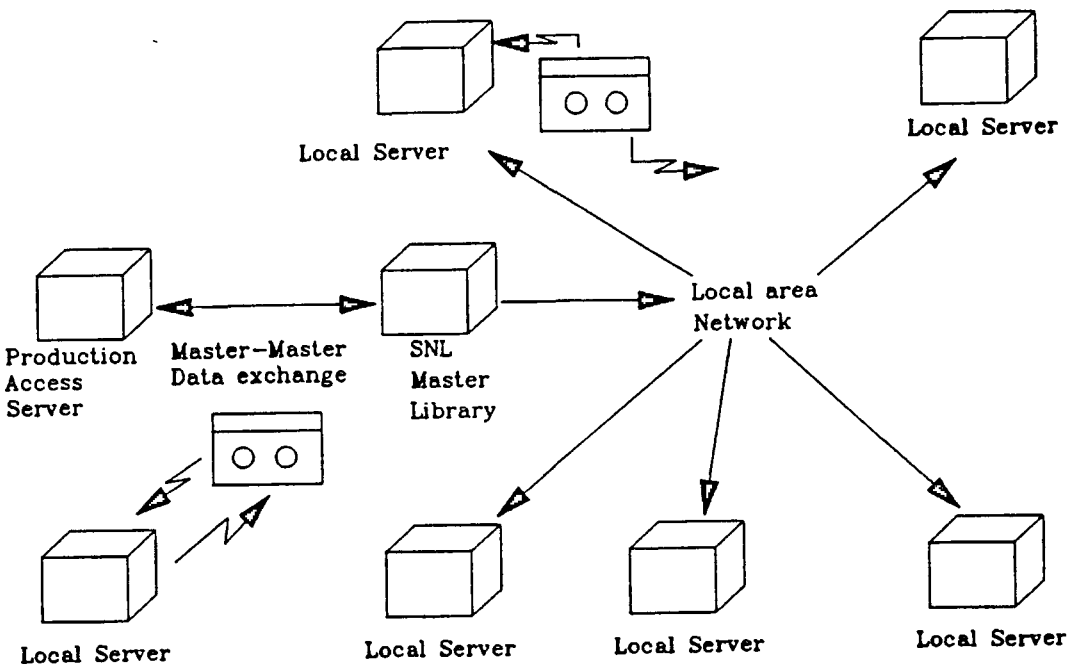


figure 4

Component Certification

An integrated or common component database is of little use unless the users are confident that the components are accurate. Accurate components are those components which meet the requirements and specifications of the customer using them. All CAD systems come with a set of component symbols. These include graphic, physical, and simulation models. The

used to check components, into a set of metrics. A component electronic rules checking (CERC) soft-ware package uses the defined metrics. The use of a CERC guaranties that all components placed in the released component library meet a common set of requirements.

Most Sandia Laboratory systems which have evolved, have not followed published standards such as MILSPEC or

IEEE. Their failure to adapt standards prevents them from using most commercially delivered component databases. This requires that each delivered component be "touched" by a librarian in some manner. The requirement for a librarian to 'touch' or alter each component to meet a non-standard requirement results in an additional price of non-conformance (PONC). The PONC's added together, constitute an additional maintenance cost for any system.

The component data certification, figure 5, serves both the local and the system librarians. In either case entry of non certified components occur when the components are:

- o used only as a local resource.
- o for review and entry into the released database.
- o reviewed and entered into the commercial database.
- o used to update existing components.

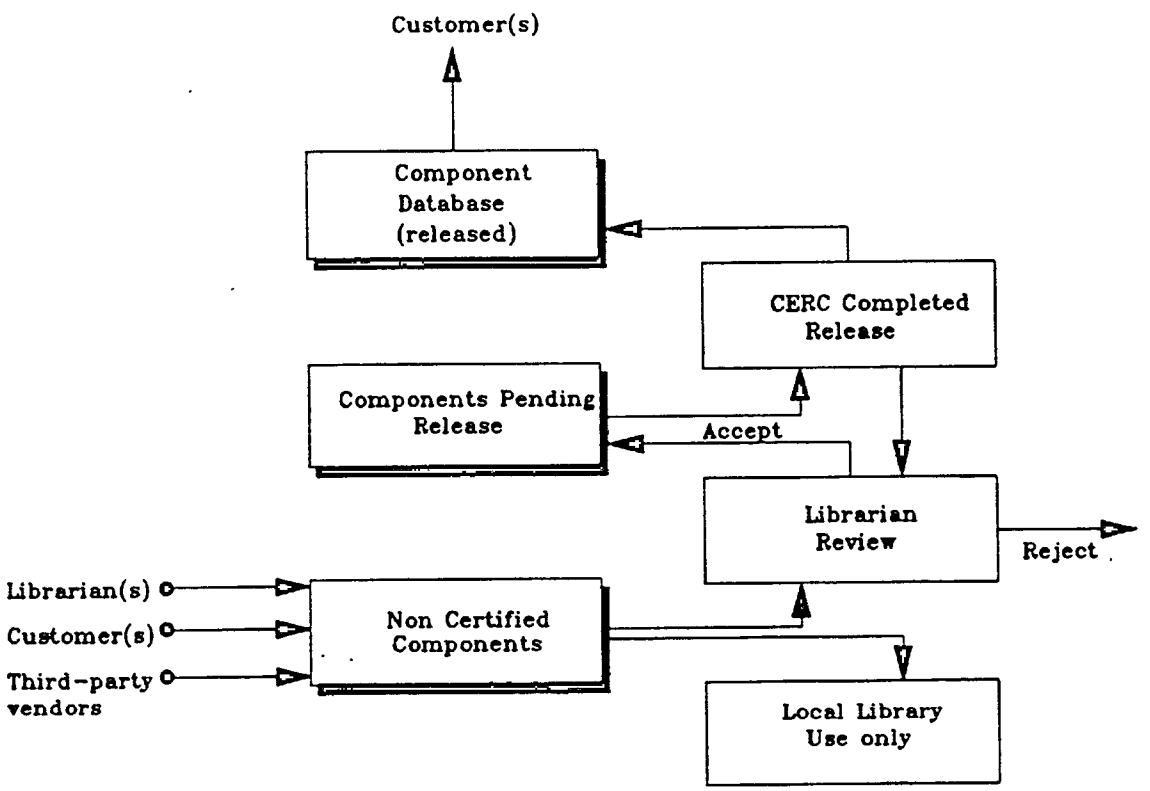


figure 5

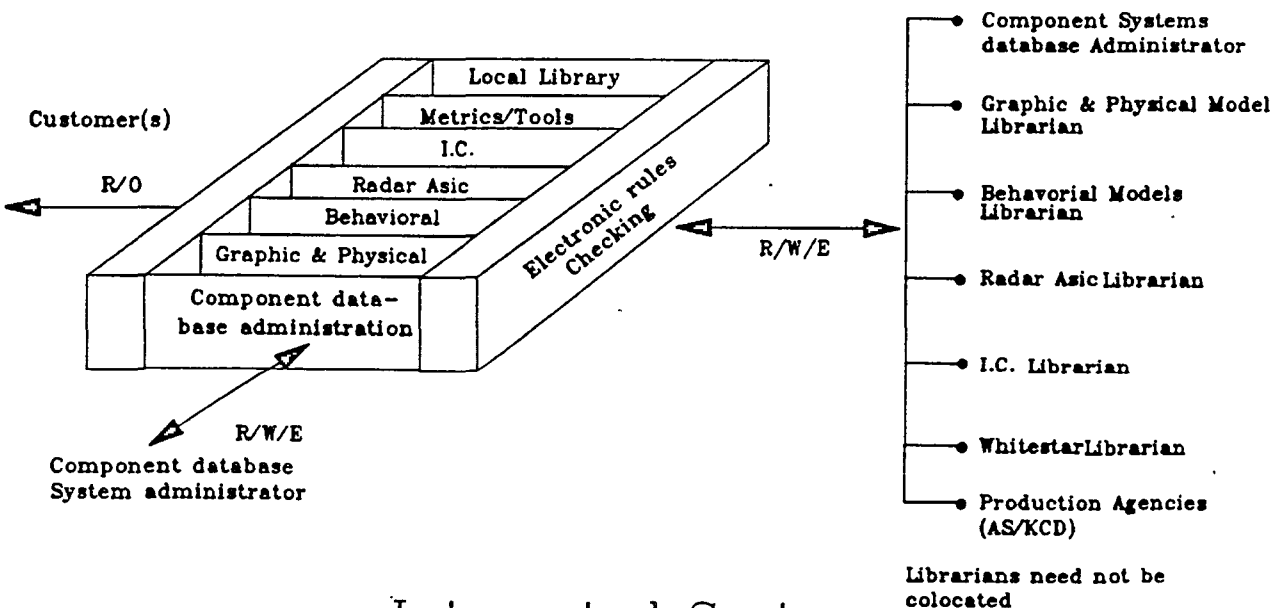
Component Database
Component Certification and Release

Engineers who use components from the local library cannot submit the schematic for any weapons related schematic. Librarians certify all components to some level prior to placing them in the user accessible portion of the database. Certification of components in an integrated data system is a joint repressibility of local and system librarians.

System Costs

The cost of installing a new component database library is the sum of many activities.

- o equipment
- o software
- o license fees
- o old data transfer
- o personnel training
- o maintenance



Integrated System
Component Database
Structure

figure 6

The dollar cost of equipment is foremost in everyone's minds. It is the easiest to measure and often considered a primary element. The Installation of a new database assumes:

- o equipment selection
- o equipment purchase
- o equipment installation

As such, equipment cost is not a consideration in the cost of installing the database.

The cost of installing new component database software contains many hidden costs and must be carefully reviewed and controlled.

Some considerations are:

Library Management:

- o Are there any library management packages?
- o Will only the librarians use the library management software?
- o Can librarians move software between workstations?
- o How are component library software updates handled?
- o How many people will the new system require for maintainence?

Cost Factors:

- o What is the initial cost per user?
- o What are the initial license fees?
- o What are the yearly renewal fees?
- o What is the maintenance cost?
- o Do the librarians need the schematic capture software to test components?
- o Do the librarians need simulation packages to test models?

Personnel:

- o What is the projected personnel support required?
- o Is any reorganization required to achieve the component librarian requirements?
- o How many librarians does the database conversion require?
- o How many librarians does long term database maintenance require?

Training:

- o What training do the librarians require?
- o Who should conduct the librarian training?
- o Where and how is the librarian training conducted?

Documentation:

- o How is the component documentation delivered?
  - paper
  - compact disk
  - tape
  - other
- o What format are up-dates delivered in?
- o What is the history of component database revisions?

Contractor Assistance:

- o Is contractor assistance required to install or maintain the database or updates?

Conclusion:

Sandia Laboratories is using the procedures outlined in this paper to install a new component database, and to migrate the existing database to the new hardware platform. Performance requirements have been written and released for the component database.

It is our intention to install a new component database, and move our present CAD component database system using the check points presented in this paper.

Installation of a new component database or moving a databases is a major effort. Done with careful planning, the migration of the database is nearly transparent to managers and users.

The cost of moving to a new system is significant. It is not a single item paid once. But rather an initial down payment with yearly fees. The control of the yearly fees is a significant factor in determining whether the migration from one database to another is successful.

A check list, using the information contained in this paper, will help determine the "real" cost of moving a database.

WARNING

Installation or migration of a database is like surgery:

THE OPERATION MAY BE DECLARED A SUCCESS -

EVEN IF THE PATIENT DOESN'T SURVIVE.

Paper 5-T-3

BUILDING MAINTENANCE AND QUALITY SUPPORT INTO THE ENVIRONMENT

Mr. Brian Gill-Price
VP Engineering
ProCASE, Inc.

Mr. Brian Gill-Price is a founder and is Vice President of Engineering at PROCASE Corporation where he is responsible for all software development of their CASE/SDE product family. He has over 15 years experience in the engineering software field including management, compilers, operating systems, network services, computer aided engineering (EE), and large software systems architecture. His prior experience includes positions with Tektronix, University of California at Santa Cruz, DHL International and Cybertron Inc. He holds BS degrees in civil and mechanical engineering from Lehigh University.

Building Maintenance and Quality Support Into the Environment

Brian Gill-Price

PROCASE Corporation
3130 De La Cruz Blvd.
Santa Clara, CA. 95054

ABSTRACT

Much of the promise of CASE Tools has been the automation of quality constraints into the development cycle. Many vendors have approached this by teaching "new" ways to do old work cleaner ie. UpperCase Tools.

This talk will address some of the unique techniques applied in the PROCASE Corporation product to support quality and maintainability of large old tired systems of existing program code. These techniques include abstraction, graphical views, structured navigation and token level incremental semantic analysis to give immediate extensive "whole program" quality checking.

Additionally this talk will cover the additional problems imposed by multi-user, shared large and small program issues.

May 3, 1990

Building Maintenance and Quality Support Into the Environment

Brian Gill-Price

PROCASE Corporation
3130 De La Cruz Blvd.
Santa Clara, CA. 95054

What is the SMART System?

PROCASE Corporation has built a Software Development Environment for the development and maintenance of software. This system allows the programmers on a project to quickly comprehend existing software systems, navigate efficiently through them, incrementally check for semantics problems in their software before compilation, and then incrementally compile just those sections of the software requiring update at the function level.

All of the environment applications are built on top of our fine grained persistent C++ object-oriented database. The system excels at dealing with large programs in multi-programmer environments. The system treats programs as entities not just as collections of files. The application tools support both programming in the large and programming in the small.

1. The Key Components of the Environment

The environment is built in four closely coupled spheres covering the foundations of work done in maintaining large existing software systems. On the following figure (fig. 1) those four spheres are shown. They are Analysis and Comprehension, Editing and Multi-User Support, Program Quality Checking and Software Construction/Development Support. All of these components working together can significantly improve the quality of the maintenance process.

One thing Lower CASE tools do is automate and facilitate many of the tasks done during the maintenance phase improving them and helping to eliminate process errors. Each of these spheres is continuing to undergo evolution with new applications being built to better support quality as the environment product matures. In each sphere the way the tools contribute to improvements in software quality are discussed.

2. Comprehension and Analysis Assistance

The Analysis and Comprehension Sphere is the first the user encounters. This is the lense through which the programmer views their software. There are many ways in which this sphere assists with Software Quality Support. The general problem here is how the individual programmer, responsible for much poorly specified code, can understand the system well enough to make any changes. Often software projects have built huge programs for which all knowledge of the structure left with the program authors. An environment can have many tools, whose focus is assisting with discovering how the old program works, and as all these tools help to give a better understanding of the program they will improve quality.

2.1. Product Support For Abstraction and Structural Navigation

Traditional software editing systems, which are file, byte and line oriented tend to be the only tools for viewing code. This product in contrast allows the programmer to deal with programs structurally and as a whole. Since the view is the program as a whole far more of a birds eye view can be given. A programmer is no longer hindered by always seeing things as segmented into the tiny components necessary for efficient compilation etc. Additionally the user collapses or exposes only the structural or textual items that directly relate to the problem the programmer is focused on solving. This immensely reduces the amount of information a programmer will need to retain in their head while

UNIX™

X Windows™

SMARTscreen™

- Configurable Interface
- On line Tutorial and Documentation
- *Conditional Constructs*

Analysis and Comprehension

- Navigation and Filtering
- CPP In Line Expansion
- Configurable Code Formatting
- Call Graphs
- *Control and Data Flow Diagrams*

Editing and Version Control

SMARTsystem™

Syntax and Semantics Checking

- Editing Filtered Views
- Vi, Emacs, Mac Text Edit Emulation
- Multi-Programmer Edit and CM
- Fine-Grained Locking Environment
- *Release Management*

SMARTmake™ and SMARTdebug™

- Incremental/Recoverable Parser
- Selectable Filtering of Semantic Errors
- *Full ANSI Semantics*

- Build Info (Automatic Makefile)
- Incremental Compilation
- Symbolic Debug
- *Call Graph Animation*

PROCASE

working on the problem at hand. In addition information relevant to the specific problem at hand can be brought close together with all the irrelevant detail suppressed out.

An example of a filtered/holophrased view into a source application is given in the (figure 2). Many thousands of such collapsed views can be constructed by the programmer each of them taking mere seconds to have the system supply the projection. Examples of such queries are:

"show only statements assigning a value to the global variable x"
"add to the display any lines containing uses of the type structure personnel_record"
"which places in this program other than file1.c make use of the results of the 'C' PreProcessor Expansion of the getc macro variable type _iobuf"
"Filter out all lines of code other than those on which I have put breakpoints"
"Show only lines of code I have changed during the last few hours of editing"
"Leave only lines with type-mismatch errors on display"

Anything that can be queried about can equally as well be navigated to. Thus the ease of moving through the tremendous amounts of source code which make up typical commercial programs is made quite easy. No longer is the programmer faced with memorizing file names and versions of the components participating in getting a version of the program built as all of this is pleasantly hidden from the immediate required attention of the programmer. The programmer moves through the programmer in a problem directed form not a tool directed (editor and revision control systems) form.

2.2. Product Support For Graphical Views

The product produces near instantanous Call Graphs and other graphical representations to allow the programmer a structural sense of how the program parts work together. Often in demonstration of these capabilities at customer accounts the comment is heard "oh I had no idea this program had that structure!". The maintainance programmer can see immediately what is calling what, and can navigate through the text of the program using the graphical form as a compressed structural view. This frees the programmer from the intellectual energy of remembering files and search patterns to move around in the source code. The tools work together to aid in comprehension and thus in improving quality of resultant products.

An example of a reasonably complicated sub-graph from the Berkeley shown in (figure 3). The whole program produces a graph of about 700 call nodes. This graph of the whole program takes about 12 seconds to display. Selecting this sub-tree in the calling sequence and filtering away all the rest of the graph temporarily took about 2 seconds. This graph can be extended as needed with other nodes or the layout can be changed as needed to improve documentation support. This illustrates the tremendous value in reducing the amount of information the programmer needs to worry about. At any point the programmer can use the graphical structure to move them to the exact desired point in the text.

2.3. Program Analysis With An Eye Towards Quality

The system has encorporated many checks not normally found during normal compilation so as to help programmers relieve the burden of finding code problems related to poor coding standards. The individual or the corporate group can establish the kinds of checks they want to insure the software checks for during development/maintainance changes to the system.

Such simple improvements in the environment as automatic Pretty Printing to any user's choice style on a per user basis are available. An example of how controllable these options are are given in (figure 4). With this support it will automatically make discovering such syntax problems as unbalanced blocks ({ }) easy to find. The code looks unaligned. A simple addition of a balancing (}) block and the code will all format properly. Since every programmer can now deal with the whole system viewed just as they particularly prefer to read code the process of looking at someone else's code has more opportunity to find problems otherwise hidden by syntactic sugar differences in programming style.

In much subtler domains one can for example look at instances of declaring the same variable name in both inner and outer scopes. This is clearly not an error but can often show up subtle bugs where the programmer thought they were referring to the outer when in reality the inner applied. In

SMART () #1

PROCASE Database Edit View Navigate Analyze Options

```
...
FILE *input, *setup, *laser;
...
        laser = stdout;
        if (laser == NULL) {
                ...
        fprintf (laser, "%%!            \n");
...
        fclose (laser);
...
                fprintf (laser, "%s", line);
...
                                fprintf (laser, "copypage\n
                                fprintf (laser, "erasepage\
...
                                fprintf (laser, "letter_hea
...
                                fprintf (laser, "additional
...
                        fprintf (laser, "newline\n");
...
                        fprintf (laser, "(%s) show newline\
...
                fprintf (laser, "copypage\n");
                fprintf (laser, "erasepage\n");
...
        fprintf (laser, "GlobalState restore\n");
        fflush (laser);
...
```

Decl Xref Patn Excp Chng Func Stmt Peer File

File Peer Stmt Func Chng Excp Patn Xref

func: <global>
file: cpp_demo.c

unparsed: 0
exceptions: 8

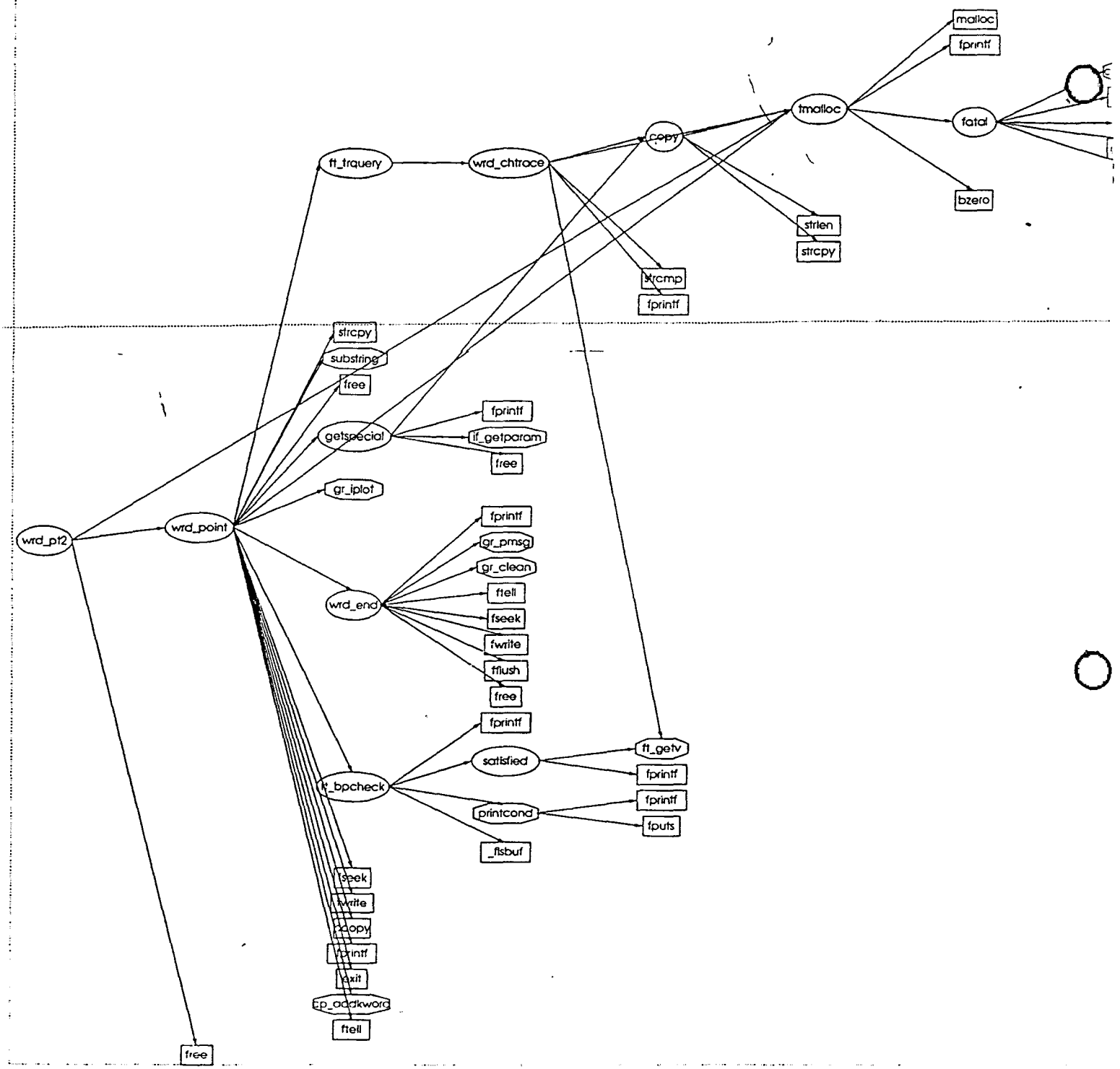
Undo Cut Copy Paste -- - cpp + ++ Line Stmt expand Func File Syntax Smantx

Edit Debug CC Hide HideX Show ShowX filter Refs +Refs Patn Excp Chng

Lock Name Goto Decl Refs Patn

Cross Reference Filter on Variable "laser"

Figure 8



picewrd_pt2 subtree Call Graph

PROCASE

Format Rules

Indentation:

block size (columns) 4 tabstop 8

☒ Use tabs in indentation (when possible)

Extra indentation (in columns) of:

open brace	0	comments	0
close brace	1	trailing comments (tabs)	1
goto labels	-999	parameter decls	0
switch labels	-999	continued lines	-1

Newlines:

- ☐ Before block begins '{'
- ☒ After funct header 'f()' before '{'
- ☒ After block ends '}' before 'else'
- ☐ After first half of metastatement
- ☐ After initializer values
- ☐ After type in func defn
- ☒ After goto labels
- ☒ After switch labels
- ☒ After statements
- ☐ After each declaration

Spaces:

- ☐ After function name (call)
- ☐ Before first parameter (call)
- ☐ Between parameters (call)
- ☐ Between empty parens (call)
- ☐ After if, etc. before '('
- ☐ Before condition after '('
- ☐ After 'for ('
- ☐ After 'for (' before ';'
- ☐ After function name (decl)
- ☐ Before first param (decl)
- ☐ Between parameters (decl)
- ☐ Between empty parens (decl)
- ☐ Before typename in cast
- ☐ Before expression after '('
- ☐ After 'for' semicolon

Filtering:

'...' placement: ○ End of line ○ Start of line ◉ Indented

Compression:

Leading whitespace	◉ Normal	○ Removed	○ Compressed
User blank lines	◉ Left In	○ Removed	
Formatter blank lines	◉ Left In	○ Removed	

Saved Rule Set:

format_rules

Read Write

☒ "Apply" after every rule change

☐ **Enable Formatter**

Apply Exit

much the same way a simple mouse click and one can tell the type a variable eliminating a common source of errors where the programmer forgets the type and due to difficulty of looking it up does not check the type.

The system does many checks that cross compilation units to insure that subtle program wide problems will not appear. All told the system has at present approximately 60 user controllable checks available for individual choice as to checking and display.

The complete description of "how to build" the program is captured so as to eliminate common "I forgot to use the right compiler flags" errors. The system automatically computes dependencies between program structure components allowing the programmer to browse said structure giving a clearer view of how the system as a whole works together. This perspective helps tremendously in allowing programmers to get at whole programmers with less intellectual energy spent memorizing how they got around in the source code. All of the items contribute to much higher quality in the resultant software produced.

3. The Editing and Multi-User Support Sphere

This sphere allows the user to interact with the environment to support changes to the system with the minimum of effort and maximum control over process quality.

The product fully integrates version control for multi-user support with the editing environment. It also integrates with structured editing to maximize the ability to detect errors. Since the user interface to the product is completely soft the user can change all the look and feel as well as model much of the tedious interaction they would do with outside environment tools within the environment in a far simpler style.

As examples of this kind of integration transparent access to remote source files on mainframe/mini file server machines including automatic remote configuration management system fetch operations can be done. This in ways that make a local programmer on a multi-window workstation not even know where the files came in from. Operations such as remembering all the files the user has changed so that all of them will automatically get checked in upon completion of the work have been fully automated.

Once a database is established the different product users of that database have access to full version management. This includes fine-grained locking down to individual lines of code. Since this reduces the number of locking conflicts encountered it minimizes the need for branching in software. Thus it will hopefully reduce the resultant number of incorrectly merged software changes. No longer does the programmer have to deal with the intellectual complexity of figuring what parts comprise their build environments. We find this simple problem is incredibly and surprisingly difficult at sites all across the country where programmers cannot tell exactly what makes up the system.

This multi-user support also conveniently allows each user to have a complete unique configuration they are working on without always having to track which pieces of which components are required to work together to make a consistent software build. The system remembers exactly what all components are that make up the system each programmer is working with.

The editing models look and behave as many editors on the market including such popular paradigms as vi, emacs or editors such as those available on the Macintosh. The editing models have been smoothly integrated to the previous sphere of comprehension support in a way such that a user who has retrieved sufficient information such as 200 lines on the screen from a 200,000 line program can still edit as normal. This is true even though all of the remaining 199+ thousand lines are not visible.

Structured editing commands work along side of typical file oriented operations such as scrolling etc. Unlike most structure editors changes are not required to be complete, correct or meaningful until the user feels like establishing such constraints. The programmer can work in any order or style as suits the problem solution they are attempting to solve.

4. The Program Quality Checking Sphere

4.1. Current Tool Support

This sphere insures that as changes to the environment are being made they will work in the context of the program as a whole and in concert with the best work in vogue for program verification. As work progresses the user can always look and see how many errors or warnings of the indicated type are present in their individual environment for the program. This is continuously updated and taylored for the user.

Current tool work includes extensive user controlled semantics checks. As a code auditor the system helps the user by allowing them to choose how to model the kinds of quality checks a particular user want performed.

Examples of the kinds of checking done include:

"what variables are declared with the same name in both inner and outer scopes"
"identifiers are declared but never used"
"for ansi checks does a function have a prototype?"
"are variables/functions declared in several files declared the same?"
"is a function declared to return a value and in turn not doing so?"

One can make the checking look like the checking done by a particular compiler or one can have it check more rigorously assisting in finding problems not found by local compilers used. All of this checking is optional so any individual user can control the quality level as suits their individual interest.

4.2. Future Directions in Program Quality Support

This is the area where for example PROCASE has been constructing such tools as the not yet available 'C' data flow analyzer to discover many common quality flaws occuring in software as well as providing a rich base for driving much of the testing process as other conference testing papers suggest. An experimental version of this tool shows a lot of promise especially for the 'C' language with it's extreme use of pointers. Many customers have had to suffer through examples of memory bashing due to undefined reference code not found by any tools in their arsenal today.

On the extreme high end the user can imagine perhaps the concepts of the mid-70's for using Symbolic Evaluators as an integral part of modelling the behavior of their software with an I towards testing the software much more thouroughly. While this is not yet a reality the vision in CASE is not that far away!

In a much more mundane however important direction the current product is gaining from many standard kinds of metrics becomming available as standard forms of queries the users can construct. Since they are another part of the environment the customer can begin to determine their own degree of interest in making metrics a part of their quality measurement process in the software life cycle.

5. The Program Construction and Development Support Sphere

The environment could not work well solving all the above problems without dealing with such programming in the small problems as how to interface with production compilers and the results of execution.

The user needs to get their software built using compilers. In the midst of the environment discussion from above this is where the ugly problems of how to produce real code come to play. The current approach taken has been to interface to external compilers so as to benefit by not removing the dearly loved outside compiler from the picture. However a major improvement is to reduce the amount of material that normally needs to get compiled by making only exact functions which have changed get re-compiled rather than the whole file as currently exists in "normal" tools.

This is a truly fruitful area for software testing in the future as a wonderful way to transparently slide in various test vector generation codes as the product evolves.

For the present though it get the new program built faster and with less strain on the programmer. This makes the programmer less error prone and more inclined to have less stress since less mental energy is required to keep their world in one piece.

6. Summarizing the Benefits of Complete Environments

The whole cloth works together in many ways to reduce the problems currently encountered building software. This contributes significantly to improving the quality of our work and enabling us to solve the larger problems coming in this industry.

Paper 5-M-1

SOFTWARE QUALITY METRICS: A EUROPEAN APPROACH

Dr. James Hemsley
President
Brameur, Ltd.

Dr. James Hemsley is Managing Director of Brameur, Ltd., a European Research Consultancy with a special focus on Software management, operating out of Hampshire, England. He is the Project Manager of two ESPRIT Projects on Software Quality Metrics: MUSE and METKIT, and is a frequent speaker at high-level technical conferences throughout Europe. He is also a member of ISO/IEC, European and British Standards working groups in the Software Quality Assurance field.

Paper 5-M-2

POPULAR QUALITY ASSURANCE TECHNIQUES: A BUSINESS ANALYSIS

Prof. D. L. vonKleeck
President
vK Systems, Inc.

Prof. D. L. von Kleeck has twenty plus years in computer industry. He is a professor of the Lubin School of Business Pace University, has an MBA from Fairleigh Dickinson University, and is a doctoral candidate at Pace University. He is an independent management consultant in systems productivity as well as a co-author of a leading commercially available software engineering estimation package.

Popular Quality Assurance Techniques:

A Business Analysis

By D. L. von Kleeck

Associate Professor
Pace University
Lubin School of Business
Pace Plaza
New York City, NY 10038

RESEARCH PROCESS OVERVIEW

* Researched eight basic QA techniques:

* Developed model based on industry research of:
    1. International Business Machines
    2. Howard Rubin Associates

* Baselined case study development and maintenance costs.

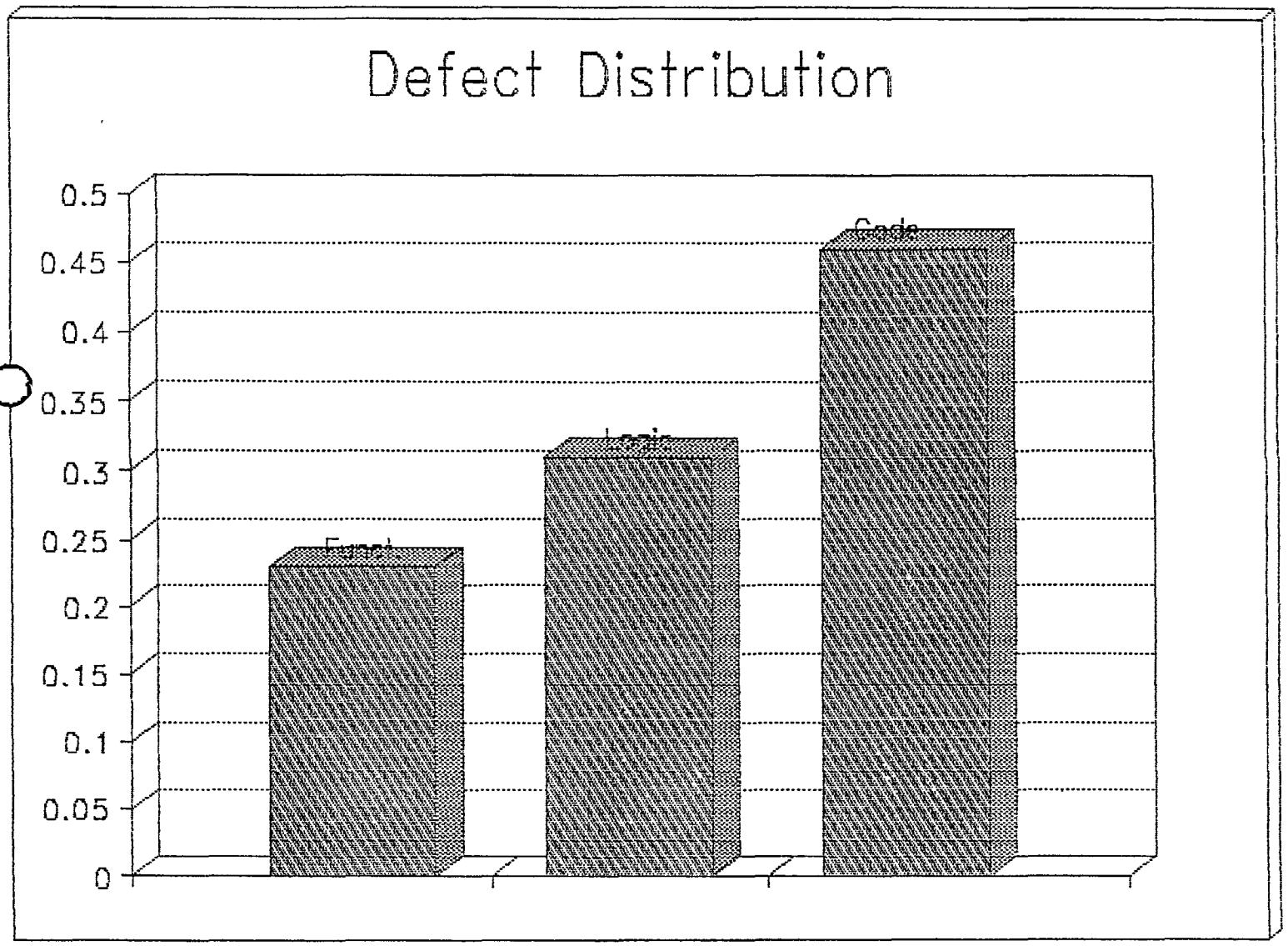
* Simulated development and maintenance costs for various QA skill and utilization level scenarios.

* Calculated ROI for various scenarios simulated.

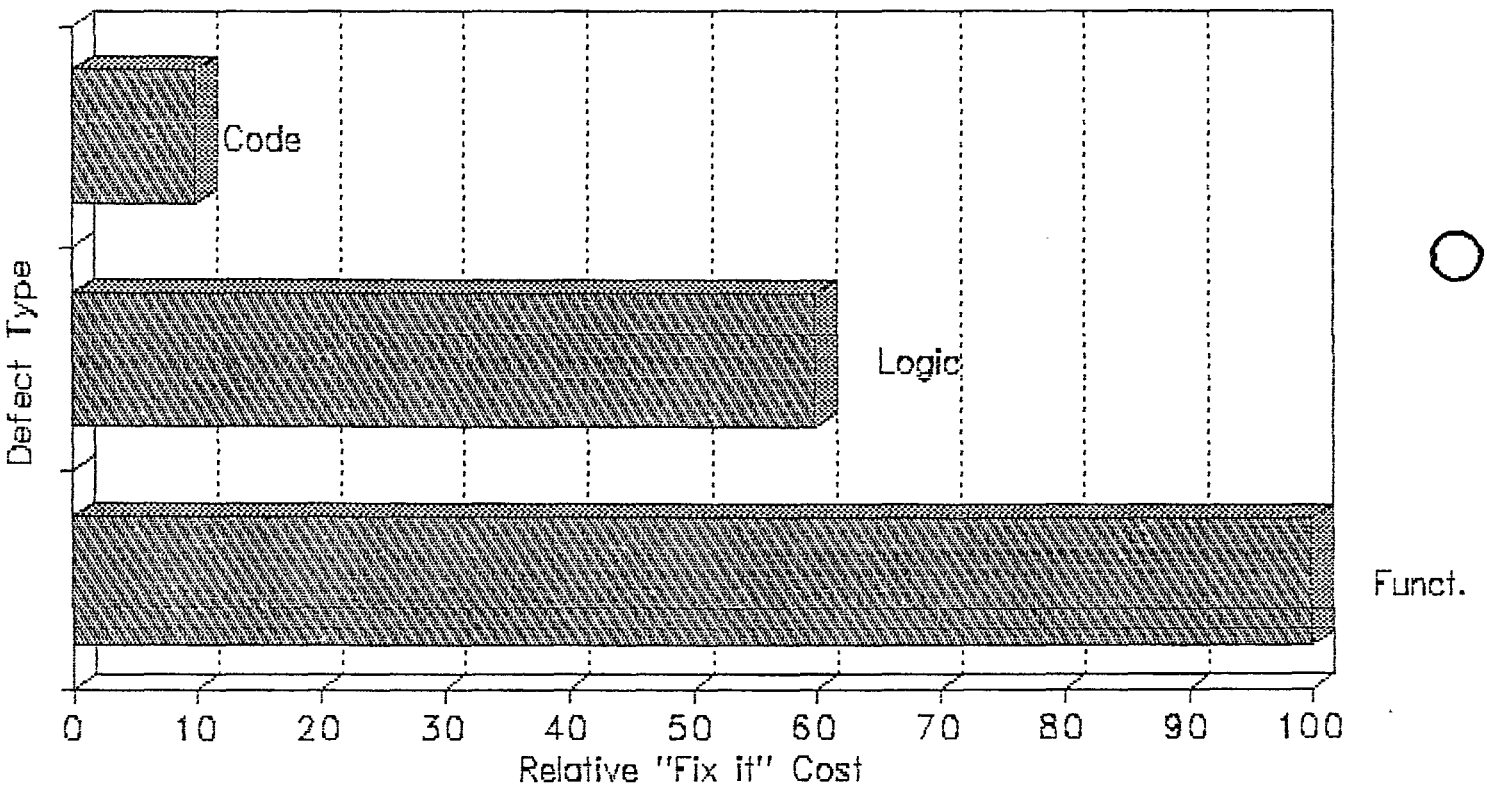
* Clustered scenarios based on ROI and observed resulting clusters were defined by genaric QA modes:
    1. Reviews
    2. Inspections
    3. Tests

* Developed cluster "power index" using weighted average ROI.

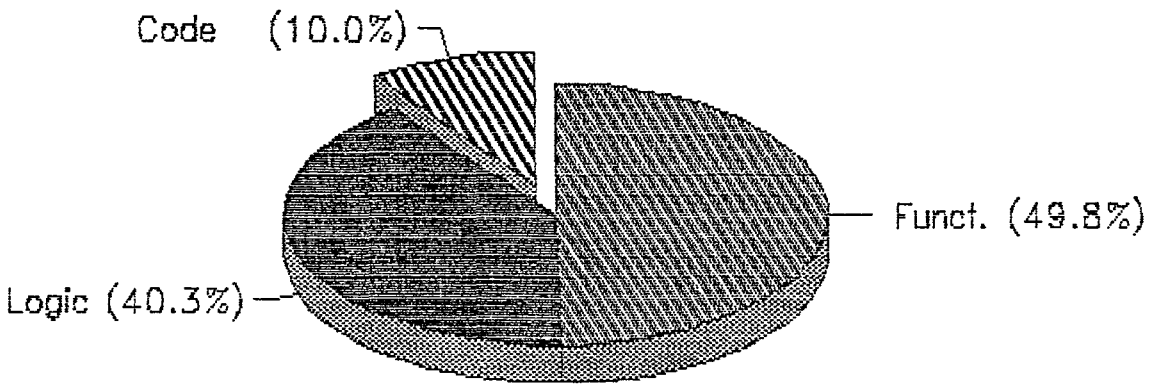
* Results indicate that the review techniques are approximately 2 times more powerful than testing, and inspection techniques are 4.5 times more powerful than review techniquies - 9 times more powerful than testing techniques.


Defect Distribution
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
Code

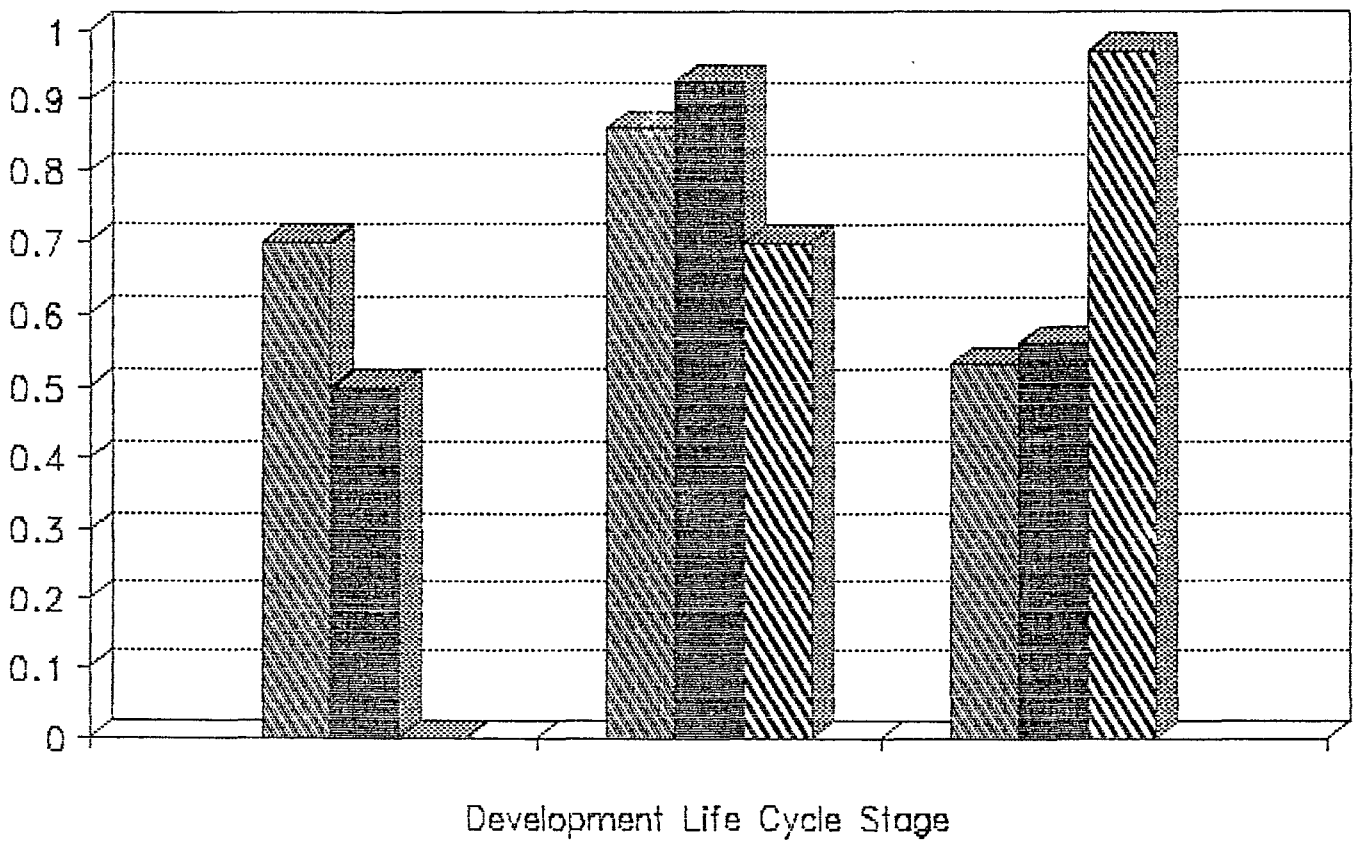
Relative Defect Cost
by Defect Type



"Fix it" Cost Distribution
by Defect Type

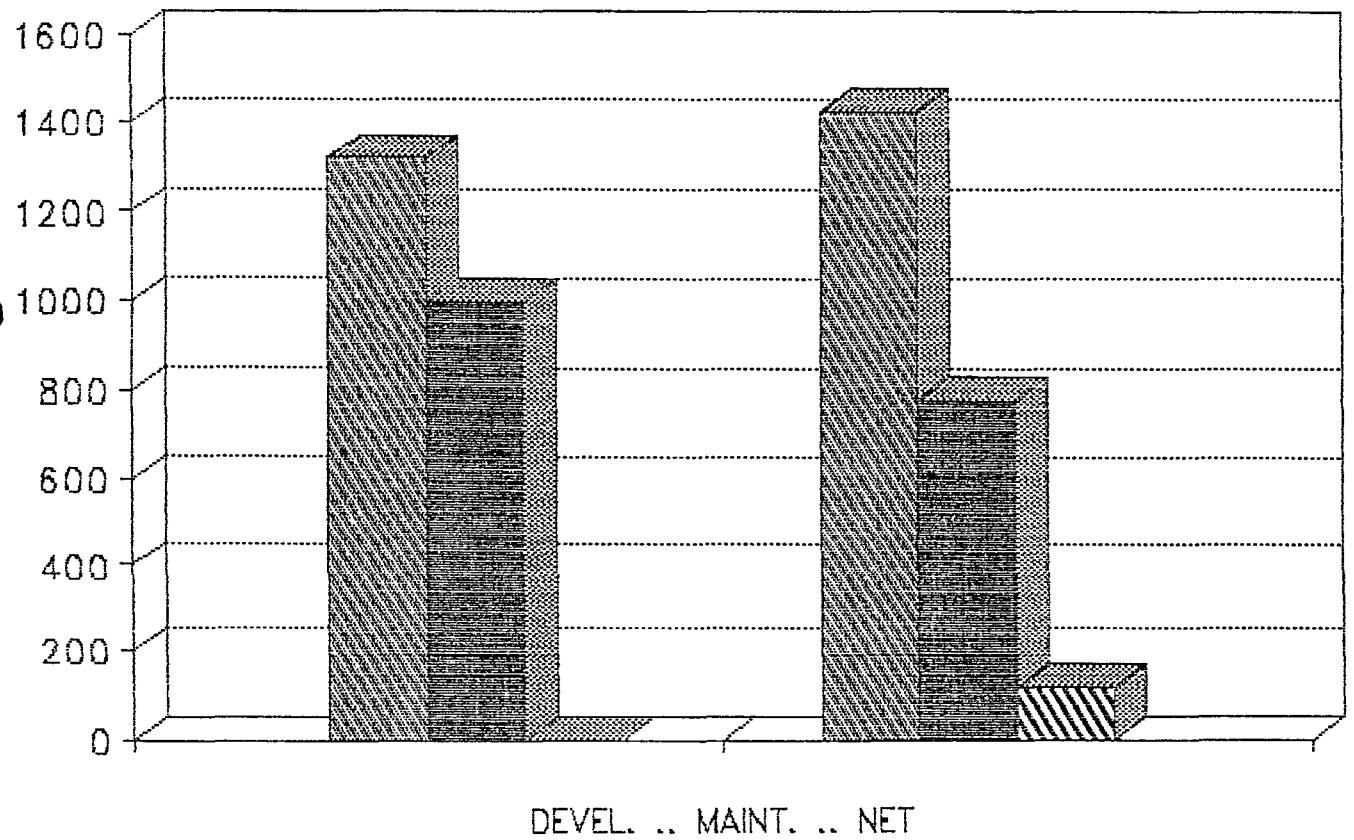


Theoretical QA Efficiency
by Defect Type



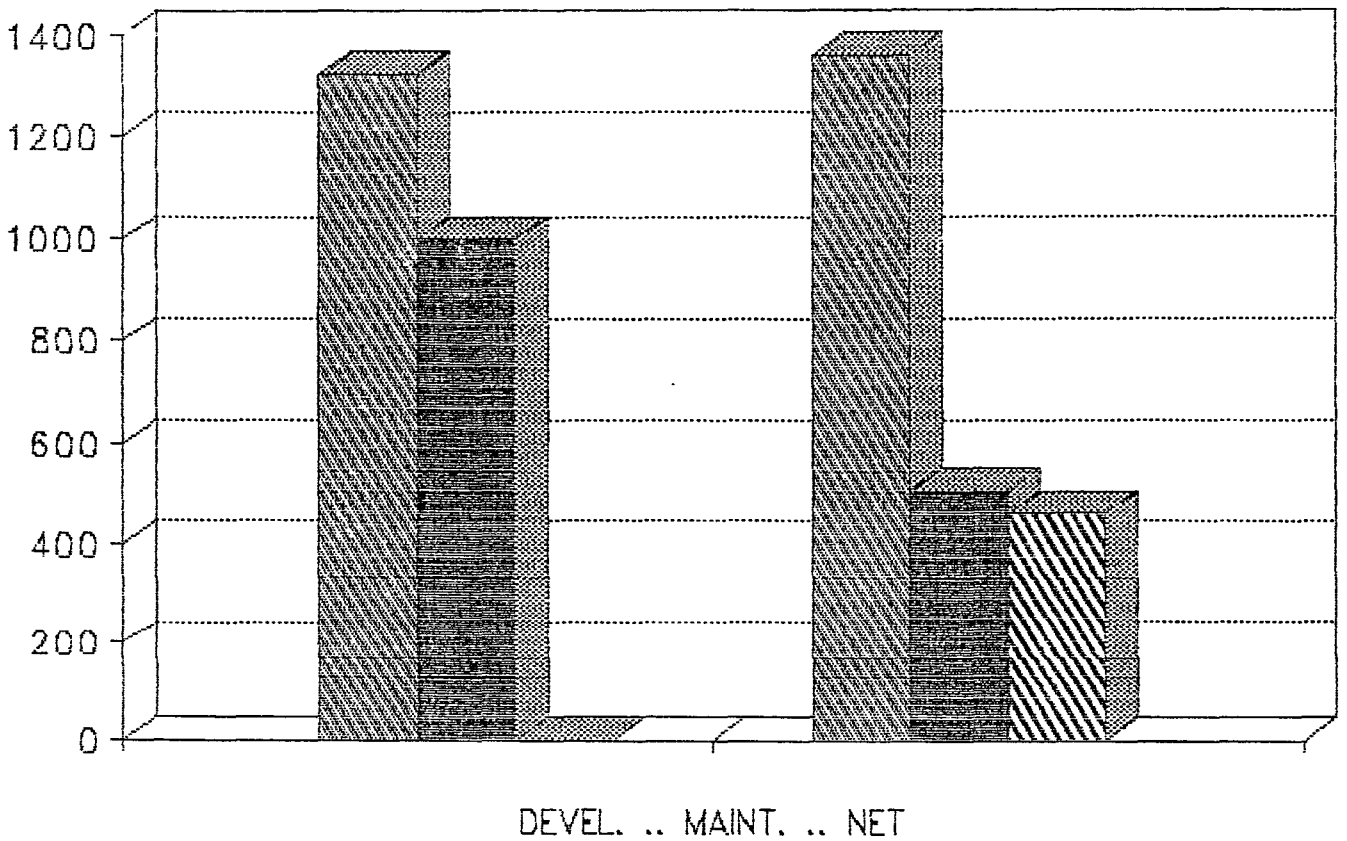
COMPARATIVE COSTS

W/O REVIEWS ..... WITH REVIEWS



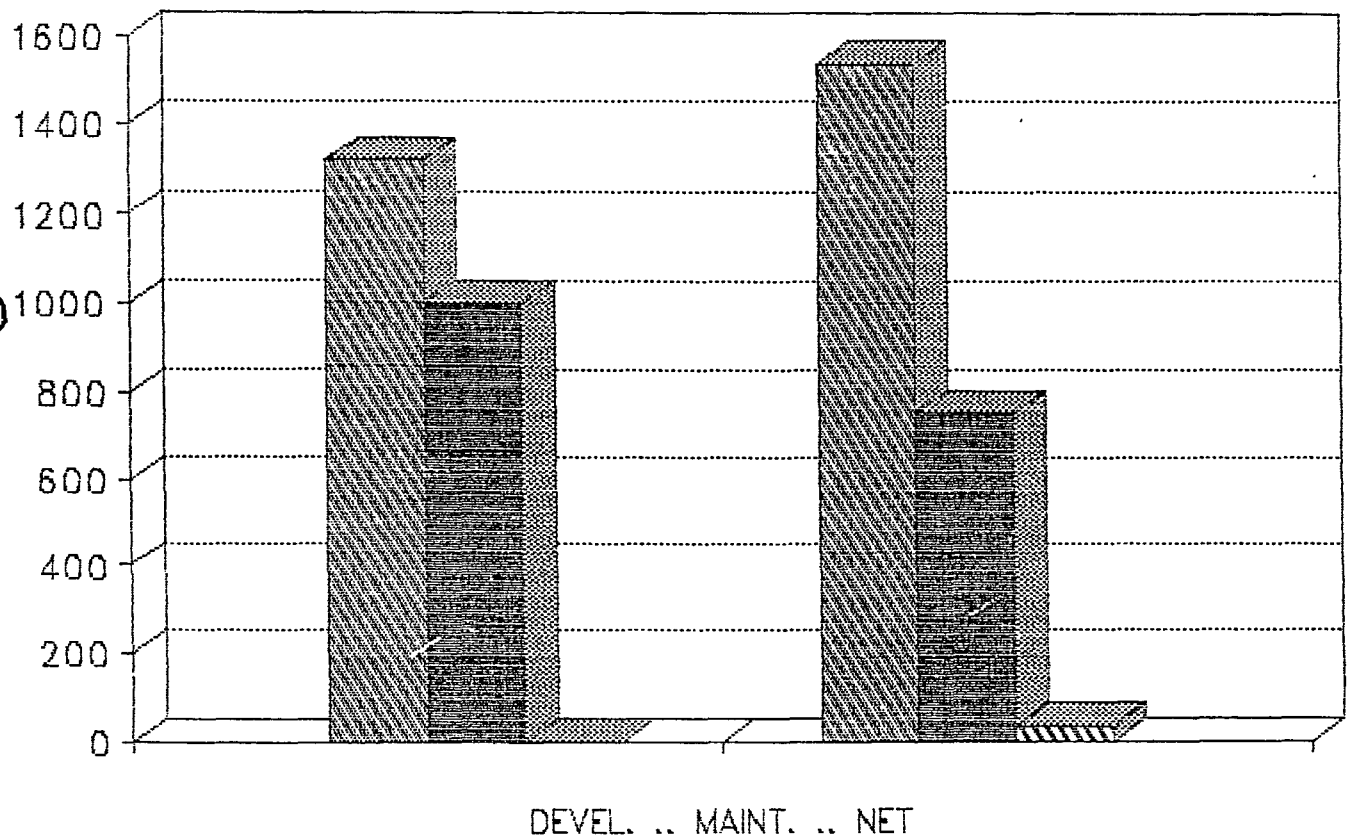
COMPARATIVE COSTS

W/O INSPECTIONS ... WITH INSPECTIONS

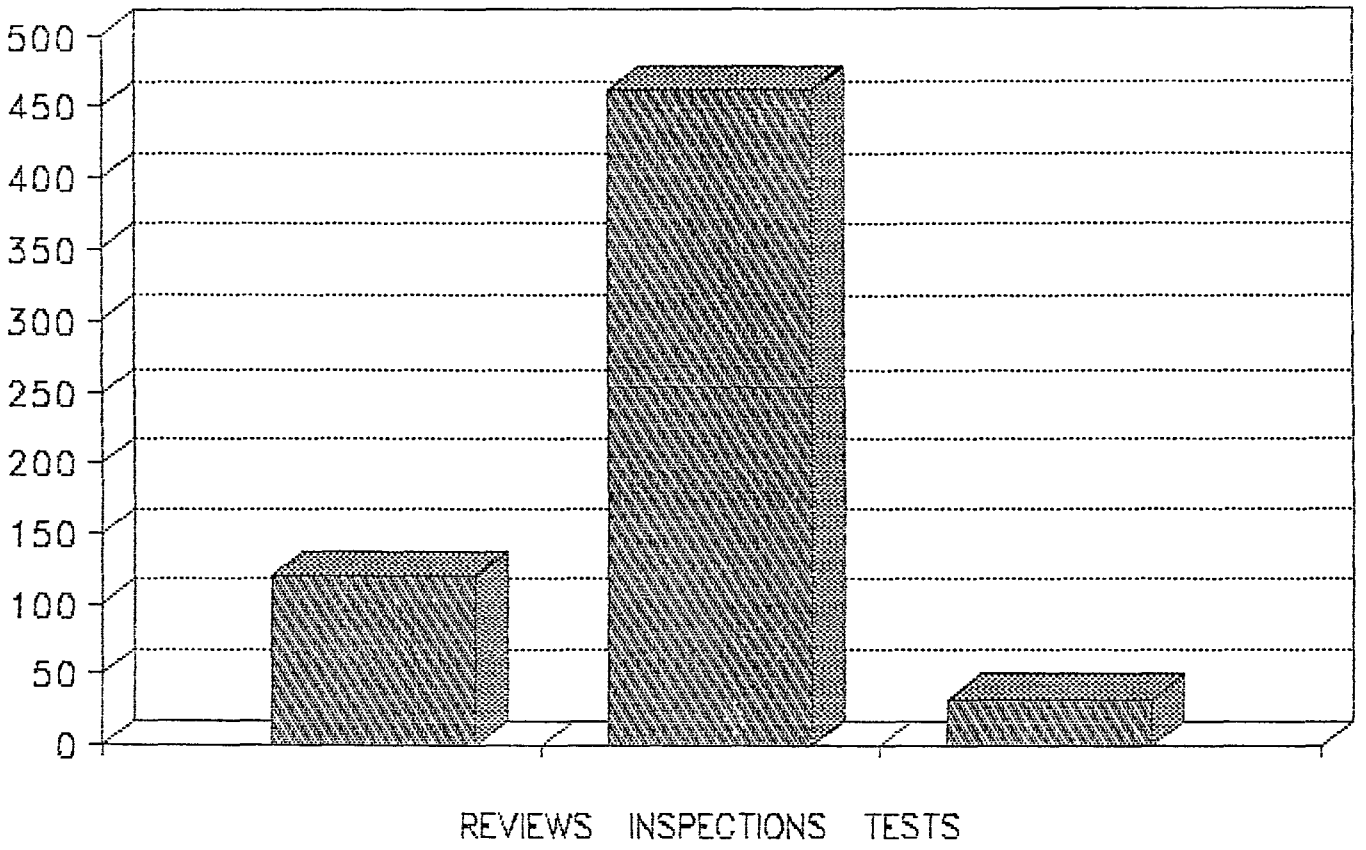


COMPARATIVE COSTS

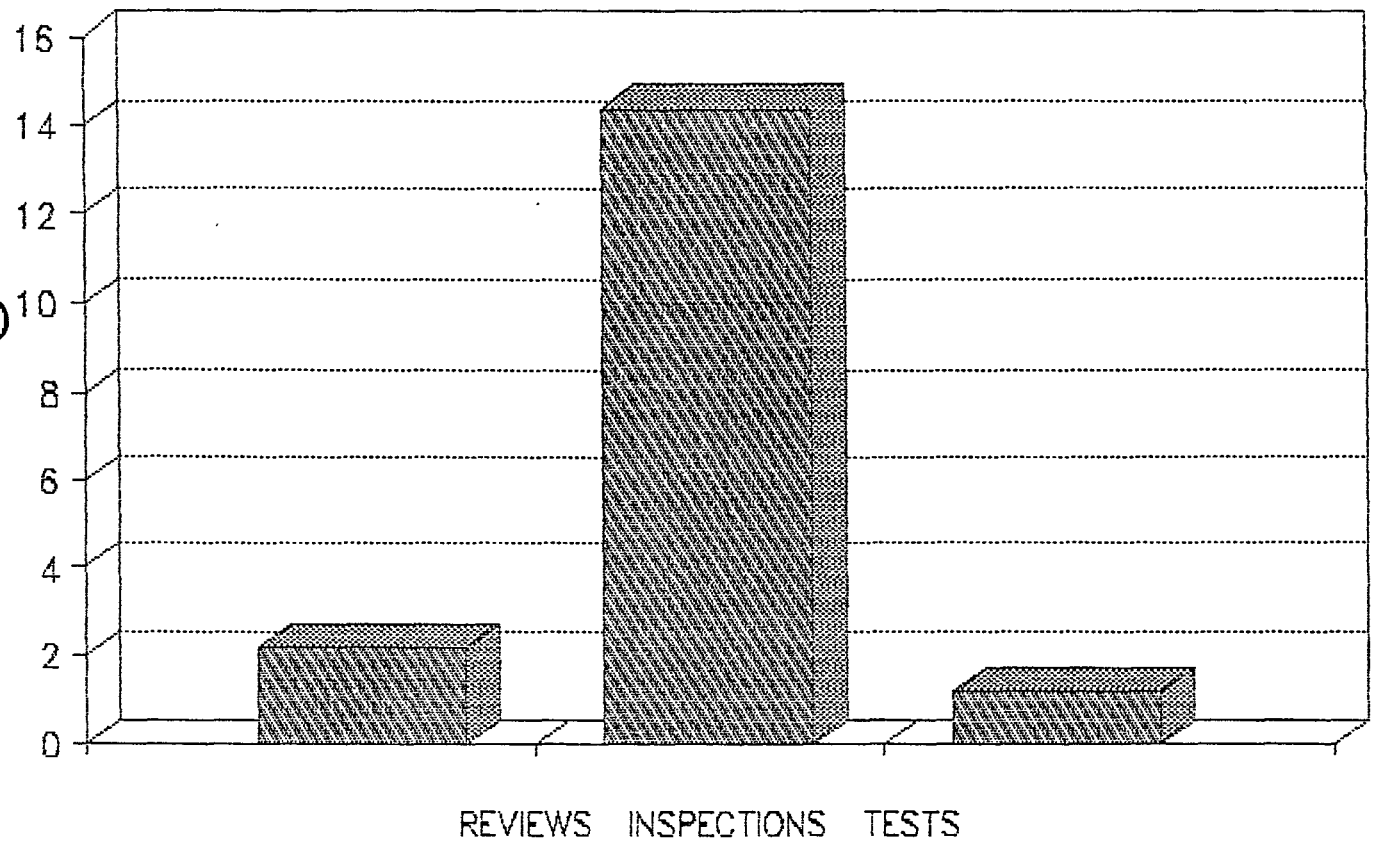
W/O TESTS ..... WITH TESTS



NET SAVINGS by
QA TECHNIQUE



RETURN ON INVESTMENT by
QA TECHNIQUE



Paper 5-M-3

CASE FOR COMMERCIAL QC: THE ECONOMICS EXPLORED

Dr. Edward Miller
President
Software Research, Inc.

Dr. Edward Miller is President and Technical Director of Software Research, Inc. (SR), San Francisco, California. where he has been involved with software quality management and high quality software engineering. Dr. Miller has worked in the software quality management field for 25 years in a variety of capacities, and has been involved in the development of families of automated software and analysis support tools. He was chairman of the 1985 1st 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.

CASE For Commercial QC: The Economics Explored

Dr. Edward Miller
President
Software Research, Inc.

OUTLINE

Major Components of CASE Systems

Integrating Systems Into the Life Cycle

Factors in Economic Assessments

Example Case Studies

Problems and Potential Solutions

INJECTION POINTS FOR QC APPROACHES

Upstream CASE

Software Requirements Analysis
System Design
Software Development Planning
Test Planning
Verification & Validation Planning
Modification/ Enhancement Planning

Midstream CASE

Software Development (Coding)
Debugging
Test Capture
Defect Reporting
Defect Repair

Downstream CASE

System Testing
Regression Testing
Defect Reporting
Maintenance Modifications
Modification Verification

FACTORS IN ECONOMIC ASSESSMENTS

Direct Costs
- Productivity Issues
- Quality Issues
- Competitive Position
- Market Forces

Indirect Costs
- Product Liability
- Regulatory Issues
- Image Loss
- Team Reorganization Expense
- Standstill Expense

Technology Injection Phenomena
- The "S" Curve
- Maximum Rate of Change
- The "Learning" Curve

THE PROBLEM WITH CASE STUDIES...

Not Believable
- By Management
- By Cost Accountants
- By Technologists

Artificial Numbers
- Hidden Life Cycle Costs
- Constructive Disregard of "Reality"
- Artificial "Economics"

Inapplicable Situation
- "Not my Situation at All"
- No Prior Experience
- "Not Invented Here"
- Unwilling to Admit Similarity
  - to Prior Experience
  - ("Bury History!")
- "Yes, But..."

The "Cooked In My Own Kitchen" Phenomena

SR CASE STUDY EXPERIENCES

What Was Done...

- Eight Case Studies Fully Developed
- Good Economic Analysis
- Nearly-Full Financial Disclosure

Constraints and Limitations...

- SR Using Own Tools
- Operating in Own Kitchen
- Evangelists for Automation?

Responses...

CASE STUDY A – DEVELOP PROGRAMMING ENVIRONMENT TEST SUITE (SR#0890)

SITUATION:

Test an object-oriented database system
Validate against formal specification
Prior field experience with product

METHODOLOGY AND TOOLS:

Comprehensive test matrix (R1 completeness)
SMARTS regression control
TDGEN generation of syntax-oriented tests
Special automated results checker (oracle)

RESULTS:

157 Test programs with 471 tests
24 Defects found
40 Hours automatic test regression time
$1K/ defect (excluding tools)

NOTES:

Major performance problems found
Stable system after one round of corrections

CASE STUDY B – DEVELOP PL/ I TEST SUITE (SR#1010)

SITUATION:

Test PL/ I compiler, G-level subset
Extensive field use
Multiple hardware systems

METHODOLOGY AND TOOLS:

Full test matrix (PL/ I language)
Validation tests for G-level subset
TDGEN generation of "random" PL/ I programs
SMARTS regression control

RESULTS:

165 Test programs, baseline output files
11,167 Lines of test control script
31 Defects found
6-8 Hours automatic test regression time
$0.5K/ defect (excluding tools)

NOTES:

Commercial release highly successful

CASE STUDY C – TEST ASSEMBLER PRODUCT (SR#0954)

SITUATION:

Test a commercial assembler (8086)
Compare against user's manual and other specs
Significant "beta site" use, repair
Regression on multiple PC architectures

METHODOLOGY AND TOOLS:

Complete test matrix (flat structure)
CAPBAK for user-interactive test
SMARTS regression control
Special oracle program for comparing outputs

RESULTS:

610 Tests
160 Defects found
24-40 Hours automatic test regression time
$0.6K/ defect (excluding tools)

NOTES:

10% Re-discovery rate on regression
Commercial release successful

CASE STUDY D – TEST HI-END PUBLISHING PRODUCT (SR#0988)

SITUATION:

Automate testing of SUN-based publishing system

Demonstrate facilities with 2-week manual testplan

METHODOLOGY AND TOOLS:

CAPBAK capability

- Mouse movements
- Window differences

SMARTS regression

EXDIFF screen comparison

RESULTS:

210 Tests built, validated
850 Subtest windows extracted
22 Defects found
$2.2K/ defect (excluding tools)
36-48 Hours automatic test regression time

NOTES:

Completion of enhanced production

CASE STUDY E – TEST UNIX OPERATING SYSTEM (SR#0877)

SITUATION:

Test UNIX/ XENIX operating system
Proprietary CPU
Kernel validation
Performance testing

METHODOLOGY AND TOOLS:

Limited test matrix
SMARTS regression tools
TDGEN generation of command variations
USVL test suites
EXDIFF output validation

RESULTS:

66 Kernel test groups
87 Library test groups
141 Utility test groups
31 Defects found
$1K/ defect (excluding tools)

CASE STUDY F – TEST XENIX OPERATING SYSTEM (SR#0795)

SITUATION:

Test XENIX operating system implementation
Multiple hardware variations
Validate against hardware standard

METHODOLOGY AND TOOLS:

Test planning from manuals
SMARTS regression support
TDGEN generation of test variations
CAPBAK for some interactive tests

RESULTS:

56 Interactive tests
404 Test groups for 116 commands
23 Compiler tests
10 Limit and performance tests
96 Defects found
$1K/ defect (excluding tools)
40+ Hours automatic test regression time

CASE STUDY G –
TEST PATIENT ORIENTED MEDICAL PRODUCT (SR#0813)

SITUATION:

Test a blood analysis program (5 KLOC, BASIC)
User-level and technical specifications
Full validation needed

METHODOLOGY AND TOOLS:

Comprehensive, hierarchical test plan
Multi-purpose "end to end" tests
SMARTS regression control
CAPBAK used for all tests
Special-purpose validation software
TCAT analysis for convergence tests

RESULTS:

39 Functional tests created
13 Module and system convergence tests
12 Defects found
$2K/ defect (excluding tools)
40+ Hours automatic test regression time

NOTES:

Prior 4-year field use of product
First regression found 2 additional defects

CASE STUDY H – TEST A MEDICAL PRODUCT (SR#1020)

SITUATION:

Test a laboratory instrument (15 KLOC, "C")
Detailed, formal specification
New software
Full validation requirement

METHODOLOGY AND TOOLS:

Detailed test plan developed from specification
Complete 5-filter QC method:

- Module inspection
- Interface/ system inspection
- Functional tests
- C1 convergence tests
- S1 convergence tests

SMARTS regression control
TCAT for C1 convergence
S-TCAT for S1 convergence

RESULTS:

38 Functional, format, error tests
13 Cosmetic tests
13 Convergence tests
74 Defects found:

- 52 During inspections
- 11 During functional testing
- 11 During convergence testing

$0.5K/ defect (excluding tools)
36-48 Hours automatic test regression time
(8-12 Hours without printer)

Paper 5-A-1

LEVERAGING QC TECHNOLOGY SUCCESSFULLY

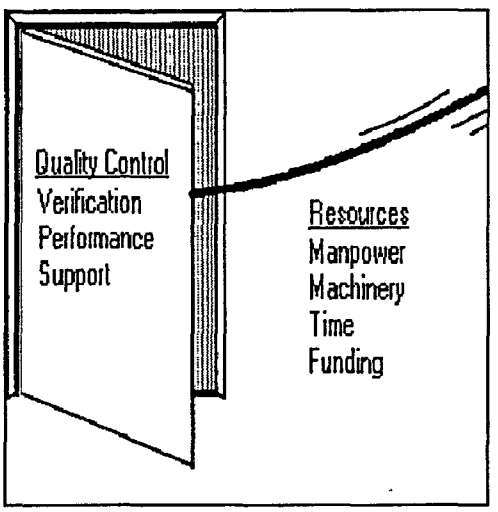
Mr. Rick Jensen
Fujitsu America, Inc.

Mr. Rick Jensen manages 3rd party software development for FMI, including Unix, Real Time OS, assembler, linker, librarians, and high level language compilers. He has been involved in system and application software design and development since 1980.

Leveraging QC Technology Successfully

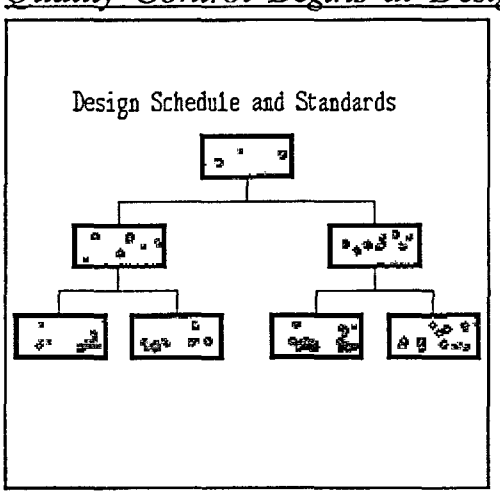
Rick Jensen
Software Project Engineer
Fujistu Microelectronics Inc.

Quality Control Resources



Although the situation may differ among private industry, academic, and government circles, software quality control organizations frequently find themselves lacking in manpower, machinery, time, and proper funding. Or, quality control responsibilities may be assigned to an organization which has no quality control experience or resources. Since quality control organizations are service oriented, cost justification can be difficult to demonstrate or explain to management. Poorly allocated quality control resources often result. This paper will offer techniques for coping with less than ideal conditions, but will focus on quality control activities rather than aspects of management relations with quality control.

Quality Control Begins at Design and Development



Know your developer. Whether development is in house or from a 3rd party, knowledge about your developer is a good beginning and will set the stage for quality control work later on. Information about the development track record, cooperative attitude, and internal quality control measures is very helpful in gauging expectations about the software which will eventually be delivered for quality control activities. What do competitors and customers say about the developer and the above characteristics? If possible, seek to have a voice in the selection of the developer as well as an active part during the development process. This may require educating management into viewing a quality control organization somewhat like a project liaison or monitor function.

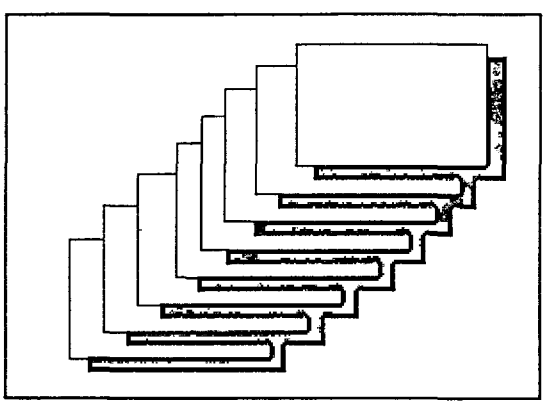
Agree on standards of acceptance. Before design or development begins determine how the developer will measure compliance. Most reputable developers are aware that software delivered to customers must meet certain requirements. Hence, built into many projects are

internal standards for product release. What are they? Work closely with the developer to establish those internal standards. Make such standards demonstrable or easy to measure. If the situation permits, arrange for periodic meetings to verify that development milestones are met and the project is in line with agreed standards.

Learn of existing standards from such organizations as POSIX, government or academic institutions. Developers frequently seek access to such software suites to exercise their products. Many of these sources have no restrictions on use and will be easy to obtain.

Exercises such as those above allow the developer to educate quality control personnel and foster a cooperative atmosphere between developer and tester. By being educated about the design and development of software, the developer indirectly provides resources and machinery which contribute to acceptance verification. Furthermore, quality control receives valuable insights about the technology and the kinds of testing needed at acceptance. Budgeting such resources is often easier to do on the development side of a project as opposed to the quality control side.

Avoid Duplication of Effort



Have the developer demonstrate acceptance. After acceptance standards are agreed upon, make it clear to the developer that delivery of software will be made after a demonstration showing that software has met internal development standards. This may require a visit to the developer's site or it may be possibly to connect remotely to the developer's system to witness this.

The object here is to:

1. Avoid spending your time repeating tests which the developer has already performed.
2. Become educated about proper use of software under the direction of the developer.
3. Allow the developer to prove that standards have been met.

Such a demonstration will also make a natural transition for use of software at the quality control site. Simple, but essential questions to the developer about:

Installation
Startup and initialization
System configuration requirements
Operating assumptions and caveats

can be answered. This avoids potentially embarrassing situations on both sides.

Whenever development goals coincide with quality control objectives, there is an opportunity to avoid duplicating efforts. If nothing else, such measures can save valuable time.

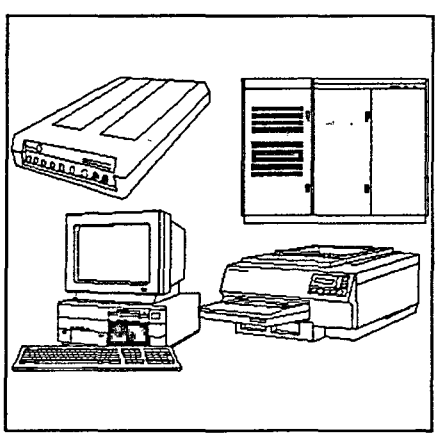
Contact other customers. Although this may pose certain problems, if it is possible to contact other customers, you may find some particularly useful test cases, learn other information about the product, or discover that extensive testing has already been conducted. Again, a duplication of effort is avoided and discoveries concerning other areas essential for a thorough test plan can be made.

Talk with a software engineer at the developer's site. Many programmers take pride in the quality of their work and are willing to talk openly about the strengths and weaknesses of the software. Knowledge about software functions which have had little or no complaints from customers or software engineers themselves will help establish areas of priorities for testing. Having a knowledge of the strengths and weaknesses of the software from the developers point of view can help with the planning and help get off to a successful start during the actual testing.

Create a Test Plan

"Practicality" is the key in making a test plan. Previous sections have dealt with activities which are preliminary to writing a test plan. Any involvement in endeavors above will acquaint the tester deeply enough with the subject to enable the creation of a sensible test plan. In some cases an outline is sufficient. In other situations, the test plan may be more extensive than the test report.

The test plan can also be a fill-in-the-blank style of test report. Some kind of system which archives and keeps test plans, notes during testing, and test reports can be a valuable resource in tracking the status quality control work. Later, such a system is a valuable source of data for documenting effectiveness of quality control efforts. Facts and figures can help make a point to management when tackling a project which is new or different.



Assemble Testing Resources

Testing resources often exceed development resources. Be prepared to spend some time on this. Tracking and verifying the quality control work will require extra disk space, printers, connections, permissions and sometimes the cooperation of a systems manager. Besides the fact that good quality control costs money, management can be caught in a catch-22 dilemma of paying a software staff to succeed in developing a product and then paying the quality control department to prove that development failed. Hence, seeking an attitude of cooperation among system managers may be met unsympathetically.

A checklist of:

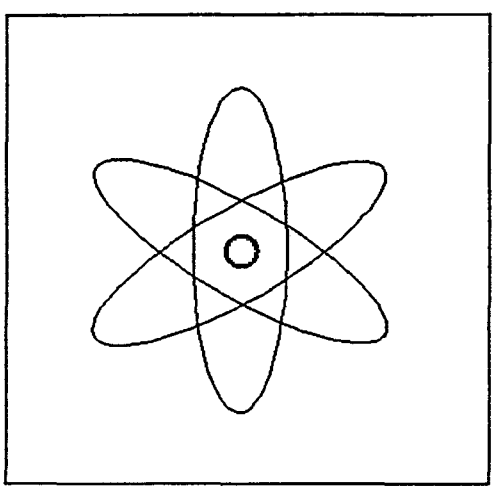
System requirements and configuration
Disk and storage space
Printers
Software and utilities
Permissions and connections
The type of acceptance report to be produced

is needed to make sure that testing can get off the ground.

Establish a Testing Schedule. Schedules are never set in concrete. Be practical and be prepared to make changes. Since resources are scarce priorities for testing the most important elements of the software must be taken into consideration. This is where the work and investigation with the developer and customers pays off. The advance work described earlier will indicate how to use quality control resources the most effectively.

Schedules also become a measurement of the efficiency of a quality control organization and provide a history of activities allowing quality control organizations to improve their effectiveness. Keeping records of schedules also provides management with evidence that quality control resources are used wisely and more should be furnished.

Identify Critical Elements of the Test Plan



Where are the dependencies and bottlenecks? This is a typical project planning exercise. Based on the expected use of the software, software testing for a particular user's needs may emphasize infrequently used portions of the software or user needs may dictate unusually extensive testing of a certain portion of the software. The identification of critical elements of the test plan often makes alternate plans and approaches obvious. Since quality control operations seldom go exactly as planned, standby options are comforting.

Classify tests into automated and manual categories. Certain aspects of testing easily lend themselves to automated testing while others must be done manually. Identify each test as automated or manual. Manual tests may require less preparation but are time intensive and have high human resource requirements. Automated tests require thorough preparations in order to obtain reliable results.

Testing Activities

Automated Tests	Manual Tests
Syntax verification	Code generation
Upper and Lower limits	Hardware dependencies
Standards compliance	Optimization techniques
Performance	User interface functionality
Special features and functions	Compatibility with other entities
Performance	On line help
Regression testing	Documentation
Reporting	
Setup and initialization options	

Take time to setup and verify automated test procedures carefully. This is an ideal opportunity to use a test which has been run at the developers site. Since results are already known, attention can be devoted to details of establishing an automated test environment and working with less known elements of the test plan and system. This phase of test preparation cannot be over emphasized. If you feel that you are spending most of the time in groundwork just to begin the testing, you are probably right on track. Great amounts of painstaking effort are needed during this time to ensure that input, output, parameters, switches, and scripts accurate perform each step of an automated procedure. Once output has been generated, attention must be given to the details of processing and analyzing the output.

The size of the quality control project and nature of the software will assist in sorting out which tests should be automated and which tests should be conducted manually. Some tests which could be automated may only need to be executed once. Spending time to setup such tests for automated execution may not be justified.

Automated Testing Capabilities

The same features that allow for development of powerful software permit extensive, exhaustive testing. Powerful software testing tools now exist which permit testing of switches, parameters, and settings in all combinations and permutations. Grammar and syntax can be tested in such a ways which probe their technical definitions endeavoring to prove compliance in actual operation. In a parser like fashion, statements and functions can be examined from simple to the most complex. By combining traditional automated procedures with new testing technology quality control organizations can address and penetrate software quality testing in numerous ways.

One conveniece of any automated test is that it can be scheduled for execution at an optimal time (i.e. overnight, over the weekend etc.)

Manual Testing

Notwithstanding the headway of automated software test tools and procedures, there remains significant quality control functions which must be conducted manually. Wise use of automated testing resources by the quality control organization can free up time to perform the manual activities of quality control.

Most of the manual activities fall into areas of analysis which require not only comparison of figures but also qualitative contrast and association. Manual quality control activities can be manpower, equipment, dollar, and time intensive, making manual acitivities difficult to manage.

Review of documentation and measuring its compliance with actual operation is an example of quality control work that must be done manually. In addition, someone with a certain level of knowledge and expertise is often required to effectively review documentation.

Conclusion

As with any "*How to* ..." advice, the situation varies from case to case. Pressures and constraints of real life conditions seldom follow textbook scenarios. The reader will note that sections on Automated Testing and Manual Testing are unusually short. Since each of these topics requires an in depth knowledge of what is being tested only general statements have been made. There are no magic formulas or shortcuts available for the actual testing.

Much of this paper has been based on a certain quality control exercise. It was initially supposed that the resources used for the actual testing and execution of software were the only quality control resources. However, during the preparation of this paper it was discovered that other activities mentioned in earlier sections played an import role in quality control work for the project. The actual testing provided the raw data from which an acceptance report was generated; but the preliminary activities revealed quality control benefits which made execution and testing activities efficient and reliable.

Paper 5-A-2

USE OF AUMTOMATED SOFTWARE TESTING METHODS ON MICROSOFT APPLICATIONS

Mr. Glen M. Poor
Member of Technical Staff
Microsoft, Inc.

Mr. Glen M. Poor is a Lead tester in the Applications Division at Microsoft. He has been working in Testing at Microsoft for 4 years, focusing primarily on the word processing applications including PC Word 4.0, PC Word, 5.0, Windows Word 1.0 and PM Word 1.0. He has been involved with automating testing tasks for the past 3 years in his work at Microsoft. He has degrees in Mathematics, Economics and Political Science from the University of Washington.

Quality Week 1990 Presentation

Use of Automated Testing Methods on Microsoft Applications

WHAT IS AUTOMATED TESTING?

Automated Testing is the act of performing some or all of the testing tasks without human intervention, usually by computer programs.

There are many tasks associated with the testing chore:

- plan which tests need to be produced
- creates those tests for execution
- manage the execution of many tests, often repeatedly.

Automated testing is designing techniques and tools to automate any of these tasks.

At Microsoft, we have done work with tools and techniques for each task. We have focused our attention on the execution of tests because this is the most expensive and repetitive, hence it is the task that is automated the easiest. We have done some work on managing the automated tests and very little on automatically planning and creating tests.

PRESENT

We have focused most of our attention on tests which are planned and produced by humans but are executed, verified and managed automatically with the tester informed of the final results, i.e. pass or fail. If the tests pass, great; if they fail then additional investigation needs to be done.

Individual testers decide what tests will be produced, which will be automated versus manual, create the test data for each test and if the test is to be automated, decide which tool to use in automating it, create the programs necessary to automate it and finally insert that automated suite into the automated test management system, if one exists on that project.

There are specific goals that we want to achieve by the automation of test execution. The most important one is to reduce the cost, in both time and money, of performing a full regression against the product as we get close to shipping. This is important if we are to keep very bad bugs from getting into the product with the final few bug fixes. A second goal is that we want to use the automated tests to store knowledge gained by a tester about the product from one version to the next; knowing that the particular tester may not be around to take advantage of knowledge gained. A third goal involves the human element; testers get burned out quickly when executing the same script repetitively. Automation not only relieves that repetitiveness to a large degree, but it provides a separate activity that testers generally find interesting.

PLANNING TESTS

At Microsoft, we have done very little work with tools that decide what tests to produce. Most of our work is functionality based testing. To automate in a significant fashion which functional tests need to be run requires a rigorous functional specification. This is not available. At best we have a functional specification describing the new features in a small

amount of detail. Because of this, deciding which tests to develop requires a high degree of intelligence which would be difficult to automate.

Currently, the testers decide which tests will be automated as part of the regression suite or part of the basic acceptance suite (generally a subset of the regression suite). The testers also investigate the code for all structural tests.

We are beginning to use tools that will help testers decide which tests need to be added to the suite. Assorted coverage and code complexity statistics are starting to be used. Information from those numbers gets fed back to the testers to improve the rigor of the regression suite. Perhaps the decision on which tests to add based on this information could be automated someday. Another source of information on potential tests come from internal and beta test sites; their reports conceivably could identify test cases that need to be added to the test suite. Again, this information is made available to the testers who decide what to add; it might also be automated someday.

We have also been thinking of more white box types of tools that could conceivably automatically produce test cases. For example, reverse engineering tools to produce digraphs and data dictionaries. Initially the testers could use the digraphs to understand dependencies in the code and design tests for them; the data dictionaries give an insight into structural tests that need to be produced. These tools could be enhanced to someday make specific suggestions for tests. For example, the reverse engineering tools that produce data dictionaries for the code can be used to generate structural tests to investigate limits and boundaries associated with a particular feature. The data dictionary can give info pertaining to data structures so that limits and boundaries are easily identified as well as make other structural tests more easily produced. At the moment, none of these tools are in use for any significant products at Microsoft.

CREATING TESTS

We break down tests into 3 fundamental parts: the data to run the test (input), the steps to execute on that data, and the results to verify against some expected result. Each one of these parts could be automated. To create a reasonable automated environment, the execution and verification steps have to work well together; in general, automation tools that have been developed provide a way of executing the test and verifying it. This is where most of the work on automation has been done at Microsoft.

Test Data Automation

Any tool or technique used to automatically produce the data to be used in a test would be considered Test Data Automation.

As with test production, limited work has been done with tools that generate test data on a grand scale. The testers are responsible for producing the test data and individual testers use different techniques and tools to produce that data when it is appropriate. We do use tools to set up specific environments. Most of the tools that we do use are normal everyday tools that we use in an innovative way to produce test data.

No tool currently exists that automatically generates test data for the majority of testing done on our products. Because the test data is so specific to the tests you want to perform, to generically automate this step would require knowledge of what tests to be run, i.e. those tools we talked about above. The output from these tools could read into some program that could be smart enough to generate actual test data for those cases.

One set of tools that we do use helps set up a particular environment for tests. For example, there is a program called eatfat which will fill up a disk leaving N number of bytes free to assist in disk full testing. This seems so trivial but it could save a tester an hour in trying to simulate different disk space available scenarios; it also makes possible the ability to test each disk available scenario in a rigorous fashion when combined with other automation techniques. A similar tool exists for setting up memory constraints. This tool saves lots of time in the DOS and Windows environments.

One technique that has been very successful in producing test data is to use the application itself. For example, there was a small PC Word macro written to produce N number of files in a particular directory structure with different text contained in the document summary section. This was then used by the File Find tests to verify the Find feature could detect different types of doc summaries in different directory structure scenarios. This is useful because the number of files can be changed easily and the files don't have to be kept around on disk someplace to be lost. Changes to these files can be made by changing the macro.

Test Execution Automation

Any tool or technique used to stuff the input channels of the application so it executes the steps necessary to carry out the test is a legitimate test execution automation technique. Frequently, the different tools that perform test execution can capture a testers input to the application and then play it back later. *In this sense, each of the techniques below can be implemented with tools that support capture/playback, a phrase often used to describe test execution tools.*

Executing tests automatically is where most of the work on automation has been done at Microsoft Applications[1]. A number of tools and techniques have been used. There are four different points where it is possible to provide input to the application: External to the computer via simulated keyboard and mouse movements; the OS BIOS; through system specific input paths; and through application macros. There are some tools that use a combination of these input points to control the execution of test cases.

Each of these techniques has risks and benefits associated with them. The largest risks to be considered are how the automation tool affects the application and if the automation technique misses something that executing through the normal interface would have found.

---

[1] This is specific to Applications because the Systems and Languages groups use only programs to test their particular products. The Systems group has used VCR to a small degree in their final systems test; it is used to drive applications that run on the system.

Obviously the more external the technique used, the better it simulates a user. However, there are distinct benefits to the other techniques as will be discussed below.

External

External based automated test execution is a technique where a separate computer system, the master, is used to drive the system on which the tests are running, the slave, by emulating the slave systems input channels, e.g. the keyboard and mouse. The master machine acts as a record/playback tool which records keystrokes and mouse movements, allows the user to edit the recordings and can play them back to the test machine. In many tools, the recording can be edited to provide conditional and control statements.

At Microsoft, the primary externally based execution tool is VCR, Versatile Computer Recorder. It is similar to other Record/Playback tools available on the market, Any one of these tools has implementation specific strengths and weaknesses; I discuss VCR's below. However, there are some generalizations we can make about this technique.

There are a number of strengths to this approach. First, it is totally independent from the application so there are no dependencies built into the automation system. It is the closest thing to a user of all of the automation techniques. As such, it is extraordinarily flexible; anything a user could do, this system could do.

A second strength is that the external system is performing separate from the slave system. If the slave system crashes, then the external system is still alive to take steps to recover. These steps can be as drastic as power cycling the slave system if the appropriate hardware is available. Since it is separate, this execution method has access to debug information if that info is generally sent to external device. This can be useful as part of verification or recovery from crashes. (It is probably true that other techniques can have access to this same information though generally they are not allowed to act as a debug terminal.)

This approach has some serious weaknesses for which some work arounds are available but annoying to deal with. The biggest problem is with the timing of the playback. Typically the playback is done using the same time between events as the recording. This is a problem because the application changes speed across different releases and on different machines so it may not be ready for keyboard input yet. If any keystrokes are lost, then the playback is off track from that point on. If the keystroke has to happen in a specific period of time, it is difficult to ensure the key will come in that time frame. There are a number of work arounds to this problem. The most common is to wait for the screen to stabilize before sending any more keystrokes. This is time consuming and won't work for keystrokes that have to happen in a particular time frame or when the screen never stabilizes. A second work around is to liberally place delays in the playback and a variation of that is called a soft delay, where each delay is increased or decreased by a factor representing the speed of that release. Again, this is not foolproof; extraordinary circumstances may cause delays to be too small and keystrokes to be lost. A third work around is used in VCR called the typing rate, which effects the speed at which VCR sends keystrokes to the slave;

a lower typing rate gets fewer keystrokes per minute. Again, no guarantee that it will solve the problem.

A second problem concerns the mouse. Due to the way the mouse works, it is very difficult to accurately record and replay mouse movements. A click 3 pixels from the original location may or may not yield the same effect and, as time goes on, the little errors add up to where the mouse is nowhere close to its correct location. Editing the recording is very difficult as it is highly dependent on the particular video it will be replayed on. The most common work around is to try to force mouse positions between each mouse move; lots of up and left moves should guarantee the upper left corner. This can be effective a good portion of the time. Another work around is to not use the mouse unless you absolutely have to. Mouse functionality is usually available through the keyboard. However, with this you're not automatically testing mouse functionality.

OS BIOS: TSR keyboard playback

The next point to stuff the application's input channel is in the BIOS of the machine. This is done by a program which emulates keystrokes by placing the right information into the interrupts of the BIOS. Some of the tools that do this are specific to testing, e.g. AutoTester by Interactive Solutions. I do not have experience with any of these tools, I suggest you contact the manufacturers for more information.

We have used the more generic tools available for IBM PCs. These are TSRs (Terminate and Stay Resident) which hook all of the keyboard interrupts to record and play back keystrokes. Keyworks Advanced Version by Alpha Software, Superkey by Borland and ProKey by RoseSoft are examples of these TSRs. Generally these tools record keystrokes, allow you to edit and replay them.

There are few strengths to this tool. They are self contained and generally easily editable. They are also quick and easy to use. We use them for really short term, very repetitive tasks.

The weaknesses are similar to the external tools. The biggest is timing. Some of the tools allow control over the rate at which the keystrokes are sent. There are much fewer work arounds with this tool than with the external tools. Another big weakness is that the TSRs are playing around with the BIOS vectors, depending on what your own Application does with the BIOS vectors, this could break your application. One weakness that is more similar to system level tools is that they require memory, hence change the memory configuration.

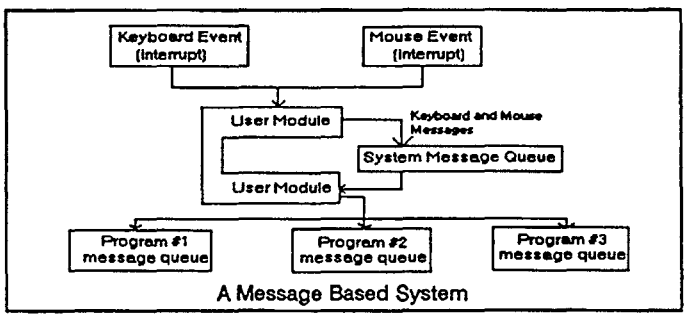
These type of tools are used rarely at Microsoft.

System: Journaling tools, input piping

The next input point to stuff for executing tests is very nebulous because it is heavily dependent on the system the application is running under. Different systems allow different techniques for sending input to applications. For example, the DOS and Unix environments allow piping and redirecting of standard input and output. The provides the

opportunity to use this feature to drive test execution if the application uses standard input and output. Unfortunately, no significant applications do this at Microsoft. We also have no experience automating through systems specific capabilities on other systems because we don't provide applications on them.

At Microsoft we are very familiar with the DOS based Character Windows, Windows, Macintosh and Presentation Manager systems. These are all event driven message based systems; the messages provide a nice system specific execution technique. The ability to record and playback these messages is what we call journaling or eventing.



A Message Based System

The diagram of a message based system at right demonstrates how the logic of the system works[2]. Keystrokes and mouse interrupts are converted to messages which are placed into the system queue and as time allows, are dispersed to the application who had the focus at that time. There are many more messages in the system than just the keyboard and mouse activity, but these messages are the most important for automating test execution. As you can see from the diagram, if you could record the messages before they get to the application, then you could play them back by placing them in the system queue. In addition, you could insert any message you would like into the queue for that application. Because the system already handles sending the messages to the application, the messages are sent at the speed the application can handle them, i.e. there is no lost keystrokes.

This is a very powerful and flexible technique. The user does not have to worry about solving the timing problems and can focus on creating the tests themselves. The particular tool can support a powerful programming language to provide conditional and control statements. It can alter the speed at which messages are placed in the queue or place a particular message in the queue at any given time. Since the tool is in the system, it can query the system for information and condition accordingly. This allows the tester to force particular situations that would be difficult to do using any other technique.

There are some weaknesses to the approach. First, the tool used has to be part of the system hence alters the very system itself. For example, the memory scenarios are different just because the tool is running. Second, the tool resides in the system so any system crashes bring the tool down; there is no chance for recovery to run other tests. This is a significant problem if you are trying to set up an automation system to get through many tests in a run; you don't want one crash to tie up a machine for a whole weekend. Some work arounds exist for this, e.g. a TSR monitoring Int 3's and rebooting when it detects one. This sometimes works.

---

[2] Diagram was taken from Programming Windows by Charles Petzold, Microsoft Press, 1988.

Application: macros

One of the most commonly used techniques at Microsoft to automate test execution is through the application's macro language.

There are two powerful features to this technique. First, macros generally operate at the speed of the application so you don't have a timing problem. Second, the language may have direct access to information about the state of the application that it can condition on. These two features make this technique very attractive.

There are some weaknesses. Foremost among them is that macros are usually too close to the application; the application may behave differently when running them. For some applications, macros have different entry points into the functional paths which in no way test whether the user interface is correct. Other applications may only change how they update the screen or respond to failures. The work around is to identify what changes in behavior for the application are when running macros and then test those manually. This can be a very effective technique.

A second though less serious problem is that the macro facility is often broken during the early development of the product. Also, there are some applications that do not have a macro language or that language has significant limits, e.g. it is not possible to run a macro when the application starts.

Combined

Each of these different techniques can work with the other techniques and when combined provide a powerful and flexible set of tools to solve almost any problem faced in executing tests. As the techniques that reside closer to the application don't have timing problems so they should be used for actual execution as much as possible. Techniques that reside further from the application are better at recovery, setting up the environment and for testing separate from the application. They should be used for these purposes and to control the other tools.

At Microsoft, we have been very successful combining the external tool, VCR, with application macros. We have also developed a tool for the Macintosh that resides externally but communicates with the slave machine's system for a combined external and system tool that is proving to be effective.

Desirable Properties

Whatever execution automation technique used, there are certain characteristics that we have found to be desirable in any tool for whichever execution technique it implements.

The first property is that the playback material be easily editable. It is rare that a recorded session will be able to be replayed exactly each time. The longer the replay, the less the chance that the replay will succeed each time. In addition, since we want the automation to last over multiple versions of the product, it is likely that small changes will be necessary to keep the script running properly and to add to it in the future.

The next property is that the tool must remain fairly independent of the application, or all dependencies are known so that the testers don't overlook problems because they assumed that the automated tests would detect the problem.

Whatever tool that is used, a method must be worked out to recover in case of a crash, assertion failure[3] or other problems. This is essential if you want to run more tests after a problem occurs.

Another desirable property is the ability to get information about the application. For example, if your application has some means of detecting when a problem is occurring, it can communicate this to your automation tool so that it can attempt a recovery.

Some applications are sensitive to the make or break of a key, e.g. some action only occurs when the key is let up, not when it is pressed. For these applications, the tool needs to have sensitivity for key down and key up events and perhaps mouse down and mouse up events.

As Microsoft sells its applications in the international market, the tools that we use have to support international concerns. This is a problem at the moment as some of our tools do not work with international keyboard drivers, or they can't reproduce international characters, etc.

Test Verification Automation

Of all aspects of automating testing, the verification is the most difficult. All verify techniques consists of comparing some piece of data extracted from the test against another piece of data that was expected. If the data compares OK, then the test passes, otherwise the test fails. The problem comes up when trying to rigorously define what you are trying to test. In our experience, a tester working with a manual script with 50 tests in it actually performs hundreds of tests when looking at the screen or file contents. If you wished to automate this, then you would need significantly more than 50 tests to do the job. Identifying those tests is difficult and designing techniques that will automatically verify the test is the hardest part.

To identify a verification technique, you just have to identify the piece of data that can be compared. I arbitrarily break verification techniques into Black Box and White Box. Black Box is that data that a user can get access to and white box is something that generally a user does not have access to.

Black Box

The primary black box technique we use is screen dumps. Screen dumps are very useful for verifying that the screen appears as it is supposed to. Screen dumps are very dependent on the video configuration. They are not easily transferrable unless converted to a device independent format. We are just now experimenting with this technology. Screen dumps are also sensitive to small screen changes, e.g. one pixel changes and screen compare fails.

---

[3] Assertion failures are problems that the application has detected with its own internal data structures.

As such it is desirable to have the screen dump utility allow for regions to ignore in the dump and compare. Screen dumps can also take up a lot of disk space.

Another common verification technique is file compares. This technique is useful but dependent on your particular file format. For example, some file formats save the date and time of the file, this obviously makes that aspect of the file different and the compare will fail. Other formats are non-unique, that is the same input data will not guarantee the same output data. The work around for all of these is to force the data into some format that is comparable.

The rest of the black box techniques fall under the general category of data compares. Where access to some form of data is allowed, it is copied and compared to an expected value. An example of this might be information that can be gained from the macro language.

White Box

There are many pieces of data contained in a program. If any one of these pieces of data can be gotten to, then it can be used in a test. Often we must ask development to provide access to data that we want to use in our tests. Generally these revolve around the state of the data structures in the program.

There is another ready source of information that can be gained from the debug terminal. If we use an external test execution technique then we might also be able to get at the debug information. This can be used for verifying tests. We are currently investigating the use of this technology.

In the message based systems, the messages themselves can be journaled and compared against in the future. There are also other system level data that could be useful in verifying tests.

Perhaps the most common form of white box test verification at Microsoft is the use of assert technology. This is driven by development and consists of placing checks for data structure validity and consistency in the code itself. If the test fails, then an assertion failure occurs, the product stops until the user responds. Some information is saved and the program can be terminated or left to continue. We have learned how to use asserts very effectively at Microsoft. One trick to be aware of is that, if your test execution is not interrupted when the assert happens, you have to place checks in periodically to see if you have asserted. If the keystrokes to continue with the assert are common keystrokes, then the keystroke may occur between the assert and when you check. Therefore, all assert response keys should be very rare keystrokes.

Desirable Properties

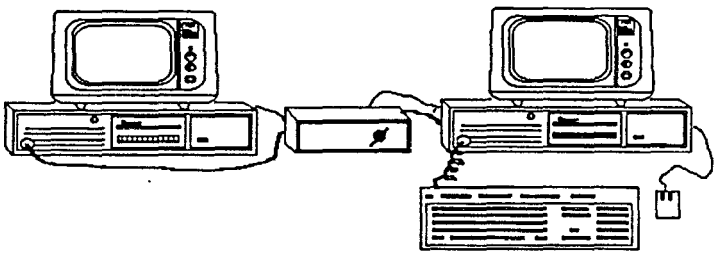
The tools that implement your verification technique should have the following properties to make it easier for the tester to reproduce problems. First, you have to be able to save the data so you can review it later and know exactly what failed. Second, the technique should be sufficiently independent from the application so that it cannot be swayed by a

bug in the application. For example, comparing data using macros relies heavily on the ability of the application to compare item successfully. This may not always work and should be tested before relying on it.

Specific Tools

VCR

VCR stands for Versatile Computer Recorder. VCR allows the tester to record keystrokes and mouse movements from one computer onto another, and to replay those actions at any time. It is similar to other record/playback tools available on the market. It works by connecting 2 computers together as in the picture at right. The computer with the keyboard and mouse is the master; it sends signals from its parallel port to the slave computer emulating a keyboard and a mouse. The slave is the computer running the tests. There is nothing in the system that knows it is running in an automated fashion because VCR is completely external to it.



A VCR Set Up

The VCR 1.0 system is comprised of 2 parts, a stand alone program that runs on the master and a screen dump driver running on the slave. Testers do most of their work with the stand alone program.

VCR has a nice set of features that make it a reasonable environment for automating tests. For example:

- It has a basic like interface with basic like control and conditional statements and subroutines.
- It allows editing of keystrokes either as text or with makes and breaks.
- It can either record keystrokes or echo them through to the slave machine.
- It records and allows editing of mouse movements.
- It allows setting ignore areas for screen dumps and compares.
- The master drives the screen dump so it can wait for screen to stabilize.
- It has a redefinable Hot key to interrupt recording.

There are some specific weaknesses to the method in which VCR has been implemented, for example:

- There is no global search and replace (because there are so many different types of entities).
- It has no true subroutines or access to OS to handle specific interrupts.
- Is too inflexible for the many different types of activities wished from it.

VCR is the tool that we have the most experience with at Microsoft. It is used in many different groups. However it is subject to the weaknesses inherent to an external system, e.g. timing. Because of this, some groups are planning on not using this tool at all in their future automation. Opting instead for system based automation tools.

Future of VCR: libraries

In order to address the 2 biggest weaknesses in VCR, editing and subroutines/OS access, VCR is currently being rewritten into a library of routines able to be called from C or Basic compilers. The testers will be writing Quick Basic or Quick C routines to do their automation. These are very flexible and powerful development environments. The libraries themselves will be available in the near future.

The advent of the VCR libraries raises the possibility of increasing what VCR can do, e.g. actually power cycling slave machine or acting like a debug terminal. We are looking forward to experimenting with this implementation.

Watt & DJT

Watt and DJT are tools that perform at the system level. WATT is designed to be used in the Windows environment and will eventually be extended to PM. DJT is designed to be used with the Character Windows development environment.

WATT

WATT is implemented through a Dynamic Link Library (DLL) that is callable by any Windows program that supports a DLL to send keystrokes and mouse movements. It has some very nice features like:

- All mouse movements are supported;
- Speed in which application receives keystrokes is adjustable;
- Timer events allow user to specify an event to occur in a specified number of seconds;
- International characters are supported.

This tool has just become available so we only have limited experience on using it.

DJT

Since Character Windows is not an environment that runs separate from the application, but rather a series of libraries that the application links with, DJT is another set of libraries that must be linked in order to work. This brings the tool very close to the application that is being tested. DJT knows about the application and about CW so it sends messages to the application through the CW libraries to simulate keystrokes.

SnapShot

Snapshot is an interface into a second set of DLLs that support screen dumps in the Windows and PM environments. It is designed to work closely with the WATT DLLs. It is the means that screen dumps and compares can be done in this environment. It also has a list of nice features like:

- Allows users to dump full or partial screens via the mouse;
- Stores screens in Device Independent Bitmap format so that all windows Video modes are supported;
- User can add, delete, view or compare any set of screen shots;

- Comparisons are done on the fly against live video memory;
- Dumps and Comparisons can be using window handles such that only the window area is used.

Marionette

Marionette is an automated testing system for the Macintosh. It performs many tasks in the organization and implementation of automated test suites, including:

- It simulates user interaction either through real time recording, or through the creation of intelligent scripts.
- It provides a means of creating a library of tests that can be used as an overnight test suite, or for tests to be used on demand as areas of a product change.
- Marionette provides ease and speed of use, allowing testers to automate their work throughout a project. This increases the value and reduces the cost of regression tests.
- Marionette is made up of several modular levels of software that can be configured for various automated testing situations.

Marionette is a tool designed to emulate user interaction with a Macintosh. This is accomplished by using one Macintosh (the *master*) to control another (the *slave*).

Marionette Slave

The Marionette Slave is an INIT (startup document) that takes control of the Macintosh's methods of obtaining input from the user. The slave software runs unobtrusively in the background of the slave, allowing the Mac to be used normally when it is not being used for automated testing.

The Marionette Slave waits for commands sent across Localtalk from the Marionette Master. To communicate, the slave and master need only be connected to Microsoft's Localtalk network.

Marionette Master

To conform with Microsoft VCR's movement toward automation interface libraries, the Marionette Master has been implemented entirely through external functions. These external functions (called XCMDS) can be infaced into many Macintosh applications and programming languages, including:

- Hypercard. The external functions are easily called from a Hypercard script. Together, Hypercard and Marionette create a flexible, easily configurable automation environment.
- Think C or MPW C. The XCMD interface is easily called from standard C programs on the Macintosh. Automation scripts could be written in C when greater speed or flexibility is required.
- Microsoft QuickBASIC. QuickBASIC is another example of an environment easily interfaced with Marionette .

- Microsoft Excel. Theoretically, even Excel with its macro language and ability to call code resources, could easily be interfaced with Marionette to control another Macintosh.

When the Marionette external functions are called, they translate the desired commands into messages across the Localtalk network. These messages are received by the Marionette slave and translated into actions which are equivalent to those of a user.

DESIGNING AUTOMATED TEST SUITES

We have talked about some of the tools and techniques used in automating the actual performance of tests. This section is going to discuss some of the issues surrounding creating such automated tests.

The first issues is that you probably want to run bunches of individual tests consecutively thus forming a test suite. At Microsoft, we write tests to run in suites. There are many consequences to this.

First is that the tests have to log informatively such that you can understand where a problem occurred during the automated suite. It can be very expensive to track down a bug that the automated suite detected, the more information available the easier it is to track down what happened. The more information you store, the more space it takes. You should also log enough information to know if tests have passed.

When tests are grouped into suites, you have to remember to check for crashes or asserts periodically. The more often that you check, the slower you go but the easier it is to determine where the crash or hang happened; referring back to the frequency of logging info. If you have crashed or asserted, then you should have some type of recovery plan if you want to run additional test suites on that machine via a test management system.

When test are grouped into suites, you probably want to check for an unknown situation periodically to make sure the application hasn't strayed from the expected path. If you have strayed, then some type of recovery has to be made.

When tests are grouped into suites, then they have dependencies on preceding tests. This dependency has some implications for recovery schemes because you can't necessarily jump into the middle of a suite and expect it to run. Designing suites to do this can be very expensive; we tried to do design our tests so we could jump to each one and ran into development costs of 3-4 times greater than just getting a suite to run. In addition, for test management schemes, it is just as difficult to ask for a particular test to run without its surrounding suite. Therefore we have decided that it is better to manage via the test suite, i.e. that is the smallest unit that can be run. A single test is not run all by itself, the entire suite is run.

Recovery schemes are important consideration when designing automated test suites. A recovery scheme should somehow recover from some test that has gone amiss and start testing again. Crashes, asserts and unknown state are the types of things a recovery scheme

needs to be able to handle. The recovery scheme we use most often is to log the problem and abort the suite. This may seem rather feeble, but since we are aiming at regression testing where in the end we expect the tests to pass, it saves us a lot of time trying to write more intelligent suites. Occasionally, we will try to recover a known state and continue, but this is rare.

Another note about automated suites is that problems detected by them have to be recreated by tester to make certain it was not a fault of the automation or other external event, and generally to find the simplest case. The cost of doing this is often not counted when sizing the automation chore. This can be very expensive and in some cases impossible if there is a severe timing issue involved. These types of problems are common in distributed applications like mail or distributed databases.

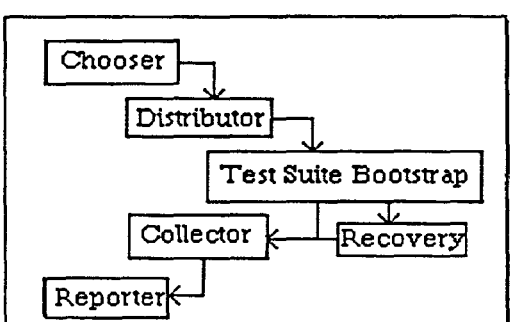
MANAGING TESTS

Conceptual

Up till now we have been discussing what it takes to automate just one test, or a bunch of tests together. When this is done, we are going to want to manage lots of tests and control when they execute, what they execute on, etc.

The test management systems we have designed work to distribute tests across machines on a network. My comments below concern such a distributed system. Other systems could use similar concepts for a single machine.

The complexity of the system you would want to install really depends on how you want to set it up and on how much information you want to have after the tests are run. The heart of the test managing system surrounds the steps necessary to execute a bunch of tests one after another. The graphic at right identifies the parts that we think are necessary in a test management system.



Flow of Test Management System

Chooser: the chooser chooses which tests are to be run and possibly which machines will run them. There could be a lot of different input into deciding which tests to run. For example, configuration might be important, or period of time since test was last run, or priority set by individual tester. The chooser section that you design should handle any criteria that you think is important. However, that criteria must be made available to the chooser. For example, the configuration of the machines running the test is important to many of our tests so part of the information made available to the chooser is the configuration of each machine.

Distributor: the distributor distributes the tests to be run to the machines that will run them. All of the material necessary must be downloaded; if an external system is used, then material has to be distributed to both systems.

Test Suite Bootstrap: this is the batch file that the automated systems runs which starts the test executing. Generally each test suite has to provide its own.

Recovery: if a reboot or power cycle is necessary to recover, then part of the recovery scheme has got to be placed into the system restart sequence.

Collector: the collector collects the log and other information from the test suite and copies up to some central location for use by the testers.

Reporter: the reporter parses the log information and attempts to find tests that failed. It can also gather other statistics like number of tests that have passed/failed. This is really an optional stage but one that we generally include to save the tester from parsing through the logs all of the time.

The system should be built to be very modular so you can replace portions with other module that will do the job better.

This system can be made more complex by handling more types of information. For example, if the tests also generated code coverage numbers, then the collector would have to collect that information and some facility would have to be made to place it in some kind of report format and database. If your tests are cataloged in a database, then a front end needs to be provided between that system and this test execution management system.

LESSONS LEARNED

One lesson learned is that part of our definition of testability for a product asks if we can automatically test the feature with the current tools available. If we can't, then some strategy needs to be developed so that we can.

Recording keystrokes and playing them back requires special care. It is not a panacea yet it can be a time saver. Keystroke recording is generally very messy hence more difficult to maintain. Recording requires less experience with the application to remember all of the keystrokes necessary for some action. However, trying to remember all of the keystrokes necessary to get to a particular feature is not always easy. Recording can help in these situations.

When the product is less stable, it makes life very difficult for automated systems. First, it becomes very difficult to develop the automated systems as you have to test to see if the automated sequence is working properly and if the program crashes you can't tell. Second, you can't get very far into the suites when the program crashes which means you can't run the tests after that point; we call this being blocked. Third, things tend to get automated which can get automated, i.e. testers kind off give up on automating a buggy section of the product so that the final coverage on that area is not always very good. For this reason, big

automation pushes usually wait for the product to reach a certain level of stability where the benefits realized exceed the time costs to get them working.

Stress testing can be easily performed by simply repeating a suite of tests repeatedly without actually having to go in and change that suite.

Monkeys can be written with any of the execution techniques above. They can be left to run to find crashes but the crashes are generally difficult to recreate. A monkey provides another means to feel more secure about the state of the product. We often write smart monkeys that run for days when we think the product is close to shipping.

When designing automated suites, keep them very modular and provide lots of APIs to the interface. Interfaces change and if every test uses the same function to open files, then only that function has to change when the interface changes.

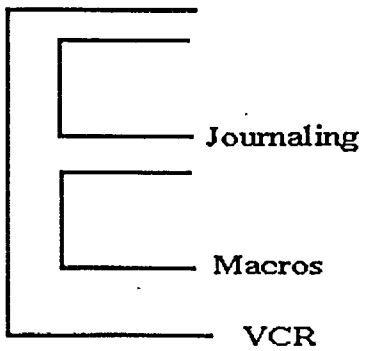
The automated test suite can also be easily used for an acceptance role. Development can use particular subsets when dealing with volatile code to make sure obvious problems do not arise before testing releases.

Testers are not the only ones interested in automation technology and tools. Anybody who has to perform the same tasks repetitively is interested in the execution aspect of the test tools. For example, our computer based training group uses some of our tools in developing their own products. The usability group uses VCR to record every keystroke a user performs while trying out new software. Our development groups are interested to the extent it can help them perform sanity checks on new code and perform some of the performance tests and benchmarks.

FUTURE

Automating our testing work is a high priority at Microsoft. We feel that it is the only means by which we can keep up with the ever increasing complexity of our applications and still staff at a reasonable level. Not everybody at Microsoft agrees with this. There is considerable resistance to change and with good reason. Microsoft Applications has been one of the most successful software development groups up to now using the current system, why should we change? However, the testing groups recognize the extent of the changes going on and are convinced that automation will play a key role in testing our applications into the future.

Right now many of our test execution tools are very new to us. We have been trying these out slowly but don't have a lot of experience with them. But as we have seen some of the tools strengths and weaknesses we are going to try the strategy in the picture at right. Basically the most external tool will be used to drive the more internal tools. The external tool will be the primary recovery tool because it has the most control over the recovery systems. The journaling and macros will be used to actually perform the tests. This takes advantage of their execution strengths. New verification schemes are being dreamed up everyday and slowly being put into practice.



We are working hard at developing better test management systems. We want to be able to track lots of different information across many different releases. This information includes number of tests passes/failed, code coverage by release, history of tests, lines of code by procedure, lines of code changed by procedure, code complexity by procedure by release, tests to bugs links, etc.

In the far future, we will probably be looking at trying to automatically create tests as was discussed above.

For internal paper, remember the problem associated with the distributed aspect of our applications. Perhaps this should go into the presentation?

OTHER SOURCES OF INFORMATION

ASTE Review, Volume IV No. 1, 1rst Qtr, 1989. This journal version focuses on the topic of Regression Testing Technology and Software Tools. It has articles on many of the common tools, i.e. PC -- Traps, Automator, DEC/Test Manager, AutoTester, CAPBAK and SMARTS.

HAZARD ANALYSIS MINIMIZES LIABILITY

Mr. Lewis Bass
Law Offices of Lewis Bass, Inc.

Mr. Donald F. Costello is President of Costello & Associates, an International Management Consulting firm specializing in Information Systems Development. He has been a faculty member of the University of Nebraska for many years and has consulted through his firm with over 130 companies and organizations throughout the world.

Hazard Analysis Minimizes Liability

Medical device manufacturers are feeling the pressure to ensure the safety of their medical software as never before. Increasing scrutiny of medical software by FDA and recent changes in product liability laws that can exact fearsome penalties from manufacturers marketing products with software deficiencies are forcing device firms to make sure their product software is bugfree—before it leaves the factory.

The best way to accomplish this, according to Lewis Bass, an attorney and safety engineer in Mountain View, CA, is to identify—and correct—software glitches through hazard analyses conducted during software development. Software deficiencies have become the Achilles' heel of many sophisticated medical devices. According to Bass, of the 41 products recalled by FDA in 1986 because of software problems, 37 (90%) were recalled because of preproduction defects—mistakes that a hazard analysis program would have spotted before the devices came under government inspection.

Bass, who serves as a software safety consultant for the medical device industry, predicts that medical device manufacturers applying for product approval will soon face an even tougher FDA, one that will take a much closer and harder look at medical software. "The accident a few years ago with a Canadian radiation accelerator that resulted in the deaths of several people because of a software problem really shook up FDA and the medical device industry to the fact that inadequate software can kill," he said. "It was the Three-Mile Island of the medical device industry, and FDA quickly realized that it lacked guidance for either the manufacturers or itself for the regulation of medical software. That situation is rapidly changing."

Bass noted that FDA is developing an internal course for software safety to better train its staff in reviewing medical software. "They are already taking steps to train their personnel to become more knowledgeable in medical software, which means more knowledgeable reviewers are going to ask tougher and more appropriate questions," he said.

But manufacturers wanting to implement software hazard analysis programs have few authoritative places to turn to for help: little in the way of governmental standards or guidance in this important area exists. According to Bass, the only established system for assessing software safety was developed by the U.S. Department of Defense (DOD) in 1987.

Contractors designing software for DOD, the Federal Aviation Administration, and the National Aeronautics and Space Administration must adhere to the rigid and minutely detailed DOD specifications. Most other companies are on their own. "Most people in the entrepreneurial world have to kind of wing it," Bass said.

In addition to the pressures of complying with ever-tightening government regulatory requirements, manufacturers of software-driven medical devices also face the specter of economically crushing lawsuits. Recent changes in liability laws have permitted plaintiffs to recover huge sums from manufacturers. "When there is an injury, almost inevitably in the United States there is a lawsuit against the manufacturer," Bass pointed out. "This is particularly true in the medical area. Physicians and hospitals have been effective in lobbying to get their liability limited. Several states even have caps on the amount of recovery against hospitals or physicians, while the liability against the manufacturer is still unlimited. So, the multimillion-dollar lawsuit is going against the manufacturer of defective software."

Bass also noted that U.S. firms that export their products overseas are at considerable risk for software defects. "Strict liability (liability without fault) is now in Europe," he said. "Under the European Economic Community Product Liability Directive, liability without fault was adopted by the Common Market about two years ago and is now being implemented by member countries. U.S. firms can now get nailed by a strict liability suit in Germany, Switzerland, or the United Kingdom."

But protecting themselves from lawsuits while at the same time appeasing FDA does not have to be inordinately difficult for medical device manufacturers, Bass noted. Rather than the elaborate and cumbersome hazard analysis program required by DOD, one designed for the medical device industry could be much simpler, but still effective. Basically, such a program for medical devices should simply be error tolerant, according to Bass. "It should have enough links in the fault tree that you don't get a death or serious injury," he said.

Such a hazard analysis program would prevent errors inherent in the software, including defects in its manufacture and logic mistakes in the code. Other methods for ensuring safety include designing software that does not depend on the user's compliance with instructions, warnings, and procedures. Properly written software may need to override the user, locking him or her out of certain functions to prevent injury. Another safety measure could be programming to automatically limit the level of radiation emitted by a particular device.

Manufacturers must also bear in mind that software is often hardware dependent. "The most common mistake in the design of software is that the program assumes the hardware will always work," Bass pointed out. "But disks fail, sensors fail, batteries wear down. You have to design so that even when hardware fails, the system can react to it."

While manufacturers may be reluctant to devote the resources necessary to implement and perform safety analyses, Bass noted that the prevention of a single catastrophic failure can more than justify the cost and time of a hazard analysis program. "A dollar invested in software safety now can save thousands in legal defense," he said.

Paper 5-A-4

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

Mr. Tim Braithwaite
Group Manager
Chartway Technologies

Mr. Timothy B. Braithwaite has over 25 years of increasingly more responsible Automated Date Processing and Telecommunications management experience. He has managed data centers, strategic planning and budgeting organization software development organizations and major computer acquisitions. Due to his special expertise in computer systems security and privacy he was appointed systems security officer for the Social Security Administration. He served as special consultant to the Privacy Protection Study Commission and in a private capacity as a system security consultant to the industry. Before joining Chartway Technologies (formerly, SAGE Federal Systems, Inc.), Mr. Braithwaite was the director of Information Services for the Bureau of Alcohol, Tobacco, and Firearms where he was responsible for the day to day management integration of computer center operations, software development and telecommunications network support. Mr. Braithwaite is currently a Methodology Security Analyst responsible for insuring that all projects conform to security/ internal controls guidelines and regulations. Assures incorporation of such guidance into the IDDM and into quality assurance and test programs.

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

Chartway Technologies

I. Security, Internal Controls, and Quality Assurance

A. Functionality, performance expectations, and qualitative measures are descriptive aspects of software to be developed; and form a major portion of the specifications against which quality assurance efforts determine the success of a project.

B. Security and internal controls are also features that describe aspects of software and are rightly within the province of a quality assurance program.

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

Chartway
Technologies

II. Statement of the Problem

A. Making automated information systems reliable and secure has been a major objective of both the private and public sectors for years.

B. Over the past decade, much direction in the form of security guidance and internal control requirements has been published.

C. "Still agencies have not made good progress in taking corrective action on many material weaknesses in internal controls and accounting systems, many of which are longstanding problems." (1)

D. And, "the situation for systems now being developed appear equally bleak. GAO recently reviewed the nine most critical automated systems under development and found they lack adequate security measures." (2)

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

III. Certification and Accreditation

A. Increasingly, management certification is being required before operational use of a system. Additionally, periodic recertification is often required.

B. The certification statement is an official document that records an explicit acceptance of responsibility for the security and internal controls of a computer application system.

C. The Certifying Official is responsible for evaluating the certification evidence, deciding on the acceptability of application software safeguards, approving corrective action, and signing the certification statement.

D. What constitutes certification evidence? Documentation that adequate security and internal controls have been specified at system inception, properly designed and programmed, thoroughly tested and implemented; and that changes to the system cannot undermine the integrity of the software. (3)

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

Chartway Technologies

IV. ENTER: Validation, Verification, and Testing (VV&T)

A. Validation determines the correctness of a system with regards to it's requirements; verification checks for internal consistency during implementation; and testing monitors system behavior when the system is executed with test data. VV&T applied to security and internal controls becomes an evaluation tool and documentation evidence for certification.

B. BUT: How do security and internal control requirements get identified and specified?

C. Evidence would indicate that they DON'T and you can't VV&T something that doesn't exist in the specification.

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

V. What security and internal controls should be identified and how should they be stated.

A. There are five security and internal control objectives that, if met, should satisfy all likely undesireable events and effects.
1. Data integrity
2. Applications software integrity
3. Data confidentiality
4. Applications software confidentiality
5. System availability.

B. Determine threat scenarios
1. What system resource is being attacked or compromised?
2. What weakness would permit the attack?
3. How would it be accomplished?
4. What type of security safeguard or internal control could be employed to prevent or reduce the loss?
5. What is the likelihood that this scenario would work and the probability that it could occur?

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

C. Other considerations
The following items must also be considered throughout this analysis.
1. Source data accuracy
2. User identification and authentication
3. Restricted use of system resources
4. Separation of duties
5. Audit trails

D. The next step is to determine the types of security feature or internal control that should be incorporated into the application system. Such controls can be categorized as: controls over input, controls over processing, controls over output, and measures to insure availability.

1. "Probably the most difficult problem facing the systems analyst . . . is how to identify controls efficiently in an existing system or in a system that is under development." (3) The selection of safeguards and internal controls is a trade-off between adequacy/effectiveness and efficiency.

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

Chartway
Technologies

2. REPEAT - Requiring the systems analyst to identify security and internal control features poses a difficult problem.

WHY?

3. Because, they are not trained in this type analysis and do not know which security and internal control features are adequate or effective in a given situation.

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

VI. What can be done to address this problem?

A. Gather the appropriate skills, experiences, and knowledge together to design the system.
   1. Systems analysts
   2. Users
   3. Security specialists
   4. Auditors/internal controls experts
   5. Functional area experts
   6. Others

B. Choose a development methodology that integrates security and internal control specifications into the design of the application system.
   1. PCMI/PCIE Model - 1988
   2. GAO Model Framework - 1989
   3. Others

C. Choose a development methodology that treats VV&T as an integral part of the development effort.

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

Chartway Technologies

D. Since CASE improves front-end requirements analysis, choose a methodology that utilizes CASE to facilitate, design, and development of software.

E. Since certification requires evidence of having defined, designed, developed, and tested security and internal control features; the methodology must provide traceability and sufficient testing to give such assurances.

TESTING APPLICATION SOFTWARE SECURITY AND INTERNAL CONTROLS

Chartway
Technologies

VII. How would it work?

IX. Summary and Discussion.

(1) Model Framework for Management Control Over Automated Information Systems - January 1988 - President's Council on Integrity and Efficiency - page 1

(2) Ibid, page 2

(3) Alan Brill, Building controls into Structured Systems YOURDON Press, 1983

Paper 5-A-5

XgA: A REMOTE TERMINAL EMULATOR HARDWARE INTERFACE

Dr. Antonio Serra
Vice President
ASIC s.r.l.
Torina, Italy

Mrs. Marina Cereser
Marketing
ASIC s.r.l.
Torina, Italy

XgA PROJECT

REMOTE TERMINAL EMULATOR
HARDWARE INTERFACE

APPLIANCE AREA

SOFTWARE TESTING
PERFORMANCE EVALUATION
TELEDIAGNOSTIC
REMOTE MAINTENANCE

XgA APPLIANCES

XT/AT, MCA, EISA BUS PC

GRAPHIC AND ALPHANUMERICAL PROGRAMS

THE XgA STRUCTURE

FLEXIBILITY AND MODULARITY

A SOLUTION SUITABLE TO THE WHOLE APPLIANCE FIELD

DIFFERENT ENGINEERING SUITABLE TO THE DIFFERENT SUT ACUSTOMIZATION REQUIREMENTS

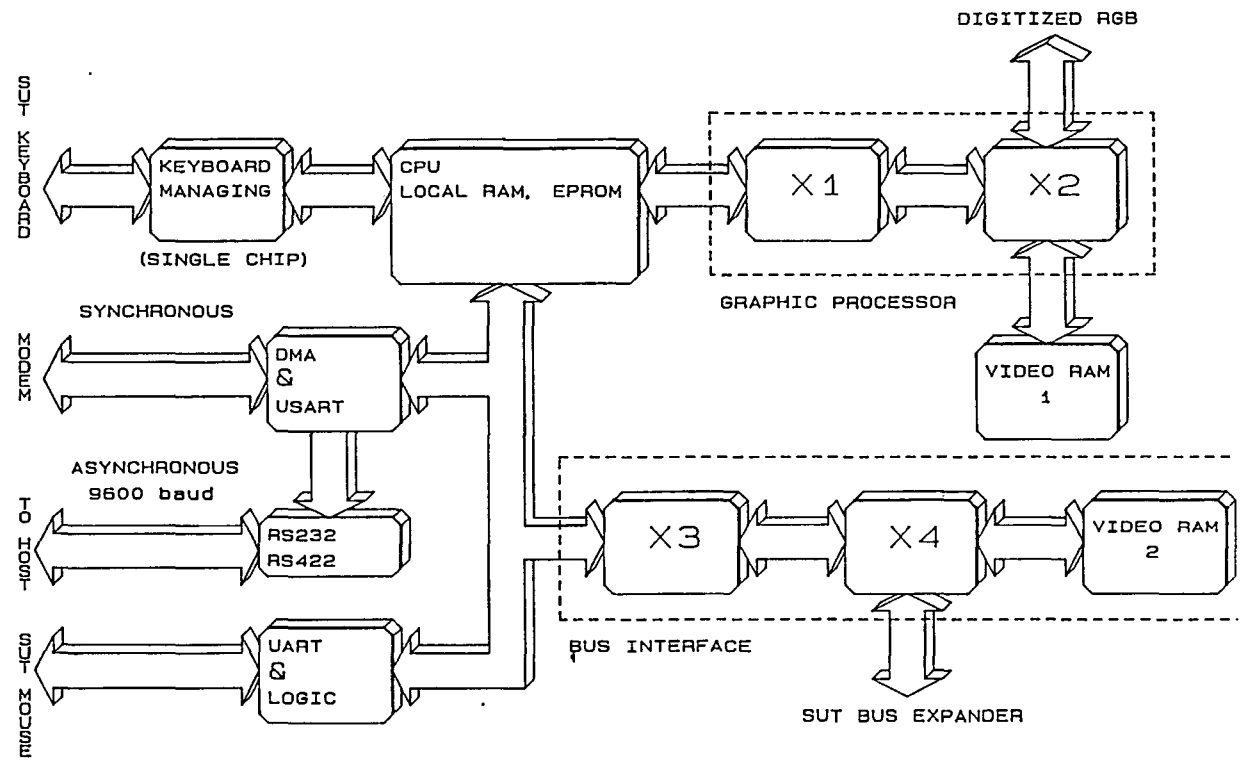
XgA

GENERAL FEATURES

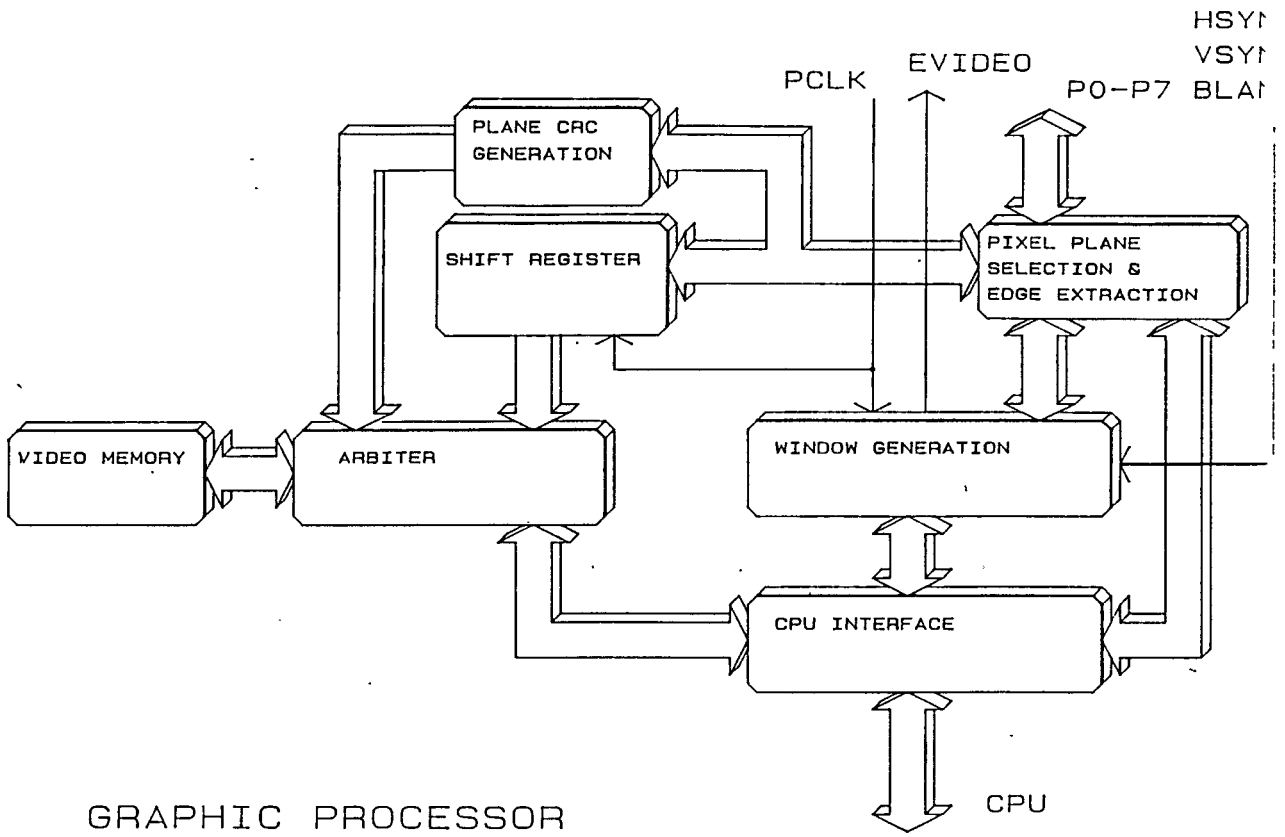
NOT INTRUSIVE SUT ACCESS
KEYBOARD/MOUSE EMULATION
CODED KEYBOARD/MOUSE PROTOCOLS

THINK TIME, KEYSTROKE RATING
AND MOUSE SPEED ENTIRELY PROGRAMMABLE

MOUSE MANAGEMENT FUNCTIONS:
FAITHFULL REPLICA
INTERPOLATION
SAMPLING



XGA, BLOCK DIAGRAM



GRAPHIC PROCESSOR

USER EMULATOR

THE SYNCHRONIZATION PROBLEM

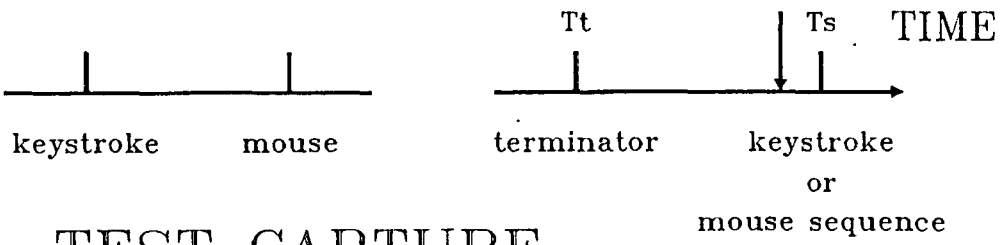
USER ACTIVITY DIAGRAM

INTERACTION SEQUENCE

ACTIVE PHASE
1. INPUT DATA SEND
2. TERMINATOR SEQUENCE

WAIT PHASE
3. PROCESSING START
4. PROCESSING END

"CAPTURE & PLAYBACK" SYSTEM DIAGRAM



TEST CAPTURE

- CAPTURE KEYBOARD/MOUSE SEQUENCES
- RECOGNIZE TERMINATOR
- WAIT FOR NEXT KEY OR SEQUENCE
- CAPTURE THE SCREEN CONTENT

GENERATION BY GUIDED SYNCHRONISM SEARCH

1. THE USER RECOGNIZE THE PROCESSING END FROM THE SCREEN CONTENT

2. THE USER INDICATES, USING KEYBOARD OR MOUSE THE SCREEN WINDOW TO BE CONSIDERED AS SYNCHRONISM

GENERATION BY MANUAL SYNCHRONISM SPECIFICATION

SYNCHRONOUS WITH TERMINATOR

THE SYNCHRONISM WINDOW IS ASKED
BY THE GENERATOR

ASYNCHRONOUS

UNDER USER REQUEST
THE GENERATOR CAPTURE THE WINDOW SYNCHRONISM

XgA GENERATION

XgA CONSIDERS AT LEAST 2 ACTIVE GENERATION MODALITIES

AUTOMATIC MODE

VIDEO CHECKSUM + EXTRA INFORMATIONS

MANUAL SYNCHRONISM INFORMATION

THE USER POINTS OUT THE SCREEN WINDOW
TO BE CONSIDERED AS SYNCHRONISM

MIDDLING GENERATION ADVANTAGES

AUTOMATED + WINDOWS SYNCHRONISM

EXTREMELY SIMPLE TEST GENERATION

UNFAILING PLAYBACK

USING THE WINDOWS, IT'S POSSIBLE TO FIX A SYNCHRONISM POINT IN EVERY MOMENT

POWER TO GENERATE WITHOUT STATING ANY TERMINATOR

XgA STRUCTURE GENERATION AND SYNCHRONISMS SEARCH BASED ON THE RGB SIGNALS PROCESSING

ADVANTAGES

SCREEN CONTENTS STORAGE NOT NECESSARY

LOW COSTS AND LOW OBSTRUCTIONS

HIGH SPEED DURING THE SYNCHRONISM SEARCH

HIGH P.E. EFFICIENCY

FURTHER ADVANTAGES USING THE RGB SYNCHRONISM SEARCH STRUCTURE

INDEPENDENT OF THE VIDEO RESOLUTION

HARDWARE MANAGEMENT OF THE EXCLUSION WINDOWS

KEYBOARD

THE KEYBOARD PROTOCOLS MANAGEMENT DEPENDS ON A SINGLE CHIP

DETAILS :

PROTOCOLS MANAGEMENT FROM AND TO THE KEYBOARD

MOUSE

WORKING PROCEDURE

TRANSPARENT

EXACT PLAYBACK OF MOUSE SHIFTS

SAMPLING

THE MOUSE SHIFTS ARE SAMPLED
WITH PROGRAMMABLE "SAMPLE TIME"

ALLOWS A REDUCTION OF THE STORED INFORMATIONS
OF A 4–8 FACTOR

MOUSE: GENERATION AND INTERPOLATION MODALITIES

THE MOUSE RUN DURING THE GENERATION IS TURNED INTO A
PIECEWISE–LINEAR

THE PIECEWISE–LINEAR COORDINATES ARE GENERATED WHEN:

1. THE MOUSE KEYS ARE PUSHED OR DROPOUT
2. OVERFLOW ON THE X OR Y AXES
3. THE KEYBOARD KEYS ARE PUSHED OR DROPOUT

ALLOWS A MOUSE DATA REDUCTION
OF 50 – 100 TIMES IN COMPARISON WITH
THE TRANSPARENT MODE

MOUSE

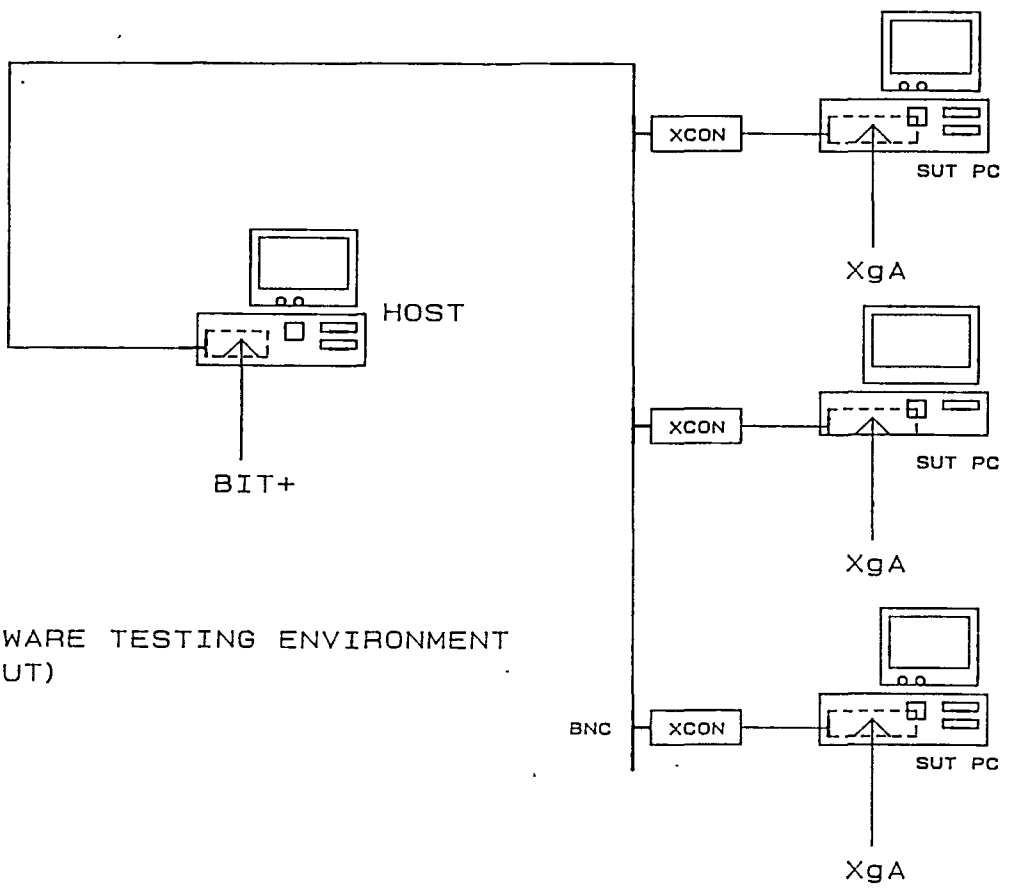
ADDITIONAL FUNCTIONS

A) "DOUBLE CLICK" TIME REPRODUCTION

B) MANAGING OF THE WINDOW SYNCHRONISM SEARCH

C) AUTOMATIC MOUSE IDENTIFICATION

D) EXECUTION WITHOUT MOUSE



SOFTWARE TESTING ENVIRONMENT
(3 SUT)

XgA TESTING SYSTEM ARCHITECTURE

THE TEST EXECUTION IS MANAGED BY THE HOST COMPUTER

CO-ORDINATES THE NODES ACTIVITY

MASTER-SLAVE STRUCTURE

PERIPHERAL NODES FUNCTIONS

"PART PROGRAM" (SCRIPT) LOAD AND EXECUTION
SCREEN COMPARISON AND TIMES TESTING

HOST FUNCTIONS

SCRIPT "DOWN-LINE-LOADING"

NODES DATA ACQUISITION
PARTIAL/FULL SCREEN
COMPARISON RESULTS
PERFORMANCE RESULTS

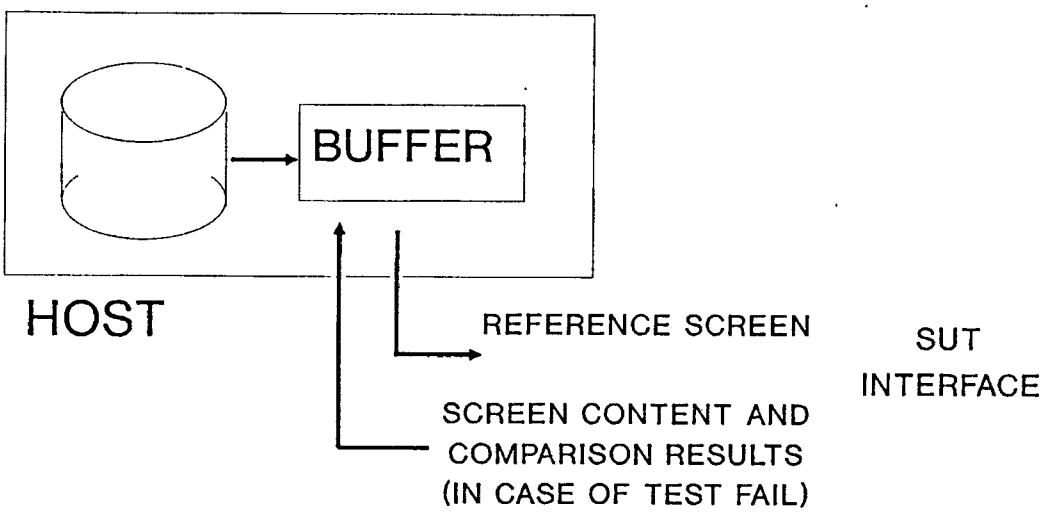
SCRIPT AND SCREEN DATABASE MANAGEMENT

SCENARIO MANAGEMENT

PERFORMANCE RESULTS ANALYSIS

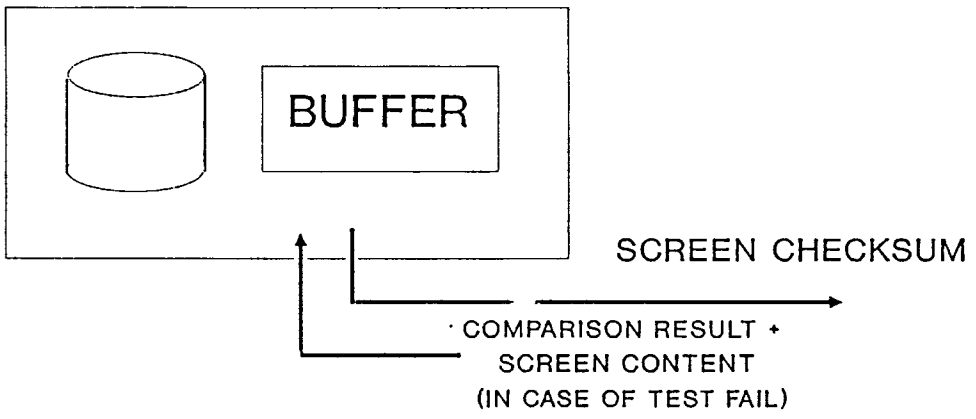
SOFTWARE TESTING APPLIANCE

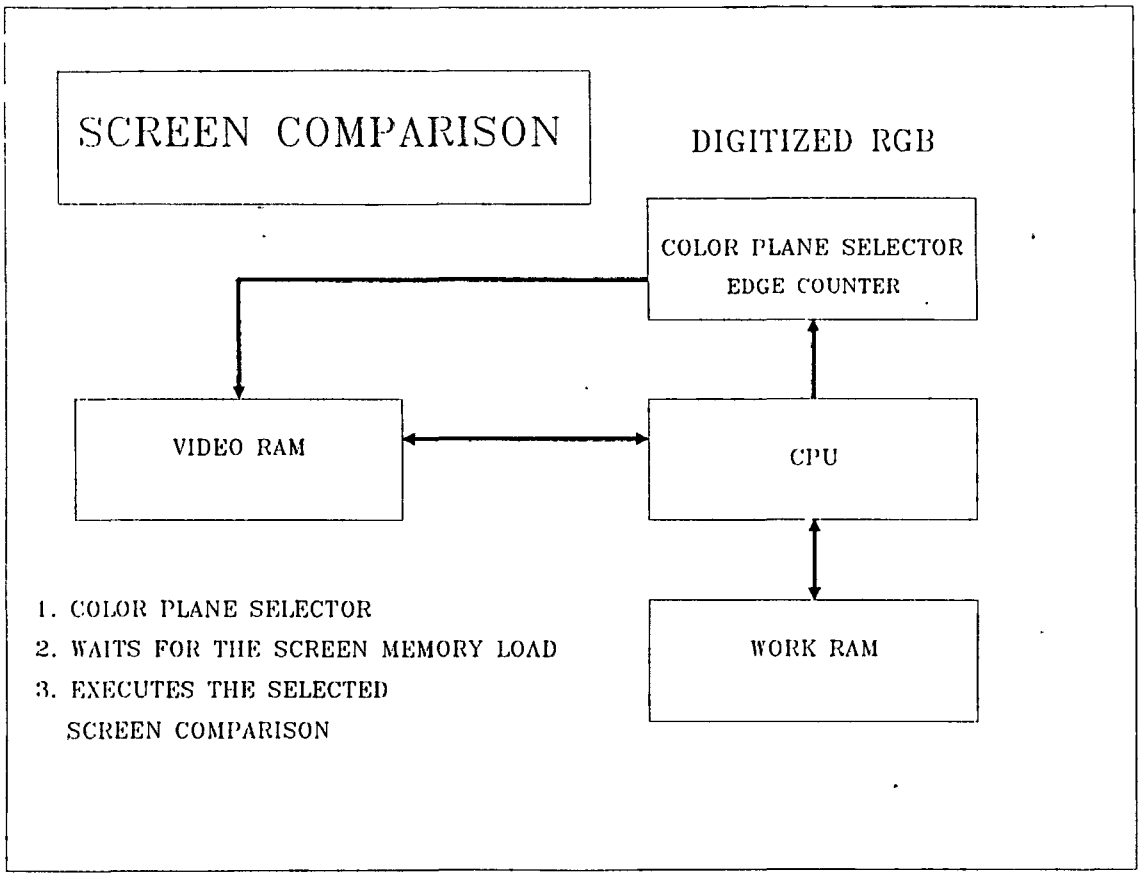
DATA FLOW



SCREEN COMPARISON MANAGEMENT USING THE "CHECKSUM VIDEO" TECNIQUE

DATA FLOW





QUANTITATIVE EVALUATIONS

HYPOTHESIS :

SUT NUMBER : 20
TRASMISSION SPEED : 2Mbit/sec.
SCREEN SIZE : 640*480
HD ACCESS : 1200 Kbytes/sec.

A) ENTIRE TRASMISSION OF THE REFERENCE SCREEN
TmR* = 200"

B) COMPRESSED REFERENCE SCREEN :
B/N REDUCTION + COLOR PLANES CHECKSUM
TmR* = 25"

C) COLOR PLANES CHECKSUM TRASMISSION (300 Bytes)
TmR* = 0,2"

* MINIMUN TIME BETWEEN TWO SCREEN COMPARISON ON EACH SUT

TECHNOLOGY
LOGIC FUNCTIONS ACHIEVED USING "LOGIC CELL ARRAY" XILINX

ADVANTAGES :

MONOBOARD ENGINEERING
NO JUMPERS STRUCTURE
EASY INSTALLATION
HIGHER RELIABILITY

HARDWARE MODIFICATIONS RETROFIT

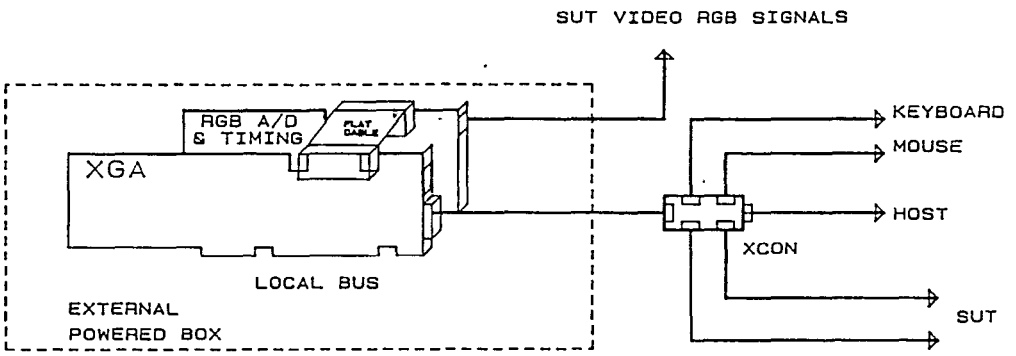
CUSTOMIZED DEVELOPMENTS

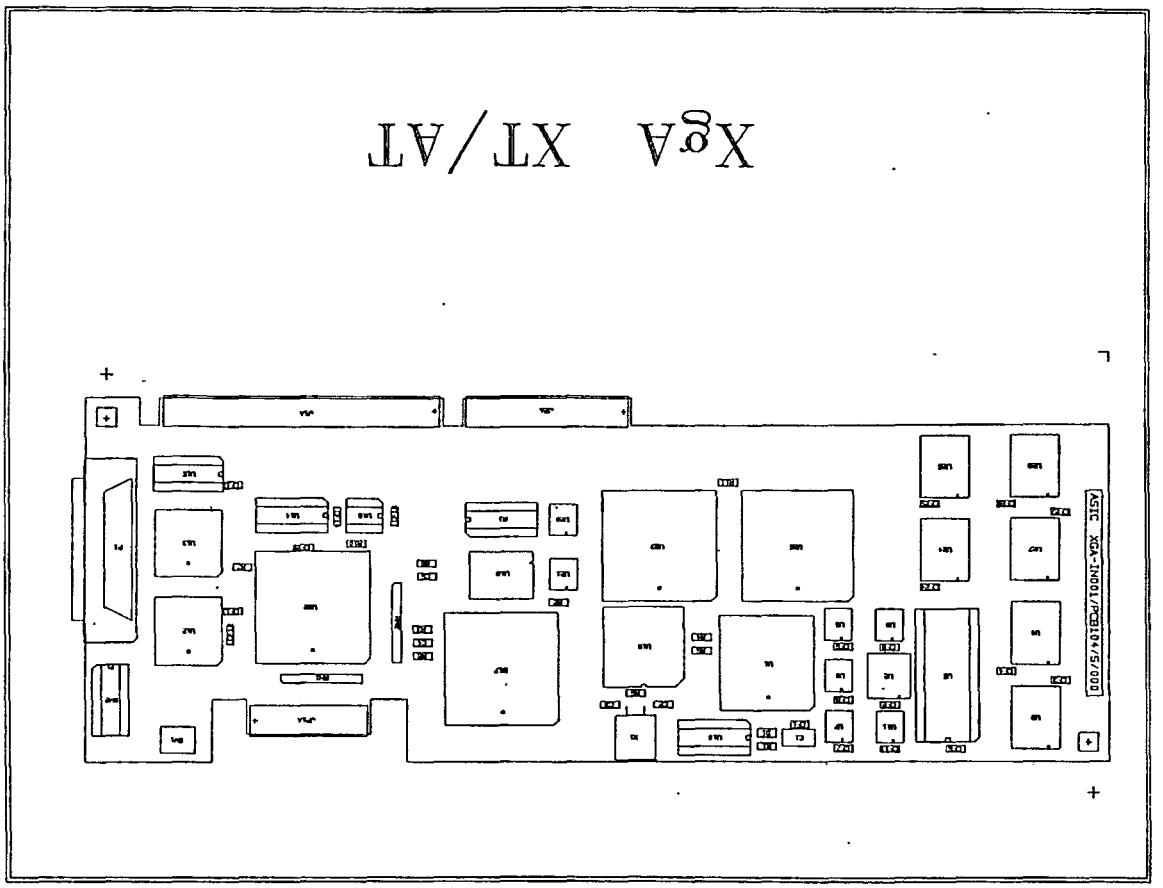
TECHNOLOGY

SMT
MULTILAYER

ADVANTAGES :

LOWER OBSTRUCTION
NOISE IMMUNITY
AUTOMATIC MOUNTING
HIGHER RELIABILITY


SUT VIDEO RGB SIGNALS
RGB A/D
& TIMING
FLAT CABLE
XGA
LOCAL BUS
EXTERNAL
POWERED BOX
KEYBOARD
MOUSE
HOST
XCON
SUT


XgA XT/AT

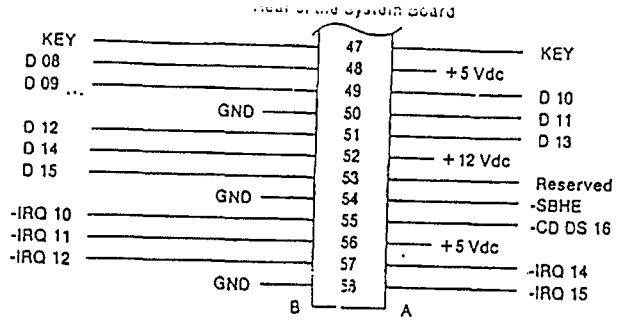


Figure 2-4. Micro Channel Connector 16-Bit Extension Voltage and Signal Assignments

Micro Channel Connector (Auxiliary Video Extension)

This connector extends the 16-bit Micro Channel connector to accommodate video adapters that interface with the system board video subsystem.

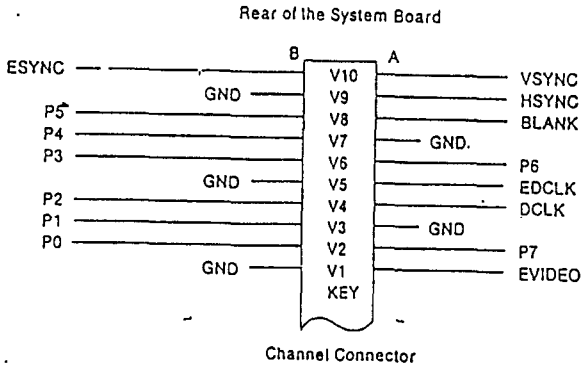
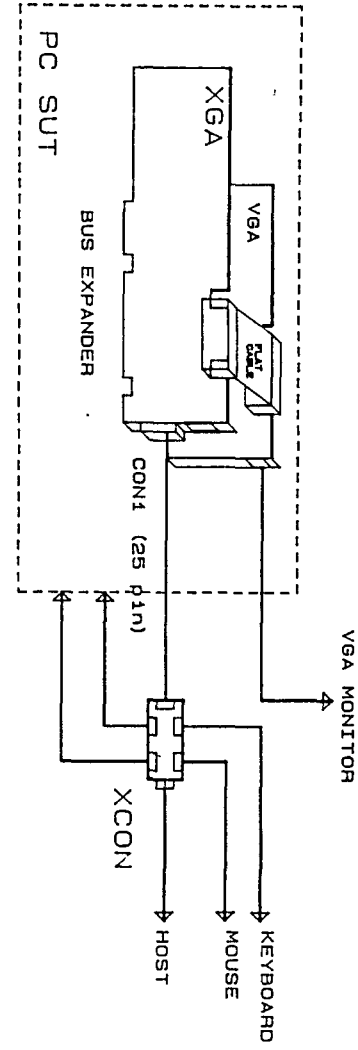
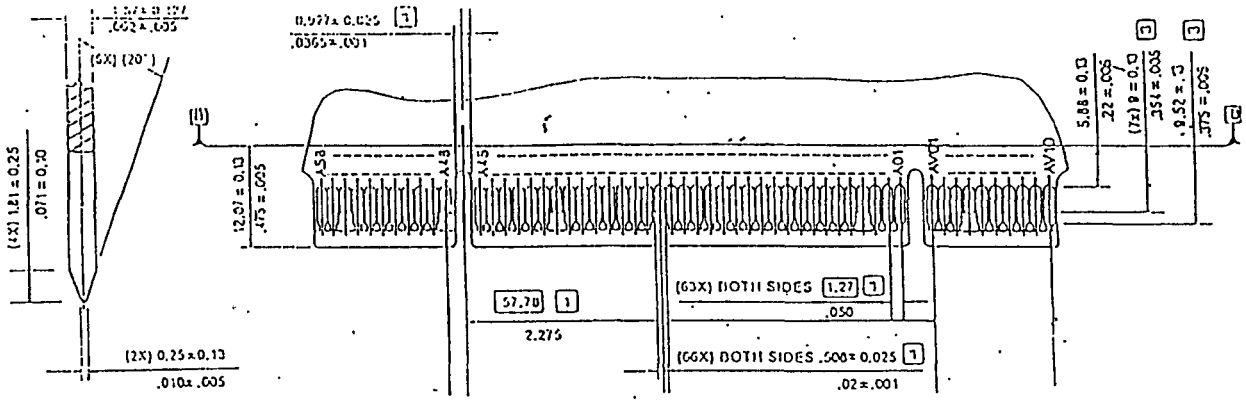
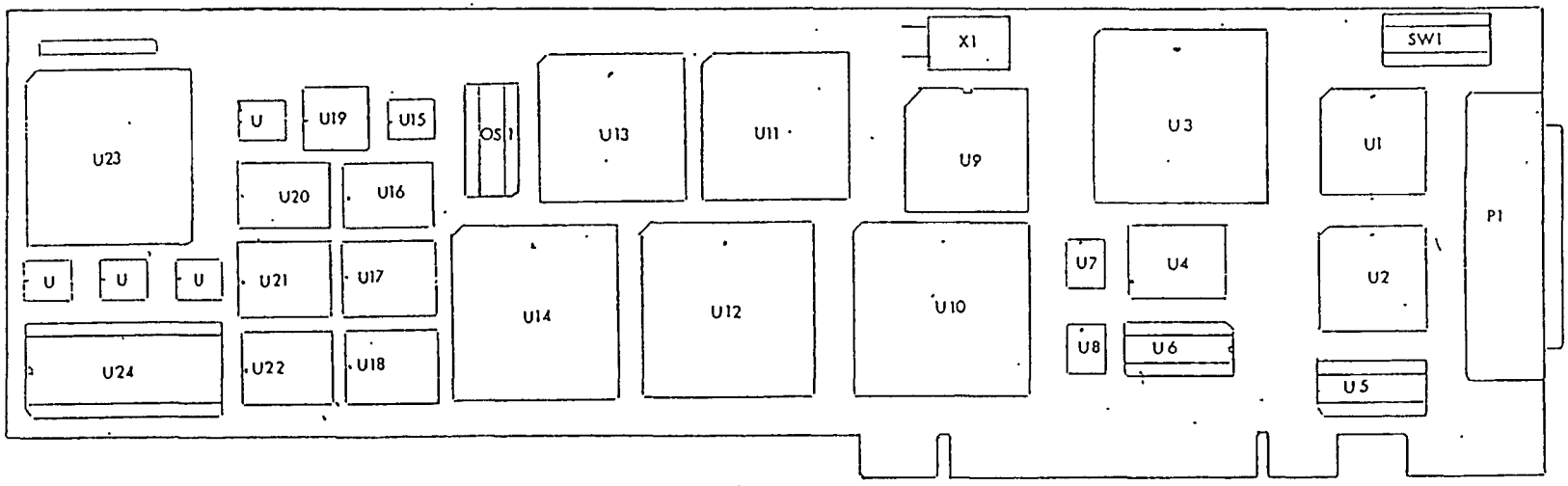


Figure 2-5. Auxiliary Video Connector







XgA MICROCHANNEL

Paper 6-1

SOFTWARE TESTING AND ANALYSIS AS PROGRAMMABLE PROCESSES

Dr. Leon Osterweil
Professor of Computer Science
UC/ Irvine

Mr. Leon Osterweil is a Professor in, and Chair of, the Department of Information and Computer Science at the University of California, at Irvine. Previously he was a Professor in the Department of Computer Science at the University of Colorado at Boulder, where he served as Chairman for four years. His research interests center on tools and environments for supporting software development and maintenance, with a special interest in testing and analysis tools. Osterweil has been the Program Chair of the 2nd Symposium on Practical Software Development Environments and 2nd Workshop on Software Testing, Analysis, and Verification, and the General Chair of the 4th International Software Process Workshop. He received his Ph.D. in Mathematics from the University of Maryland.

Software Testing as a Programmable Process

Leon Osterweil

UNIVERSITY OF CALIFORNIA
IRVINE, CALIFORNIA
USA

SOFTWARE TOOLS

Items of software that help software practitioners to do their jobs better

OUR TOOLS

COMPILERS

EDITORS

LOADERS

STATIC ANALYZERS

DYNAMIC TESTERS

DESIGN AIDS

REPORT GENERATORS

An enviroment is a collection of tools EFFECTIVELY integrated so as to provide software practitioners with effective support for their activities.

WHAT IS EFFECTIVE INTEGRATION?

Enviroment Integration Rationales

"The purpose of an enviroment is to support users in the activities they perform...."

-Tool Centered
Lowest Level Activities

- Object/Product Centered
Tools as Vehicles for Building Products

- Process Centered
Focus on Activities Aimed at Producing Product thru Coordination of
People, Tools, Objects, etc.

The quest for effective
INTEGRATION
of Tools

SOFTWARE ENVIRONMENTS

TOOLPACK/ODIN PROJECT

- Done by university/government/industry consortium
- Research: Build and evaluate innovative tool integration mechanism ODIN
- Development: Use it to integrate large diverse collection of Math. software tools--Toolpack
- Technology Transfer: Transition it to general use--->600 sites now have toolpack/1

user group meetings book ...

The ODIN Project

Philosophy: The goal of software development is the creation of a software product that is a structure of software objects of various types

An environment should foster and support this view

ODIN PHILOSOPHY

- Software is a product composed of smaller software objects
- User specifies how the product is structured from objects
- Odin activates tools automatically in an optimized order to create the final product

ODIN ARCHITECTURE

- Maintain a store of software objects created from each other by action of tools
- Objects are organized by:
  - Hierarchy Relations
  - Derivation Relations
- Record history of object derivation by tools
- Encourage synthesis of larger tools from smaller fragments through effective reuse of objects and fragments

Significant Efficiency Advantages

OBJECTS ARE TYPED

ODIN COMMAND LANGUAGE ENFORCES

STRONG TYPING

FOCUS ON DATA REPOSITORY

- What goes into it?
- How do users view it?
- How do tools create/update it?

:

Some Object Types

Source

Parse Tree

Flowgraph

Formatting

Parse of formatting

Instrumentation

Test data

Test results

Test report

Edit script

Edit update

Parse of Formatting
of Edited Version

ODIN Object Names

JOE

JOE: PRS

JOE: FGR

JOE: POL

JOE: POL: PRS

JOE: INS

LUNCH

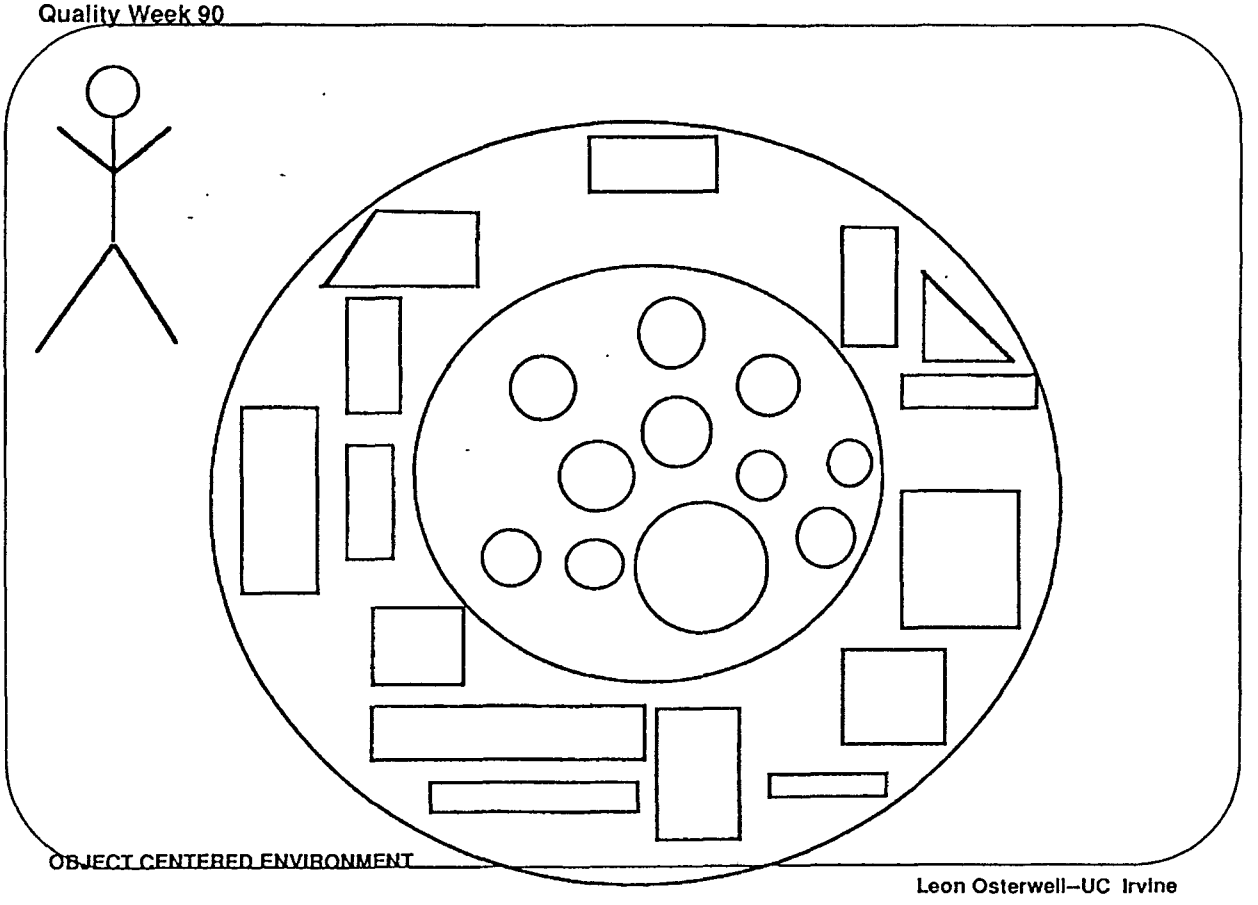
JOE + IN = LUNCH: RUN

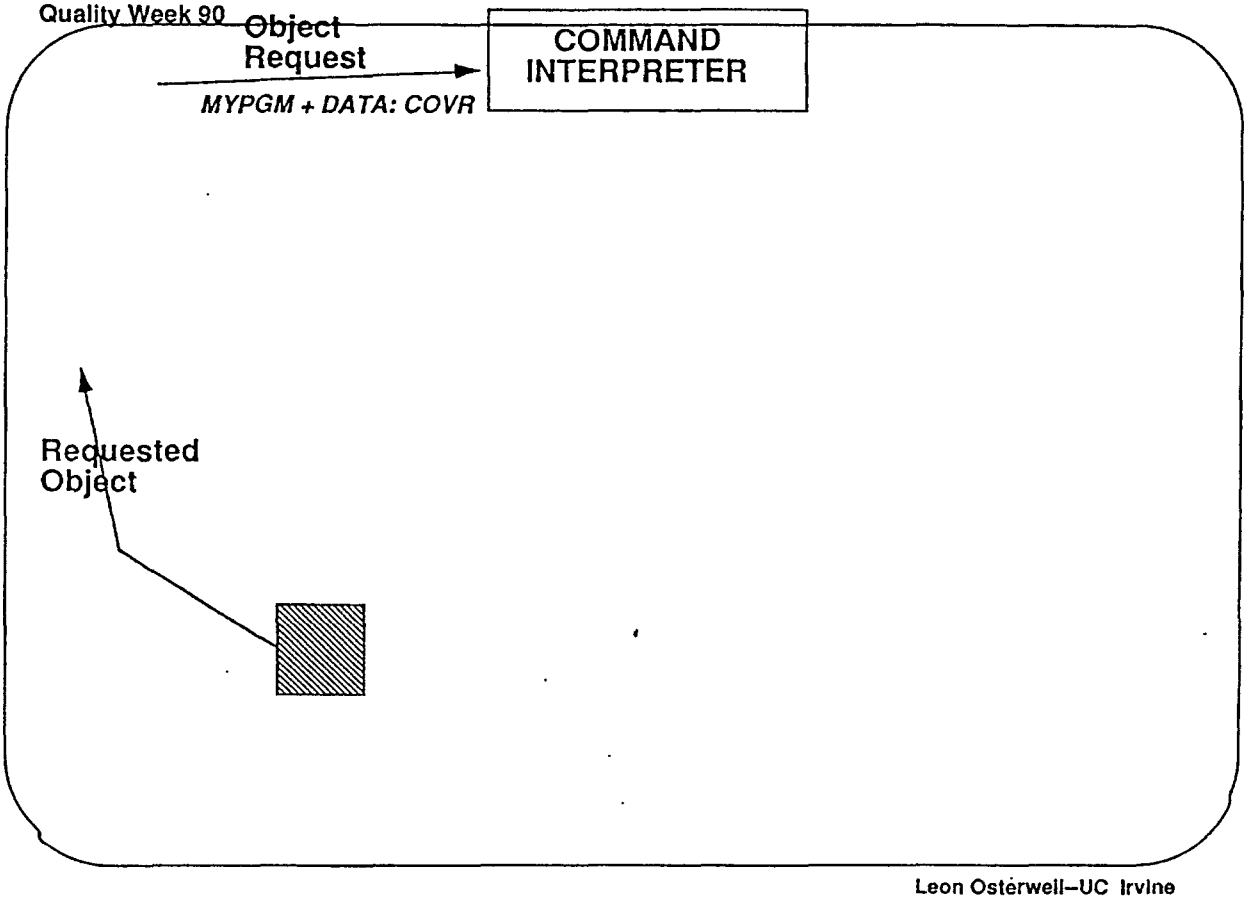
JOE + IN = LUNCH: RUN: SUM

EDIT5

JOE + EDIT5< : ED

JOE + EDIT5< : ED :POL : PRS


OBJECT CENTERED ENVIRONMENT


Object
Request
MYPGM + DATA: COVR
COMMAND
INTERPRETER
Requested
Object

WORK LIST:
Instrument
Compile
Load
Execute
Measure
Coverage

C

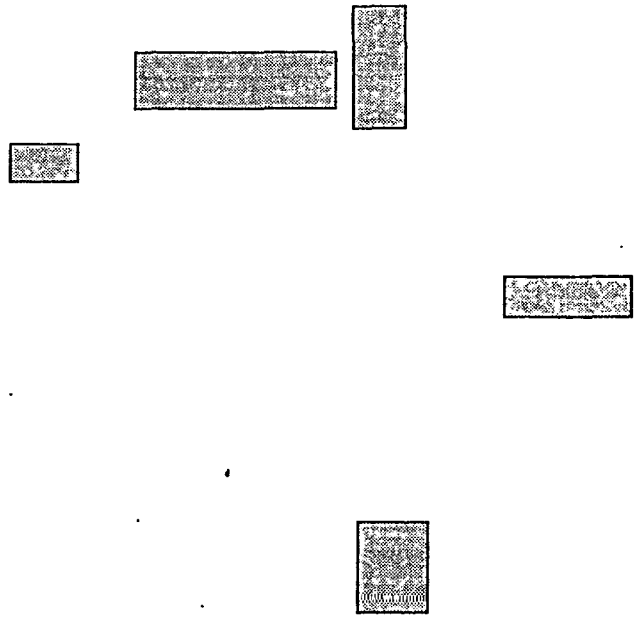
I

L

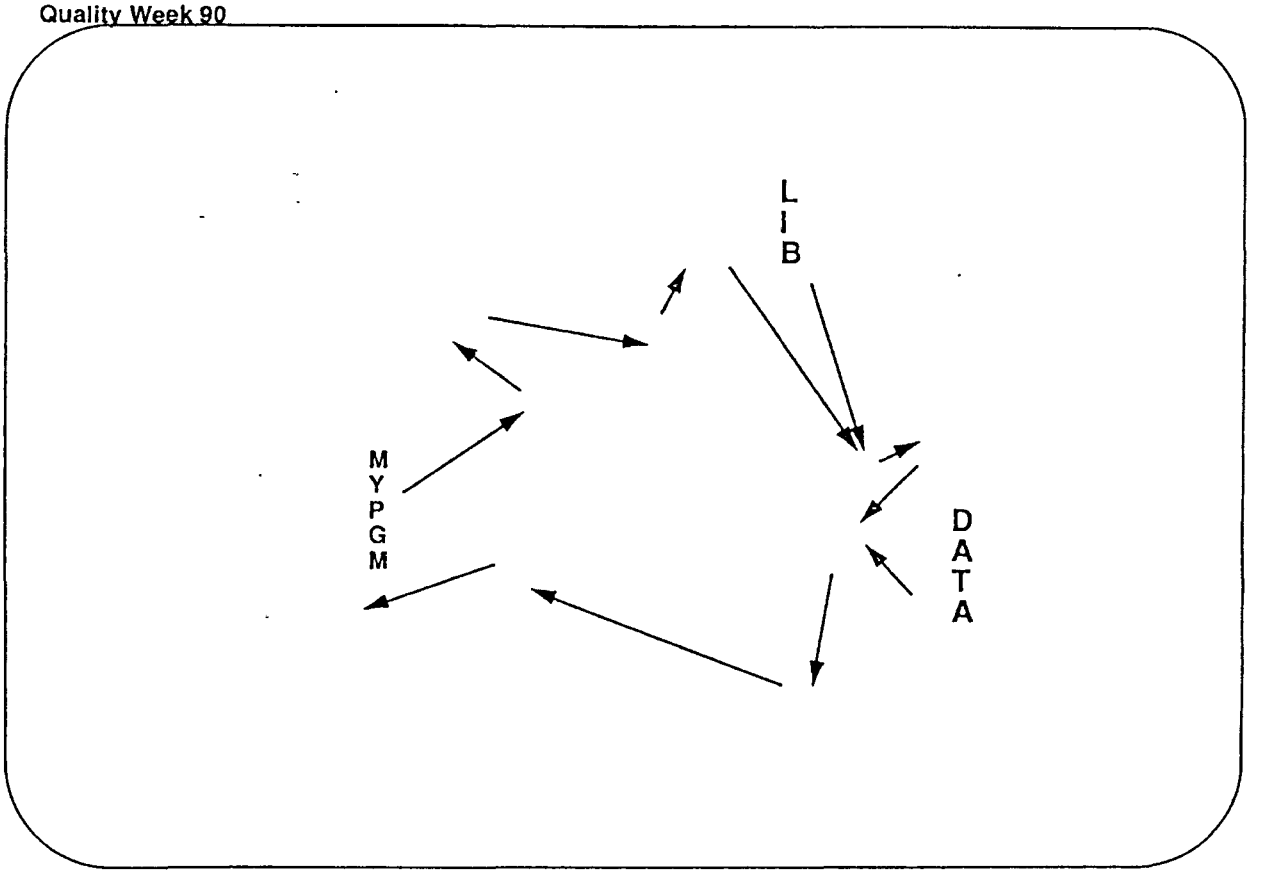
E

M

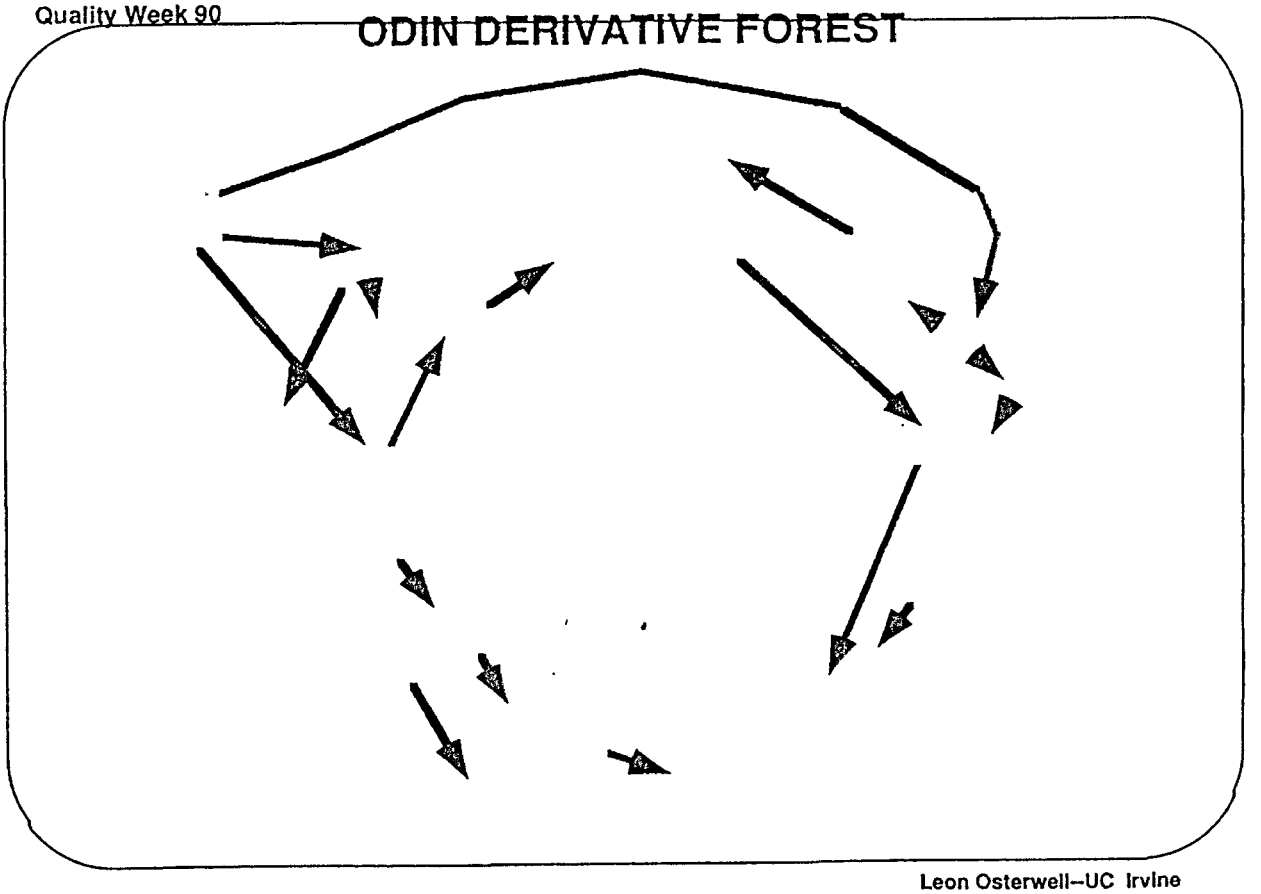
Quality Week 90



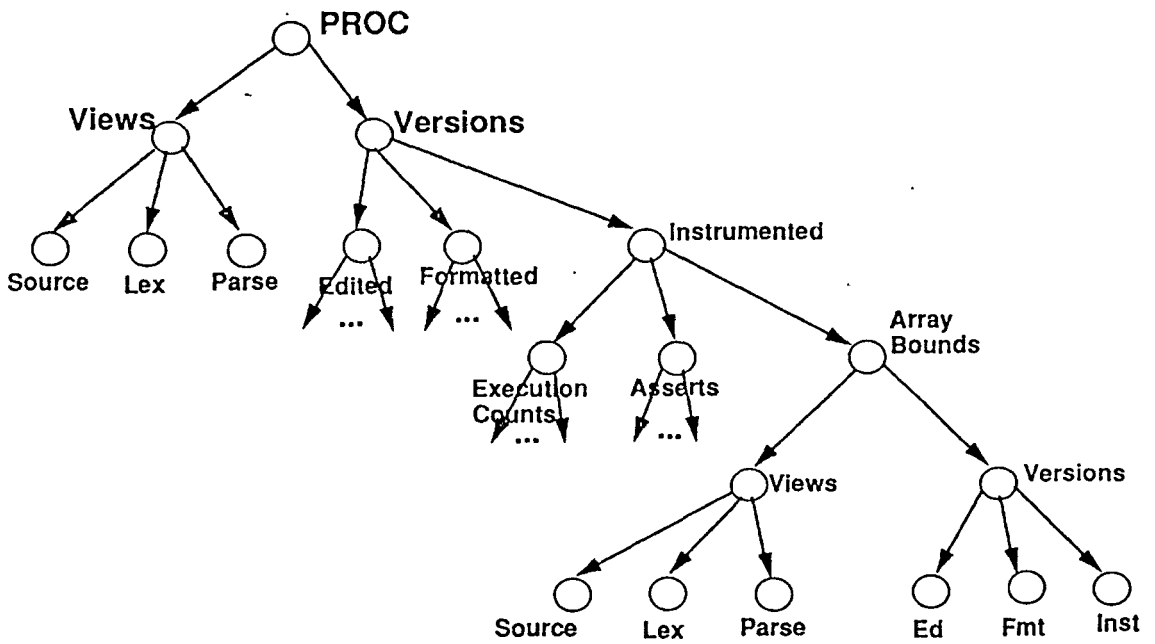
Leon Osterweil--UC Irvine



ODIN DERIVATIVE FOREST



ODIN DERIVATIVE TREE FOR PROCEDURE PROC



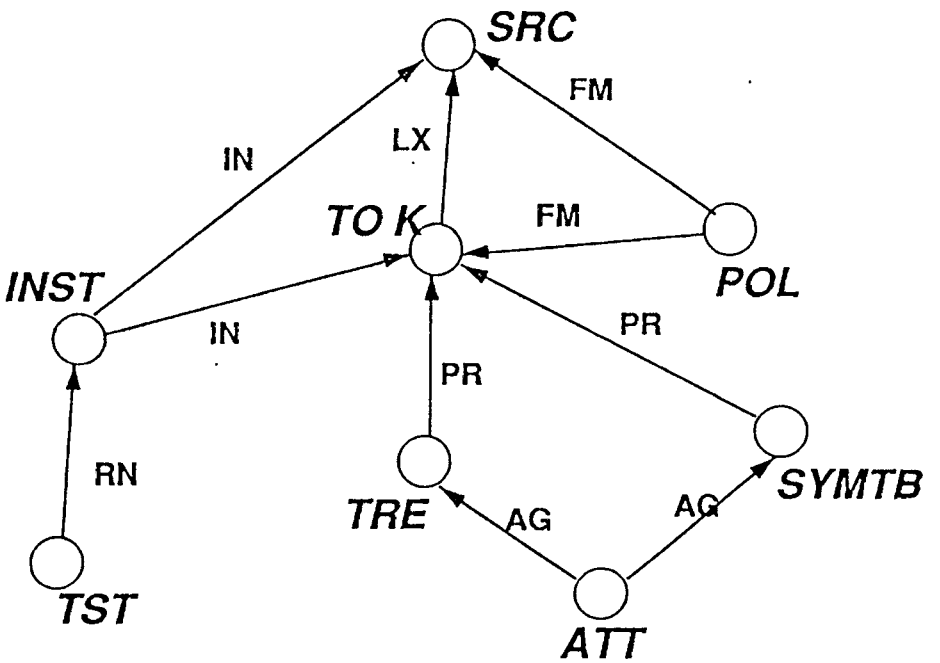
DERIVATION TREE IS BASIS FOR

-- Consistency Control

--Updating Mechanism

- All Objects Have Time Stamps
- If Object σ Has a Time Stamp LESS Recent Than SOME Ancestor
  => σ May Be Out of Date
- Lazy (Demand) Evaluation Strategy: Don't Check Until σ Requested
- Rederive From Highest Newer Ancestor
- Check Rederived Objects
- Stop When They Are No Longer Different

DEPENDENCY DAG

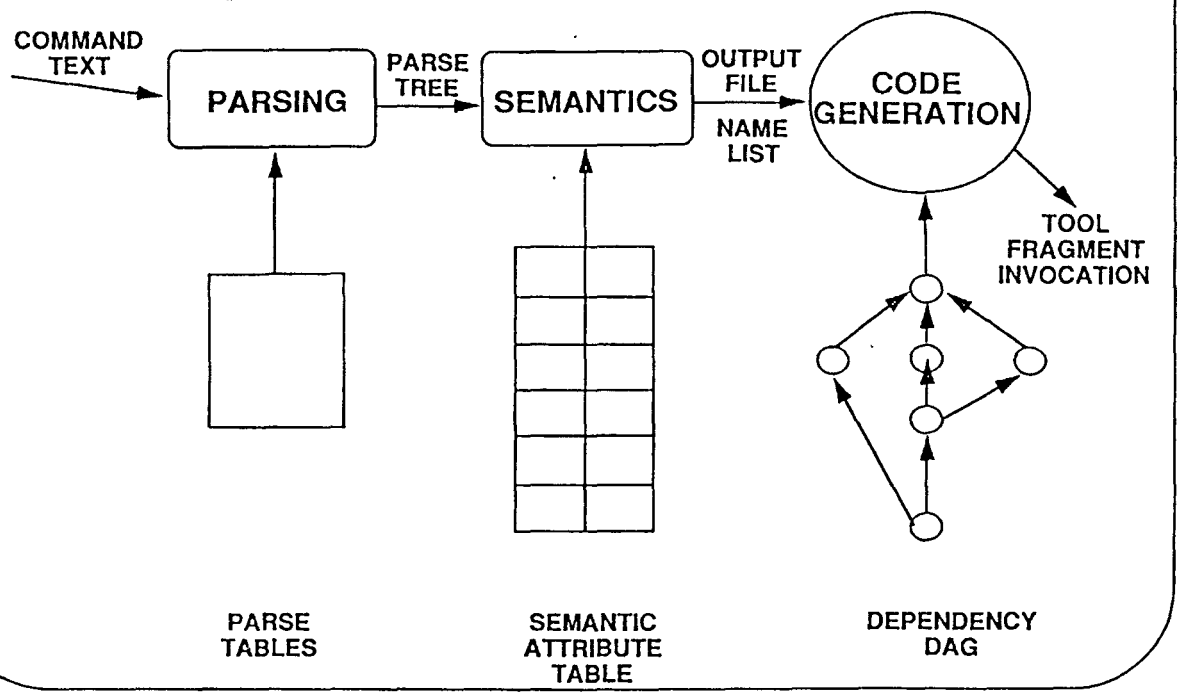


CAPABILITIES OF SYSTEM CAN BE EXTENDED/ALTERED

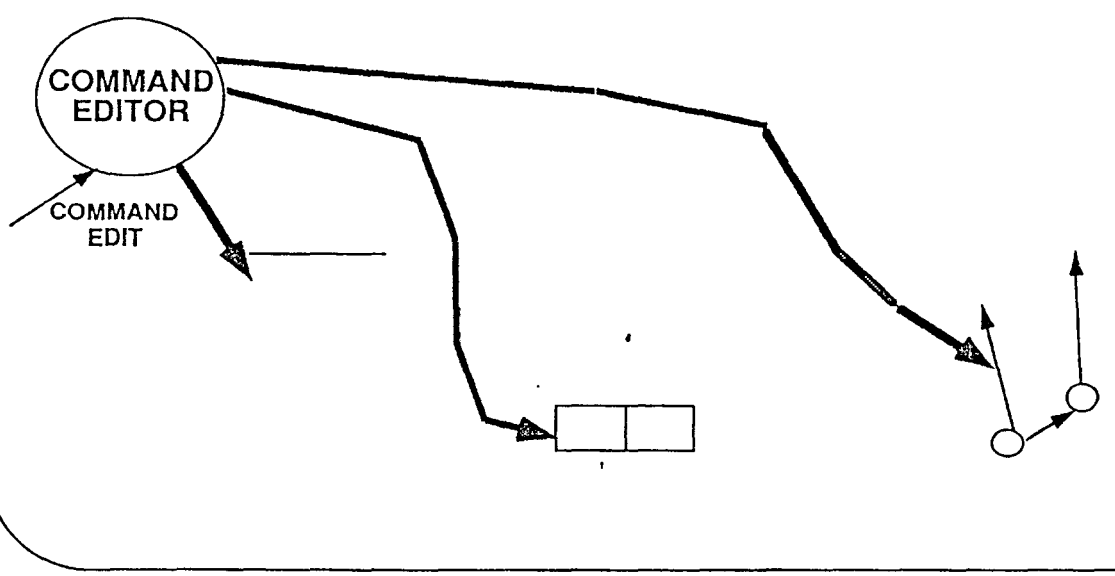
- By addition of new tool fragments or alteration of command semantics
- Requires consistent alterations to all three phases of command compiler

  ---Parsing
  ---Semantics
  ---Code Generation

ODIN COMMAND COMPILATION



Leon Osterweil–UC Irvine



Leon Osterweil–UC Irvine

TOOLS AS COLLECTION OF "TOOL FRAGMENTS"

- Modular view of tools
- Advantages of viewing software as built from modules
  - Reusability
  - Orthogonality
  - Flexibility
  - Extensibility
- Module outputs are the data repository objects

EFFICIENCIES FROM REUSE OF OBJECTS GENERATED BY MODULAR TOOL FRAGMENTS

- Lexical string, parse tree reused by various higher level tools (eg. formatter, instrumentor)
- Entire subprocedure derivation structures reusable through overlapping groups
- Automatic updating from changed "includes"
- Extensibility facilitated as well

GROUPS OF PROGRAM UNITS

Are used to represent
---programs
---libraries
---hierarchical structure of a body of code

A group is a set of program units and/or other groups

---must be non-circular

But.....

Groups may overlap, share lower level groups

CAN SPECIFY DERIVATION OF A GROUP

IF
GROUP.REF

is a group consisting of

A.F
B.F
C.F

THEN

GROUP.REF: INS

specifies the derived group consisting of

A.F : INS
B.F : INS
C.F : INS

REPORT GENERATION AID

- Reports viewed as compositions of objects
  - ---Source text
  - ---Formattings
  - ---Diagnostic output files
  - ---Analyses of dianostic outputs

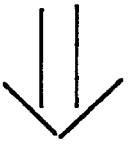
MANAGEMENT AID

- Through assured up-to-date reports on testing, analysis, etc.

THINGS WE DID RIGHT

Data center

Modular tools



Extensibility
Efficiency
Flexibility

SOME POSITIVE LESSIONS LEARNED

Object Orientation Is Useful

Tool Fragment Architecture

- Encourages Object Reuse
- Encourages Fragment Reuse
- Yields Better Tools More Readily
- Enables Optimized Object Creation

Object Typing Facilitates All This

Dependency Graph Structure

- Yields Extensibility
- Enables Inferencing
  (terse commands)

Linguistic View

The user of an Odin-integrated-environment as a
PROGRAMMER

Language has---
Types
Aggregation
Optimization

but lacks---
Formal typing mechanism
Flow of control
Scoping rules
Relations

Odin as a Language Interpreter

Objects are typed

Objects can be aggregated

Tools are operators

Strongly typed

Procedures enable macros

Extensible

Optimization thru object reuse

WHAT IS A PROCESS?

Device for:
- Producing a product
- Getting jobs done

Indirect nature--
- A process is an instance of a process description (program)
- Instance creates product/solves problem
- Process description (program) created to describe wide class of instances

Humans create process descriptions (programs) to solve classes of problems

COMPUTER SOFTWARE PROCESSES

Devices for creating, manipulating computer software products

Devices for evolving software products

CLASSICAL APPLICATION "PROGRAM" AS A PROCESS DESCRIPTION

Description of device for creating an INFORMATION product

Each execution (binding to input data) is an instance

Example of indirect problem solving

Product often complex, interconnected aggregate

Description is always formal, rigorous

-----What is the difference between this and other processes (e.g., software development) ?

Complex process to

Develop it

Evolve it

This process VERY informally specified

---- What is the difference between

this and other processes

(e.g., software development) ?

Software Development as a Programmable Process

Software Development process as Software

Why not program software development?

--- with a programming language?

Higher level of discourse, abstraction

Many "operations" interpreted by humans

Manager chooses "interpreter" by binding execution of process to human/machine

Such an enhanced language as a vehicle for

PROCESS PROGRAMMING

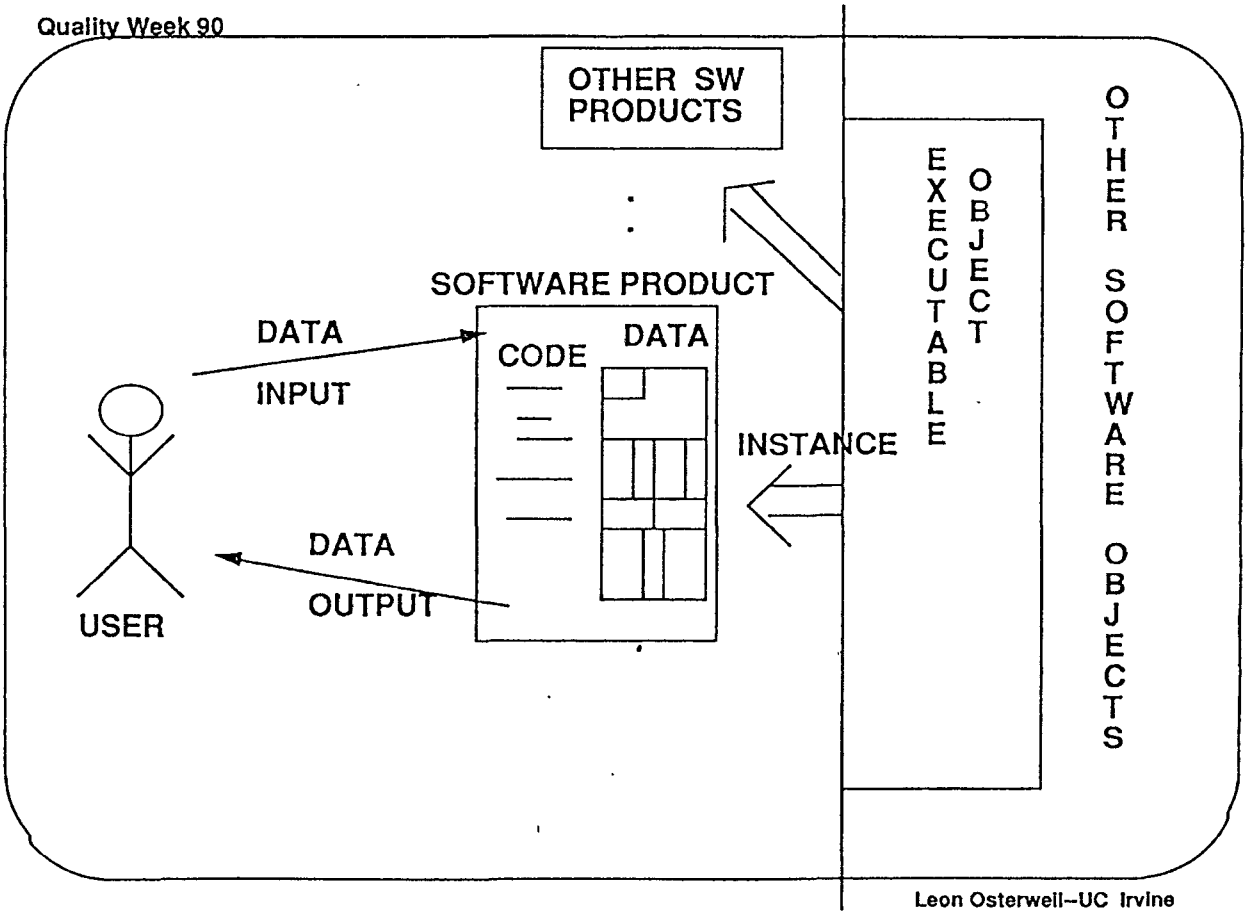
PERVASIVENESS OF PROCESS DESCRIPTIONS

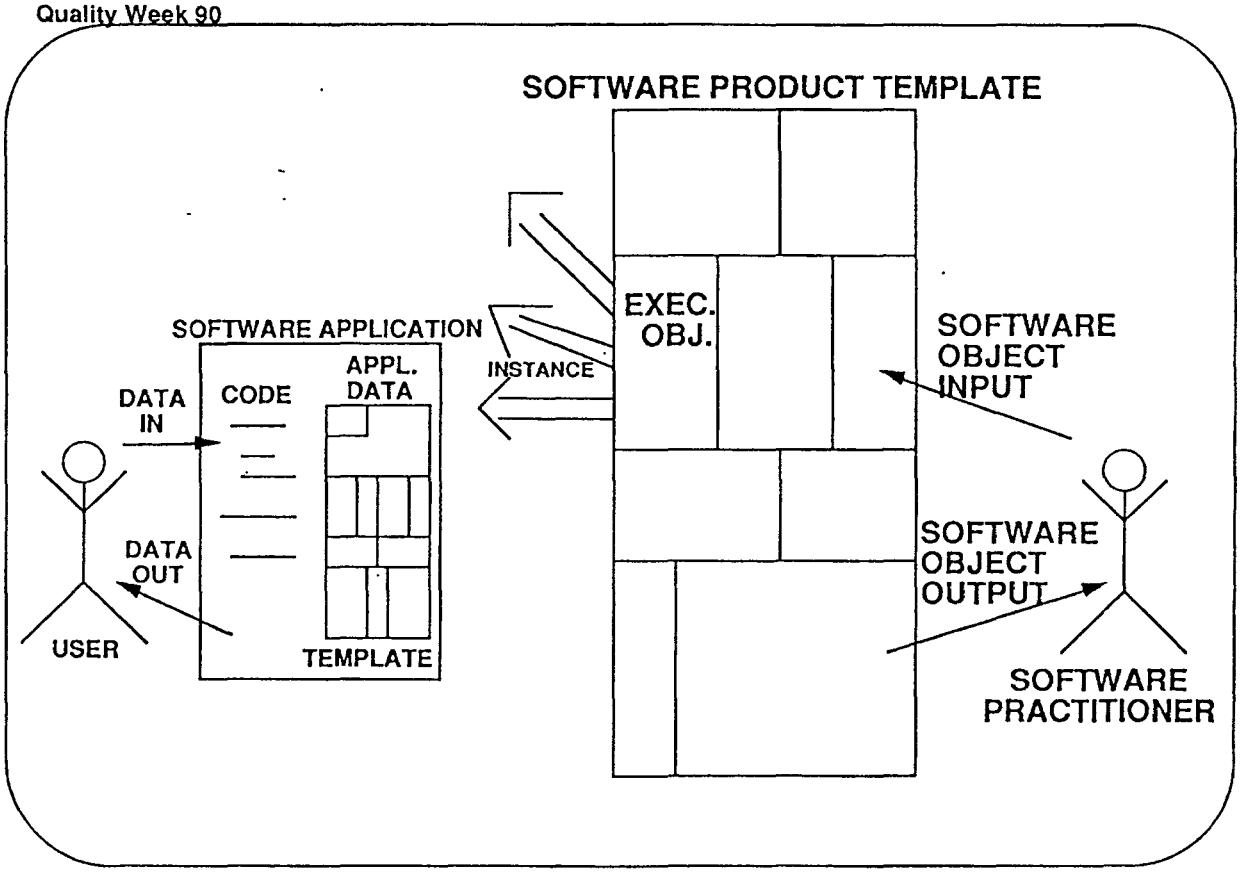
SOME EXAMPLES -----

- Cooking receipes (Knuth)
- Kit Assembly
- Game Playing Instructions
- Driving Directions
- Academic Programs
- Office Procedures
- Manufacturing Processes
- Laws

Humans are adept at process description/execution

Descriptions are too informal





WHERE DOES THE PROCESS COME FROM?

PROCESS is Software Too
It is to be Developed
BY
Process Development
Process

There is NO Single,
Fixed "Ideal" Software Development Process

SOFTWARE DEVELOPMENT PROCESS
Is Developed By A
(Software Development Process) Development Process

SOFTWARE DEVELOPMENT PROCESS

- Must Satisfy SDP Requirements

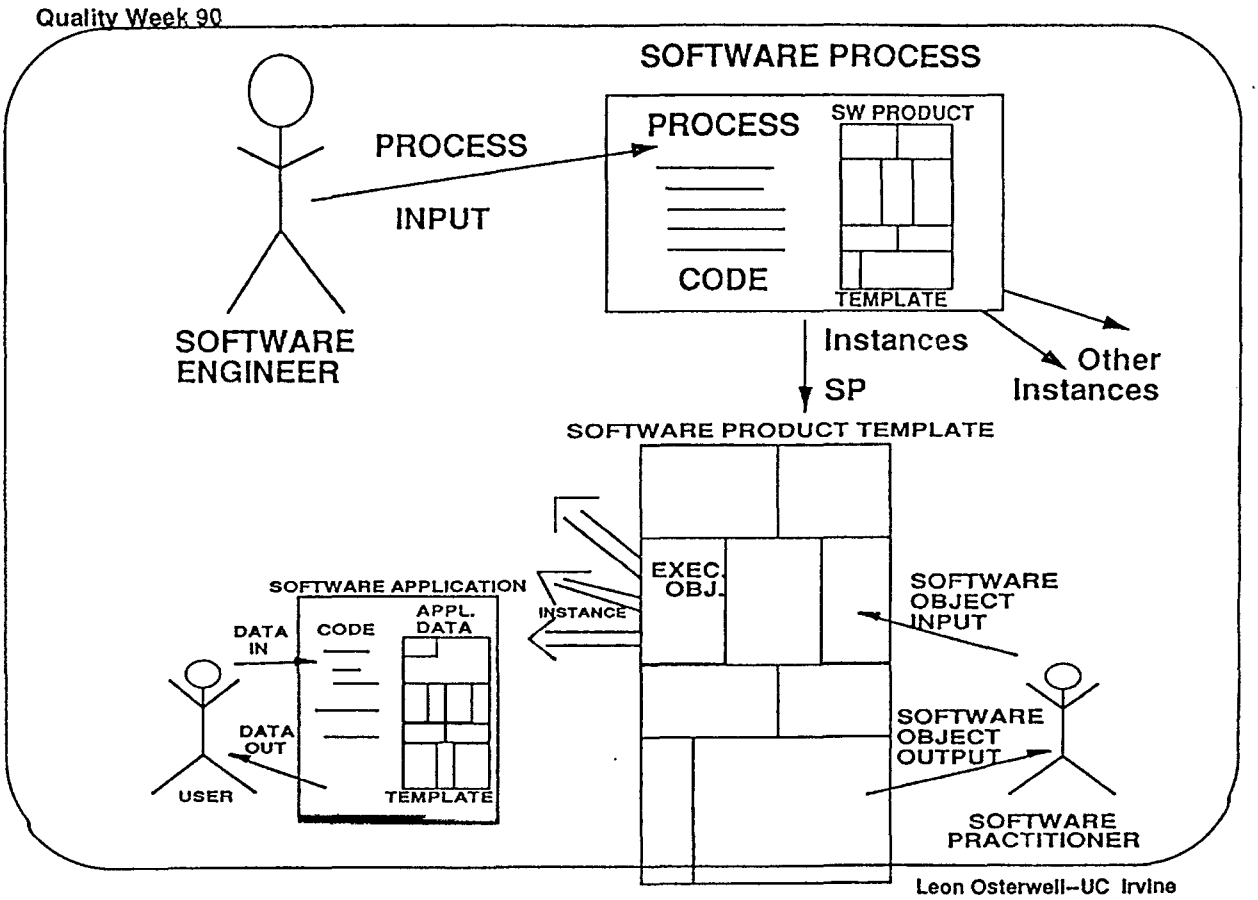
  Such as:
  Visibility of Progress
  Execution Speed
  Quality of Product Built

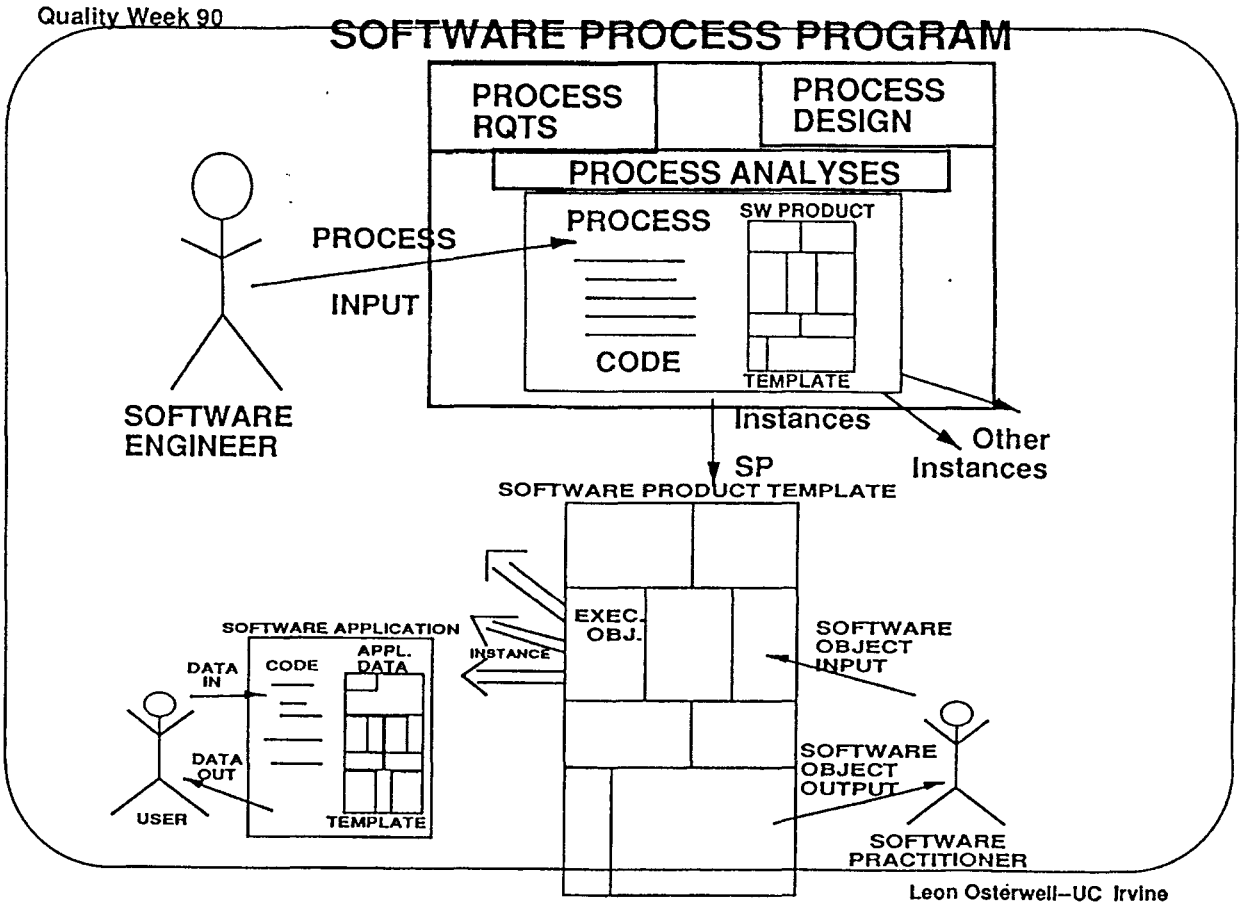
- Should Use Reusable Design Modules

- Process "Code" Must Be Analyzed, Measured, Tested, and Altered

  Strong Experimental Flavor

Diverse Software Development Processes Not One Ideal





Software Process Programming Is

OLD Discipline Of Programming Applied To The

NEW Application of SW Process

What's NEW Here

Process Tasks Can Be Bound To Humans or Computing Devices

Processes Can Produce New Process Tasks Dynamically

Problems Caused By Binding Tasks (e.g., Design, Debugging, Documentation) To Humans

- Results Are Indeterminate
- Process Semantics Not Rooted in Mathematical Semantics
  - -- Based on Rigorous Languages or Hardware

INSTEAD

- Semantics Can Be Defined In Terms of Pre/Post Conditions
  - -- Enables Reasoning About Higher Level Processes
  - -- Possible Elaboration of "Human" Processes Iteratively

Software Engineering Must Show How To Temper

Human Intuition, Insight Informality

WITH

Mathematical Rigor, Precision

Process Programming

Suggests A

Way

SOFTWARE PROCESS RESEARCH IMPACTS

Process programming is .....

- A tool for sharpening discourse about process
- A tool for circumscribing and clarifying "hard" process issues
- A STEP towards goal of adequate process description formalism

CAUTION: Not the ultimate formalism
Vehicle for precipitating
-- understanding
-- progress

ANALOGY:

REAL-TIME INDUSTRIAL PROCESS CONTROL

In A Rigorous Precise Language

Some "Invoked" Processes

Such As
"Smelt"
"Mix"
"Roll" ....

--- Are Untrustworthy, Unclean
--- Have No Rigorous Semantics

And, Yet, Are Effectively

--- Integrated Into Process

PROGRAMS =

SOFTWARE =

PROCESS DESCRIPTIONS

All are passive/active

Uniform character of all

Environment as manager of all

Software engineer as process programmer

RESEARCH DIRECTIONS

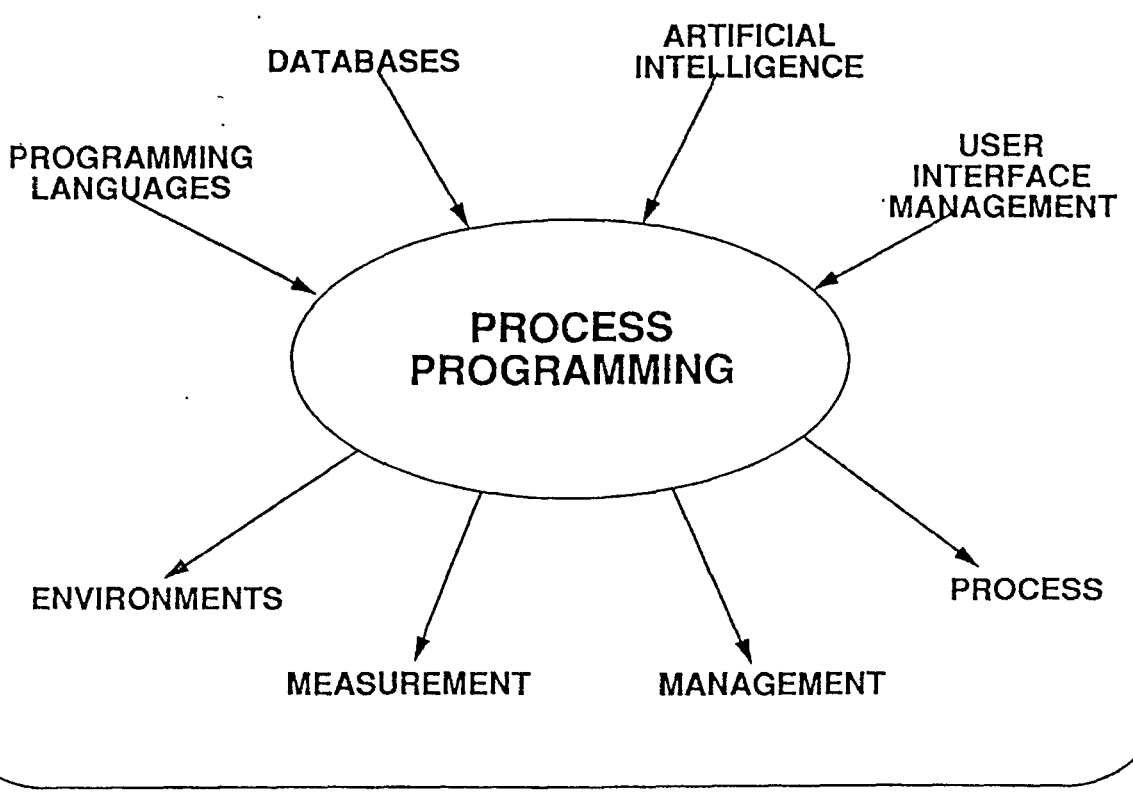
Need to draw upon---

- Programming Languages
- Databases
- Artificial Intelligence
- User Interface Management

Should enrich---

- Software Environments
- Software Measurement
- Software Management
- Software Process

AN IMPORTANT RESEARCH NEXUS



PROGRAMMING LANGUAGE RESEARCH ISSUES

WHAT IS RIGHT LANGUAGE PARADIGM?

- Algorithmic
- Applicative
- Rule Based

WIDE SPECTRUM ??

Need language offering powerful, flexible

- -- protection, visibility,,naming
- -- binding
- -- concurrency
- -- typing system
  with flexible inheritence scheme

DATABASE RESEARCH ISSUES

Effective support for this requires database with .....

- Persistent objects
- Processes as objects
- Types and type structures as objects
- Management of large (megabyte) objects
- Long (weeks or months) transactions
- Automatic inferencing
- Dynamic type creation, alteration

CHALLENGES FOR AI

Elucidation of "hard" processes

-- e.g., design

-- e.g., debugging

Artful, incremental integration of knowledge into processes

-- e.g., through incremental addition of rules

USER INTERFACE MANAGEMENT SYSTEMS ISSUES

How to view all of this interactively?

-- Multiwindow communication with running processes

-- Process stop/start/alteration

-- Superior pictures

ENVIRONMENT ARCHITECTURE RESEARCH

Environment is a device for

Specification

Analysis

Compilation

Interpretation

--- of software process programs

ENVIRONMENT ARCHITECTURE RESEARCH

- What are key operators?
  - Canonical tool fragment decomposition
- How to store/access software objects
  - Database research applied to software engineering
- How to define type system
  - Language design applied to software engineering
- Scope/visibility rules applied to persistence management
  - Exploit language design experience
- How to make humans effective interpreters

SOFTWARE MEASUREMENT RESEARCH

Are software process programs the right objects to measure?

Merging performance measurement with software measurement

Completion assessment based on execution status

Cost estimates based partially on

- software object complexity
- software process complexity

SOFTWARE MANAGEMENT ISSUES

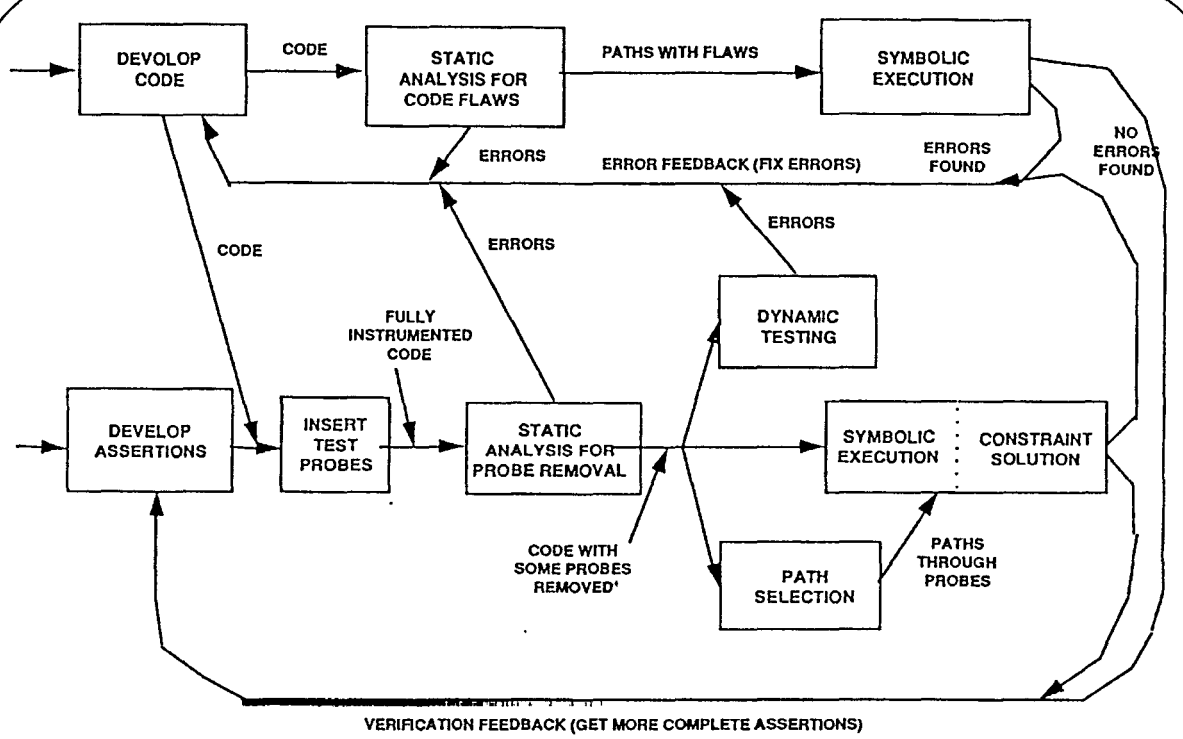
Management seen as software process task binding
-- and execution monitoring

More effective management from better visibility
-- provided by process programs

Qualitative improvement over vague, inconsistent,
undermaintained "project plant"

Managers as process program writers/readers
-- return to "native element"

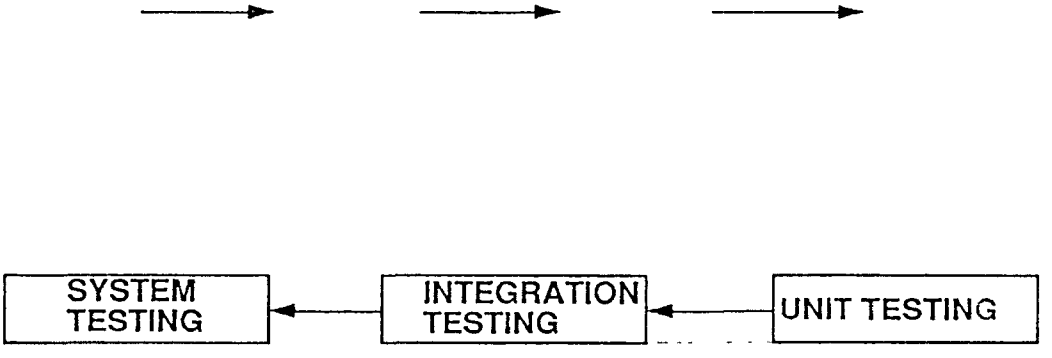
Materializing process enables reuse by others,
survival beyond career of successful manager



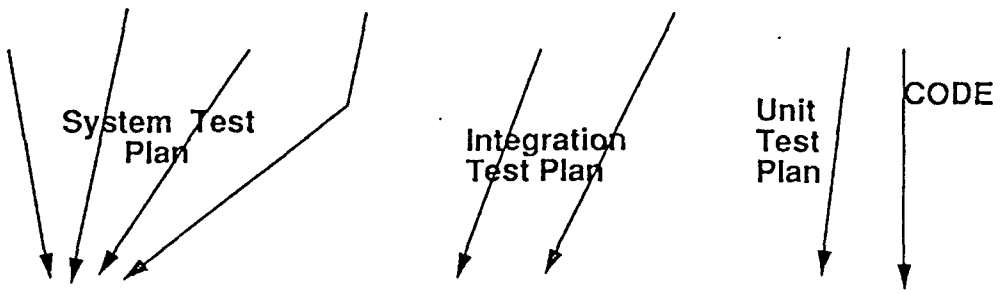
Testing & Analysis Process Architecture

DEVELOPMENT PHASES

TESTING PHASES

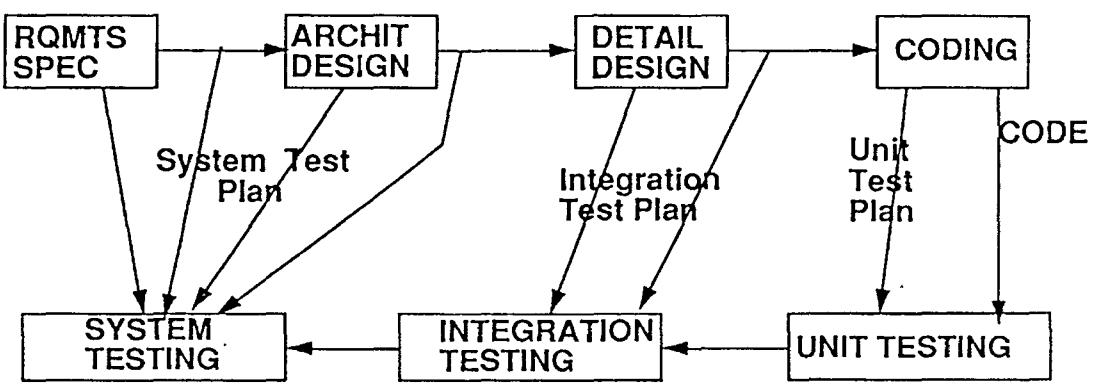


TEST PLANNING



DEVELOPMENT PHASES TESTING PHASES

TEST PLANNING



PROCESS PROGRAMMING PROJECTS

- TESTING / TEST PLANNING
  ---- BOWTIE
- REQUIREMENTS SPECIFICATION
  ---- REBUS
  ---- JSD
  ---- SREM
- DESIGN
  ---- DEBUS
  ---- RDM
  ---- YOURDON
- MAINTENANCE
  ---- METEOR / ORBIT

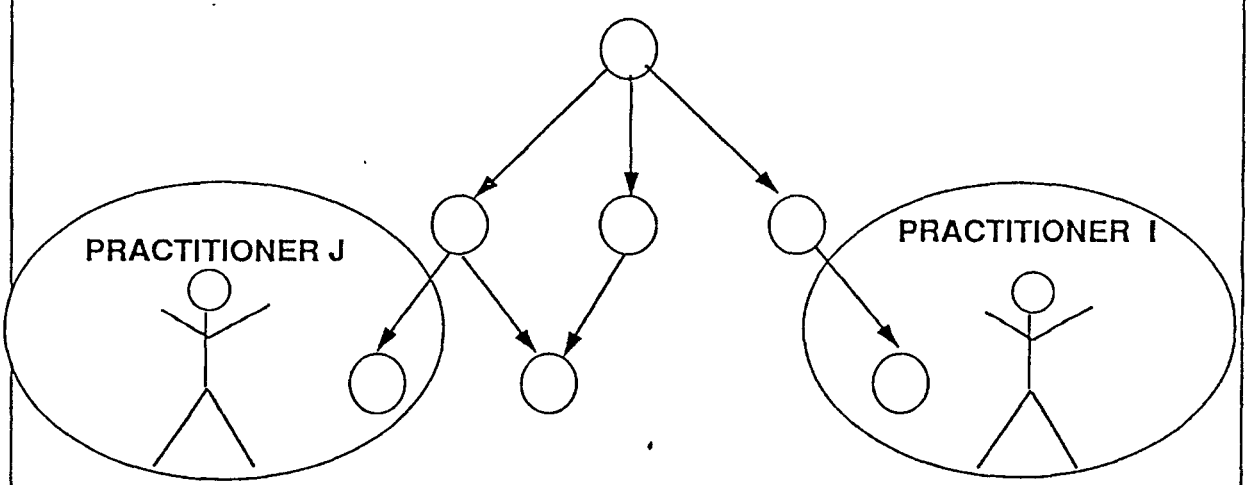
REquirements BUilding System (REBUS) Concepts

- Requirements Specification as hypertext
- Structure of Requirements Elements
- Requirements Element is a record
- Requirements Element fields carry information as:
  - Instances of preset types
  - Instances related to others by Appl/A relations
    -- express consistency rules
    -- define consistency determination
    -- define inconsistency remediation
  - Relations among
    -- requirements elements
    -- requirements elements and other SW objects (e.g., testplan elements)

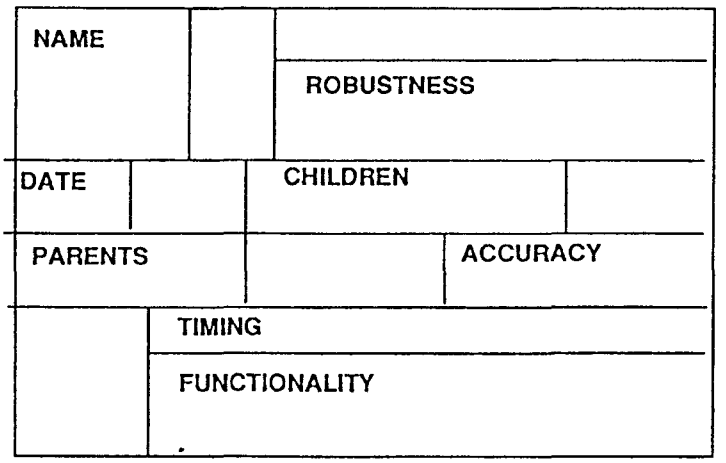
REBUS USER VIEWS

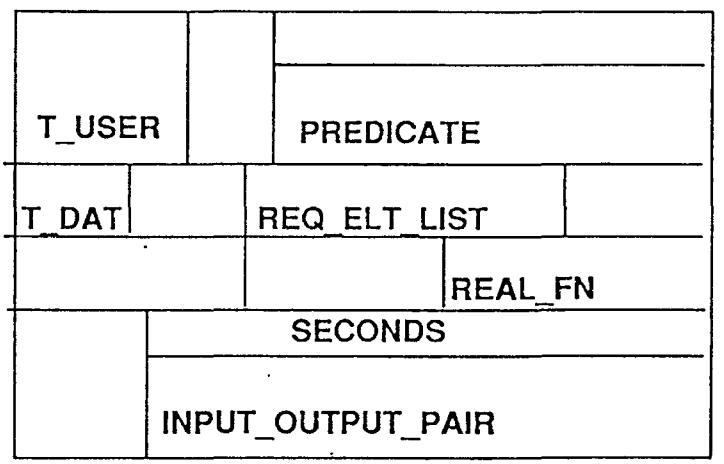
- Practitioner View
  - View structure
  - Select node to elaborate
  - Insert values in fields
  - Get guidance/diagnostics from Rebus
  - Indicate descendant nodes
- Process Programmer View
  - Design node format
  - Specify constraints
  - Design views
  - Control interactions
- Researcher View
  - Rebus as object to measure
  - Vehicle to study requirements specification
  - Consolidate past requirements work

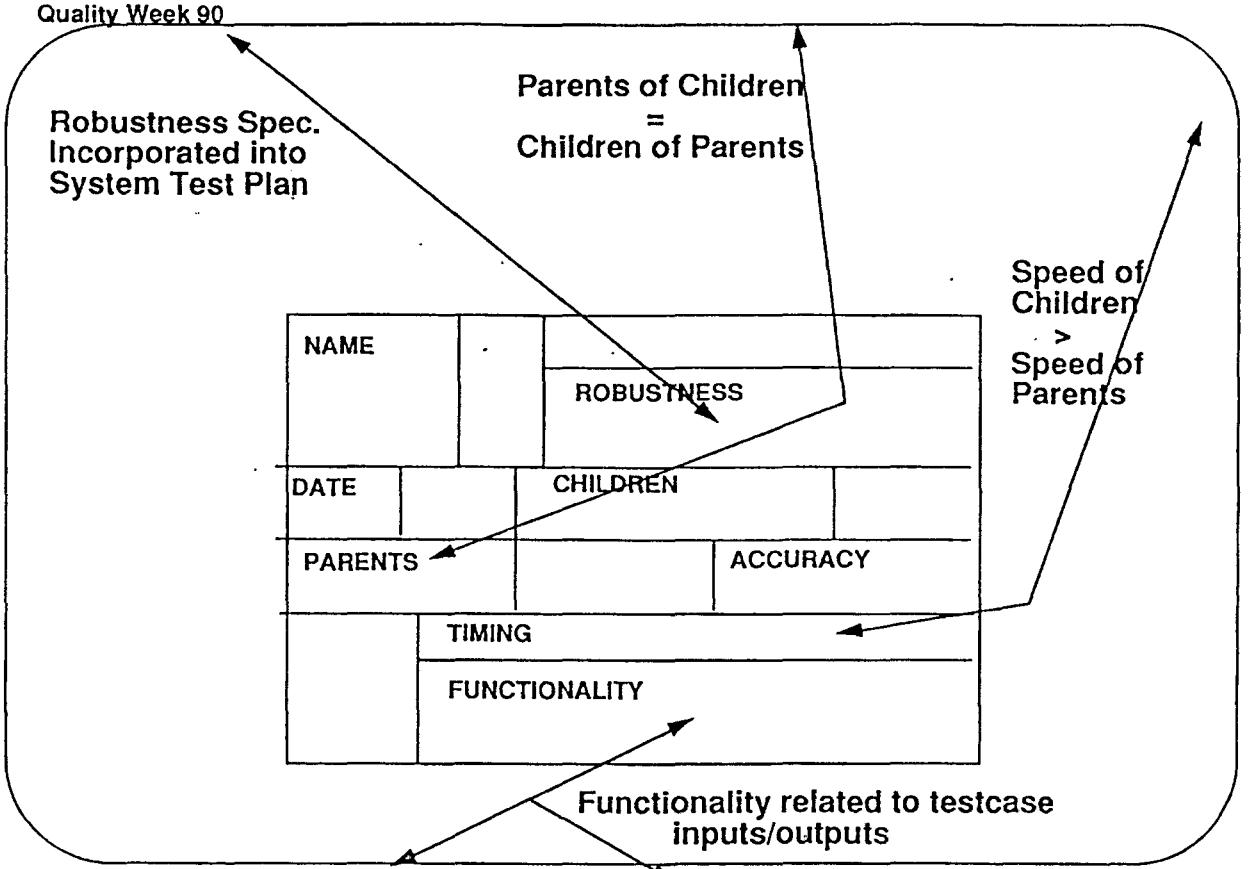
REBUS WITH MULTIPLE USERS



REBUS NODE


NAME
ROBUSTNESS
DATE
CHILDREN
PARENTS
ACCURACY
TIMING
FUNCTIONALITY


T_USER
PREDICATE
T_DAT
REQ_ELT_LIST
REAL_FN
SECONDS
INPUT_OUTPUT_PAIR



Introduction to APPL/A

APPL/A: Ada Process Programming Language based on ASPEN

ASPEN is a data model for software engineering

- Recognizes the importance of relationships among software object
- Provides an Entity-Relationship data model
  ----Relations over objects
- Emphasizes *programmable* semantics and implementations for relations

APPL/A is an extension to Ada that includes relations and other constructs based on ASPEN

APPL/A is a *prototype PPL* for experimentation with software object management

Relation of APPL/A to Ada

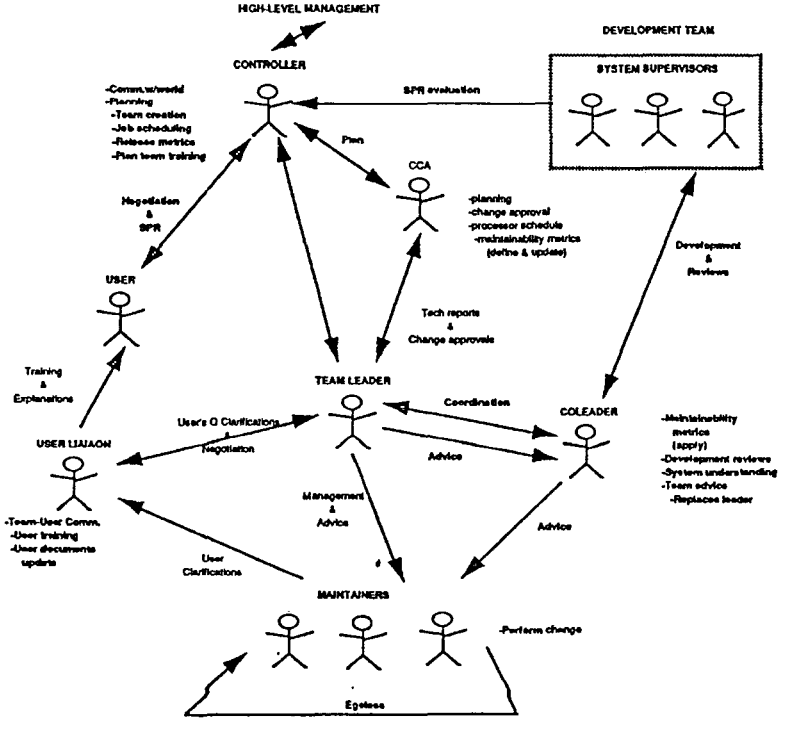
- APPL/A is a superset of Ada
- APPL/A extends Ada with
  - Relation Units
    --- persistent storage
    --- programmable implementations
    --- active and reactive
  - Tuple type constructors
  - Loop constructs for interative, selective retrieval of tuples from relations
  - "Upon" statement for "reaction" to operations on relations
- APPL/A otherwise relies on Ada type definition and control mechanisms

DEBUS (DEsign BUilder System)

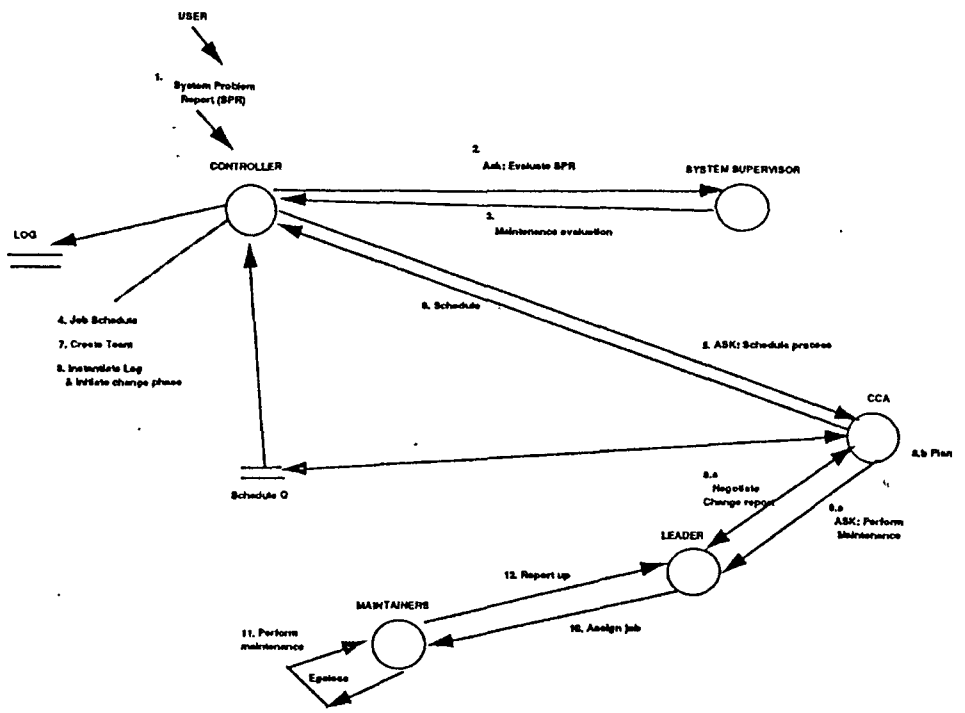
- Design as a hypertext-like object
- Support for RDM and Yourdon
- Process for configuring the design process
- Coordination of multiple designers
- Relations among design elements
- Relations to requirements objects

TSURU

- Management Process Program
- Assigns Workers to Tasks
- Monitors/Tracks progress
- Specification of skills required by tasks
- Driven by process graph



Maintenance Process Architecture



Maintenance Process Dataflow

Leon Osterweil--UC Irvine

Paper 6-2

THE "SOFTWARE FORTRESS EUROPE" OF 1992: A POSSIBILITY EXPLORED

Dr. James Hemsley
President
Brameur, Ltd.

Dr. James Hemsley is Managing Director of Brameur, Ltd., a European Research Consultancy with a special focus on Software management, operating out of Hampshire, England. He is the Project Manager of two ESPRIT Projects on Software Quality Metrics: MUSE and METKIT, and is a frequent speaker at high-level technical conferences throughout Europe. He is also a member of ISO/IEC, European and British Standards working groups in the Software Quality Assurance field.

SOFTWARE FORTRESS EUROPE - A POSSIBILITY EXPLORED

James Hemsley

I INTRODUCTION

This paper addresses the danger perceived by some U.S. and Japanese observers that West European countries are moving towards protectionism in the software field and that '1992' will signify a form of 'Software Fortress Europe'. The position taken in this paper is the contrary one in general and that the 'New Europe' will offer increasing opportunities for U.S. trade and investment in the software field as well as in others. However, it is also argued in the paper that unethical or poor quality suppliers may well perceive increased defences against them.

Beginning with a brief overview of the meaning, nature and importance of '1992' for Western Europe, the paper then presents thumbnail sketches of West European software markets and industries. This provides the background to the centre piece of the paper: the issues and developments in Software Quality with particular emphasis on the Standards and Certification activities aimed at reducing internal barriers to trade inside West Europe. Three case studies are then considered:

(a) General Software Quality

(b) Safety-Critical Software Quality

(c) Healthcare Software Quality

The momentous events in Eastern Europe are briefly considered before the development of the paper's main conclusions.

II WESTERN EUROPE : 1992 & BEYOND

'1992' has now achieved a remarkably high symbolic importance in West Europe. Five hundred years after Christopher Columbus 'discovered' the New World of the Americas, Western Europe is creating a 'Single Market'. This means one in which West European goods, services and people can travel, trade and work freely - as in the U.S.A. The enormous effort, time and cost involved in national border checks will have vanished totally inside the Single Market - this is the aim.

It would be wrong however to present an oversimplified picture - the 'Single Market' will not happen overnight so that the European Community will wake up on January 1, 1993 to find everything magically different. The transition to the 'Single Market' is being made by some 300 different legal measures each of which cuts at least one chord of bureaucracy. For example one set of measures is to remove the need for 'border control'. This measure is first passed by the European Parliament after preparatory staff work in consultation with the individual nations' governments. This measure has then to be ratified by each nation's parliament before it has validity for that country. This overall process takes time; it was initiated several years ago and to date progress is well on track to achieving all the 300 measures in the target time frame.

'1992' thus represents a dynamic programme of activities each contributing towards the overall goal. The 300 measures include a wide range of issues from passport control abolishment inside the European Community to abolishing technical barriers to trade as discussed later with specific reference to the IT industry.

The European Community currently consists of 12 nations with over 300 million people. Already the largest economic bloc, it should become the largest internal market, but there will still be substantial differences between the U.S. and Japan which will still put Europe at a disadvantage.

However, '1992' should not be regarded as a static target which once achieved will represent a status quo. Already there is planning going on to

(a) make the European Community a political and monetary one as well. This would make the European Community closer to a 'United States of Europe'. However, this is not an aim which pleases all of the Community.

(b) extend the geographical scope by including at least some EFTA countries (European Free Trade Association).

Furthermore other countries have already expressed a desire to join - including most significantly Eastern European countries as discussed further below with specific reference to the software situation. In particular East Germany may already be regarded as a 'fait accompli' member. Michael Gorbachov's vision of a Europe stretching to the Urals however, still looks unlikely. The 1990s thus appear to be a challenging decade for Europe.

III THE WEST EUROPEAN SOFTWARE MARKET : '1992 TRENDS

The software market in West Europe is characterised by national differences - including firstly the language one. With nine main languages and a host of minority ones this is evidently not a minor issue. The major ones are:

- German
- French
- English
- Italian
- Greek
- Spanish
- Dutch
- Portuguese
- Danish

A second source of diversity arises from the differing legal and taxation systems which affect a wide range of applications in matters as diverse as Value Added Tax calculation to Payroll charges. There are many other national and local influences which render software produced for one geographical area unsuitable for another including

- Social Security System differences
- Identity and Regulations differences
- Health Insurance Systems differences
- Educational System differences

Many of these differences will not be swept away by the Single Market. Thus for many Vertical Market Applications Software - Package and Bespoke - there are, and will remain, many market differences for the foreseeable future.

In the case of Horizontal Applications Software, however, such as Spreadsheet, Database and Wordprocessing systems the reduction of trading barriers should facilitate European wide sales. Similarly, for package software close to the Operating System, the '1992' effect should be positive e.g. for LAN software package suppliers.

The overall impact of '1992' should therefore be helpful to many software marketeers operating in the 'horizontal business' sector but not necessarily for others.

'1992' trends include:

(a) Greater spending by 'Pan-European' bodies on systems - sometimes multi-sourced (e.g. by Consortia)

- European Commission
- European Space Agency

(ii) Creating the 'European Nervous System'; a new approach from the European Commission on a pan-European basis

(iii) Regional Markets Versus National Markets in which the West European market is divided into regions for marketing and sales purposes on the basis of Geography, Language and other rational factors, rather than country boundaries e.g.

(A) German Region, including Scandinavia & Flemish Belgium

(B) Latin/Mediterranean Europe

(C) U.K. & Ireland

These and other trends make the West European Software Market of the 1990s likely to be an interesting one.

IV THE WEST EUROPEAN SOFTWARE INDUSTRY

The industry is - as elsewhere - a fragmented one consisting of the following main types of supplier:

(a) Computer System Manufacturers e.g. IBM SIEMENS etc. These account for a very large proportion of software sales.

(b) Large Software Houses there being now well over 20 with over 1000 staff but only a handful over 5,000. These are increasing rapidly.

(c) Small & Medium size Software Houses

(d) Freelancers

The industry is dominated by U.S. suppliers in certain countries e.g. the U.K., whereas in others it is National/European e.g. France and Germany.

Overall in Western Europe the industry top ten is dominated by French companies, the largest by far being CAP GEMINI SOGETTI.

The industry is changing rapidly, with a very high rate of acquisitions and mergers as well as new company start-ups. The process of industry concentration is expected to continue through the 1990s as part of a global process.

The Japanese role is currently very low except in the Hardware & Systems Manufacturing sector. However, there are rumours that they are trying to acquire at least one large U.K. company as a means of entry into this market place. Already, there are small Japanese software houses in Europe due to major Japanese Clients bringing their 'software component' suppliers with them.

As in the case of the market, it can be expected that there will be many changes in the supplier industry, with some of the major developments occurring in West Germany, the 'sleeping giant' of the European Software industry.

V THE SOFTWARE QUALITY PROBLEM IN WEST EUROPE

As part of a British Government commissioned study of software quality problems in 1988, Price Waterhouse estimated that poor quality software cost the U.K. #500 Million per annum. This estimate referred only to software sold to customers and thus excluded the costs of poor quality software developed in house. The estimate attracted considerable interest, a common view being that the figure was probably an underestimate. However, even taking it as approximately correct would indicate (since the U.K. accounts for some 20% of the total W. European market) the corresponding figure for West Europe of at least #2,500 million or US$ 4 Billion per annum.

Figures for the cost of low quality software for all software would be substantially higher.

However, regardless of any figure there is no doubt that poor software quality is a subject of serious concern in Western Europe. This is apparent in sectors as diverse as

(a) Safety-Critical software

(b) Financial software

In general there is increasing concern about the impact of poor quality software as regards its influence on the defence industry and commerce and other sectors in Western Europe. Viewed as a behavioural process there was an upward shift in the 1980s along the reaction curve shown in Figure V.I.

A range of responses have already been initiated including

(a) 'Constructive' quality assurance as it is termed i.e. the introduction of CASE methods and tools as well as other 'best practice' software development approaches

(b) Greater 'user' control of software development management

(c) Introduction of software quality assurance approaches in suppliers

(d) Independent Validation Verification & Testing of software

(e) Standards for Software Product Certification

(f) Software Quality Assurance standards which are used to assess suppliers software quality management systems.

It is the latter developments which have caused concern outside Western Europe that a protectionist or 'Software Fortress' Europe may be developing, and thus is given particular attention in this paper.

VI DIFFERING QUALITY STANDARDS PHILOSOPHIES IN WESTERN EUROPE : 'PRODUCT' & 'PROCESS'

There has been a focus on product quality standards in Western Europe for many years. A driving concern was safety, as shown for example by the establishment of the German TUVs over a hundred years ago with a prime aim of testing the safety of steam pressure vessels.

Product quality standards were thus developed for a wide range of items. However, another view was developed that 'product' quality assurance was not always appropriate or even possible. This view was taken by powerful purchasers of software, in Western Europe as in the U.S. led by defence purchasers. The basic reason was that complex software was (and is) impossible to test thoroughly and that therefore the only alternative was and is to 'assure' that the development process was properly conducted. This has however led to a 'Euromountain' of software quality standards as indicated in Figure VI.1. This has created problems for suppliers by presenting them with a confusing set of standards to follow and multi-assessment/certification.

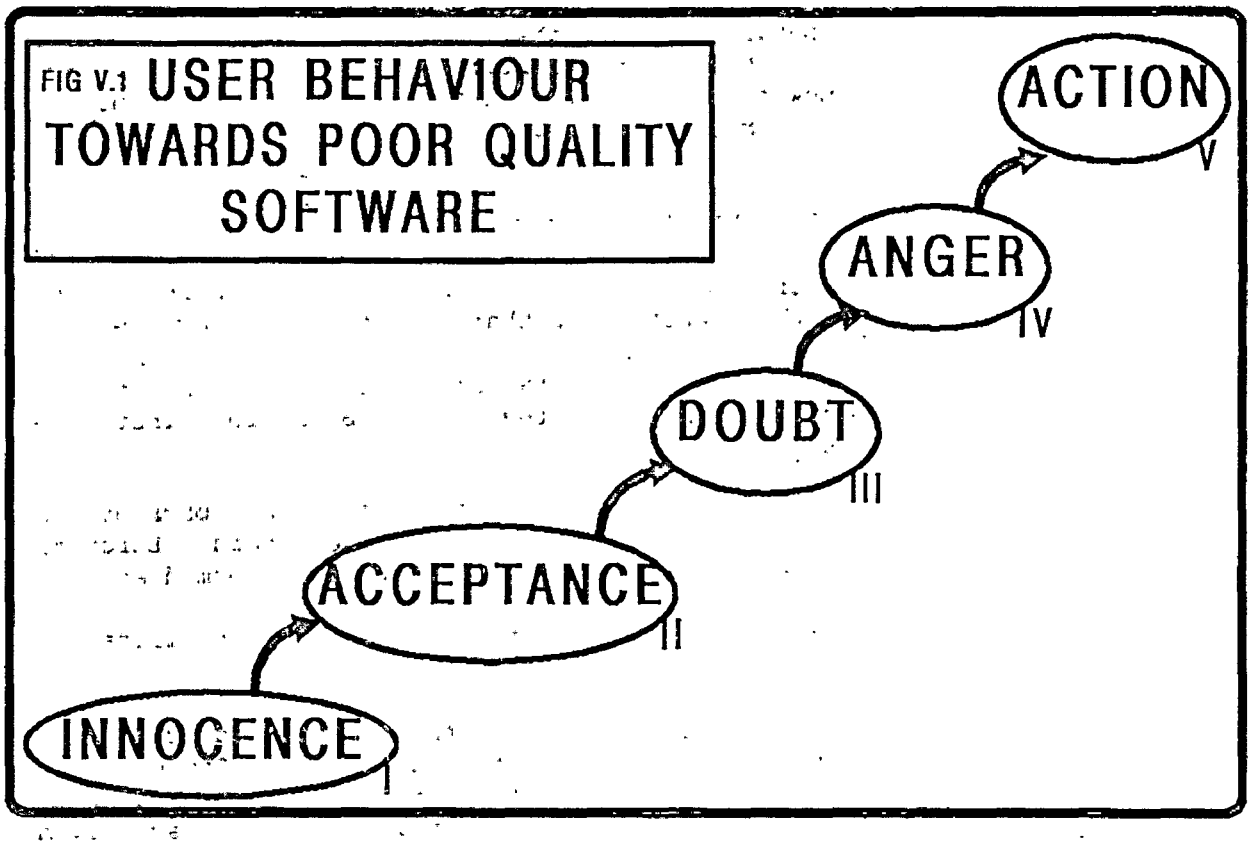
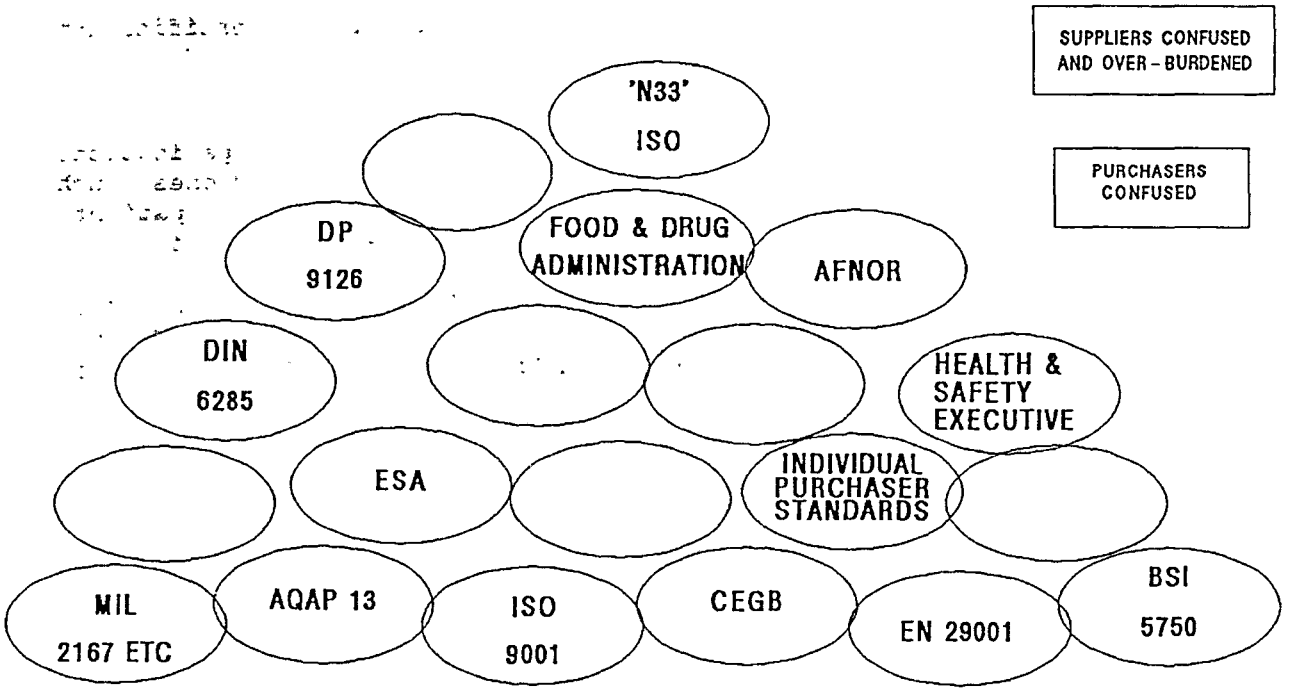


FIG VI.1

A 'EUROMOUNTAIN' OF SOFTWARE QUALITY ASSURANCE STANDARDS



Notwithstanding the above it should be noted that the West Germans, with their strong emphasis on product standards and certification (DIN Norms) have developed a German standard now being submitted to the International Standards Organisation (ISO) for Software Product Testing.

VII EEC & THE REDUCTION OF INTERNAL BARRIERS TO TRADE

The possible consequence of differing national interpretations and certification mechanisms for the Standards is that certification awarded in one country - even against the same International Standard - might not be accepted in another e.g. AFNOR, the French Standardisation body, might not accept as equivalent a certificate issued by the British Standards Institute (BSI).

Therefore in order to eliminate this type of potential technical barrier to trade the European Commission and CEN/CENELEC the main European Standardisation organisations have been working towards what is termed as

'Harmonisation & Mutual Recognition' across Western Europe of Certification practices and procedures.

In the quest for this desired state of affairs - which is perceived as a more attainable goal than any efforts to go from 12 National Standards Bodies to a single 'European Standards Institute' (Perhaps in timeframe from 2000 - 2010!?!) there are a number of new organisational entities which have even Europeans in the field confused e.g.

- ECITC - European Committee of Information Technology Certification
- EOTC - European Organisation for Testing & Certification
- EQS - European Quality System Assessment and Certification Committee

When added to the apparent plethora of longstanding, quality organisations such as EOQC ('European Organisation for Quality Control') and new ones such as EFQM ('European Foundation for Quality Management) even patient, understanding participants and observers may be understandably confused.

The targeted result however is a very desirable one - the reduction of such technical barriers to trade within the European Community. Hopefully this will extend to EFTA countries also (European Free Trade Association i.e. Austria, Switzerland etc).

These moves - once achieved - should also benefit external suppliers including in particular North American and Far Eastern ones, who presumably find it even more confusing and difficult than the Europeans.

VIII SOFTWARE QUALITY ASSURANCE STANDARDS & CERTIFICATION/ASSESSMENT AUDITING IN W. EUROPE

THREE CASE STUDIES

Case A: A General Software Quality Assurance Standard

In 1987 the International Standards Organisation (ISO) published a standard for Quality Assurance:

'Quality Systems - Model for Quality Assurance in Design/Development, Production, Installation and Servicing'.

Confusingly, this not only is numbered differently in the various EEC countries e.g. BS 5750 = ISO 9001, but also the same standard is numbered differently by CEN/CENELEC i.e. EN 29001, so that in fact

ISO 9001 = EN 29001 = BS5750 = ...

This standard is used not only to provide requirements for contractural situations between suppliers and customers, but in an increasing number of European countries as a basis for certification by 'third parties'. We may distinguish between

(a) 'Second Party': Assessment/Certification/Auditing

and

(b) 'Third Party': Assessment/Certification/Auditing

In (a) a purchaser, typically a very large purchaser such as the Ministry of Defence (MoD) audits the supplier to ascertain whether or not they are compliant with the appropriate standard, which in the case of the MoD is the NATO standard AQAP 13.

In (b), an independent third party such as British Standards Institute (BSI) Quality Assurance carries out an assessment to check whether the supplier is compliant with the BSI equivalent of ISO 9001 which will be discussed below.

General acceptance of ISO 9001 instead of a multiplicity of standards would eliminate the problems of the 'Euromountain'. There is however a problem regarding software with ISO 9001 since it is manufacturing oriented and not easy for software people to use. This problem led to the establishment of an ISO/IEC working group led by Brian Young of Canada to develop an additional standard based on ISO 9001 which would complement the latter for the case of software. After three years of hard work a Draft International Standard ISO 9000-3 has been produced:

'Guidelines for the Application of ISO 9001 to Software'

It is hoped that this will serve the need to provide software sector guidance for the application of ISO 9001 without however placing additional compulsory requirements on suppliers. Since it is 'only' a Guideline document ISO 9000-3 will not have the same authority as ISO 9001 which is aimed at providing requirements for contractual situations which must be met if compliance with the standard is to be achieved. However, it is still an open question as to the degree to which powerful purchasers will wish to use it as a basis for obtaining greater confidence in their software suppliers.

Although this new standard has been produced with significant Japanese [N.B. The basis was provided by the Japanese] and U.S. involvement there has been concern expressed that it may serve as an instrument of protectionism by the Europeans. This issue is discussed further below.

In Europe it is hoped that this new standard will eliminate the Software Quality Assurance Standards 'Euromountain' and replace it with - if not one - at most a very small number of closely related standards.

Case B: A Safety Critical Software Quality Assurance Standard

In 1989 an IEC (International Electrotechnical Council) Working Group produced two working documents on systems and software development for safety critical applications. These documents are at an early draft stage, lengthy, comprehensive and build on previous sector-specific work for example in the Nuclear industry. The concern for safety is very high in West Europe as observed by several U.S. commentators. A strong European involvement in this IEC effort is reinforced by major efforts at the national level in Europe e.g. the U.K. and W. Germany. The efforts in the U.K. are driven by a combination of MoD and the Health and Safety Executive, a Government department, whereas the German efforts reflect industry concern to produce safety-critical systems in a rigorous engineering manner.

The IEC documents are due to undergo a public comment and redrafting process this year. It is unclear how long it will be before they reach the same level of international agreement as the ISO DIS 9000-3 (with which they need to be made compatible), but it is certain that there will be continuing pressure - including media - for attention to safety-critical software (e.g. AIRBUS software).

Such pressure should not affect high-quality software developers which continually improve their practices to produce software of the highest possible safety levels. However, it is a clear intention of the people involved in these standards making activities to try hard to protect the population at large from the threat of faulty safety-critical software.

Case C: Healthcare Software Quality Assurance

Currently only the U.S. Food & Drug Administration (F.D.A.) has issued draft Guidelines, a document which serves as a basis in Western Europe (as well as in the U.S.A.) for auditing of software in medical equipment. However, two current European Commission projects under the Brussels AIM R & D Programme (Advanced Informatics in Medicine) are directed at producing draft standards for submission to the International Standards making procedures.

- MASQUES : Medical Application Software Quality Enhancement by Standards

- QAMS : Quality Assurance of Medical Software

The MASQUES approach aims for maximum possible compatibility with both the FDA draft guidelines, ISO 9000-3 and also to serve as a basis for intercepting the future IEC safety-critical standards since some poor medical equipment software may be dangerous. QAMS aims to produce a standard based on ISO 9001. Thus the two projects which collaborate closely should cover the main possible routes for international standards making in this sector. Even more so than the general safety-critical field, this work is at a very early stage but the existence of the FDA Guidelines may make this process faster. It would be unjust to view this work as directed at creating a Software Fortress Europe.

IX EASTERN EUROPE & THE 'OPEN FRONTIERS'

The momentous development in Eastern Europe should bring major changes to the European Community:

- East Germany is regarded by many as being to most intents and purposes a part of West Germany and thus also of the European Community

- Hungary and Czechoslovakia are particularly keen on early entry to the Eureapean Community

These and other possible changes should make the European Community larger and, hopefully, stronger after a reconstruction period.

The need for improved software quality is doubtless just as high in Eastern Europe as in Western Europe.

In the short term COCOM rules will prevent many types of software to be exported to Eastern Europe. COCOM indeed may be regarded as a strong rampart of Software Fortress Europe, but like the Berlin Wall aimed at preventing flow outwards, not flow inwards.

Overall the impact of the 'peaceful revolutions' of 1989 in Eastern Europe is not expected to make West Europe more protectionist.

X CONCLUSIONS

As Colchester & Buchan (1990) observe:

> 'On balance, it is still fair to say that Europe is building a more open internal market without becoming more protectionist to the world outside'.

Looking ahead to '1992' the principal conclusions are as follows:

(1) The spectre of a Software Fortress emerging in Europe in '1992' is a valid concern only for software suppliers which are Unethical (e.g. Software 'Pirates') or Low Quality Producers (N.B. especially in the Safety-critical field).

(2) For non-European Community ethical suppliers of high-quality software there should be no particular problems arising from '1992' - except that European based suppliers should not have the current internal barriers and thus should be more competitive all across Europe.

(3) Europe is opening up Eastwards, but no new 'Atlantic Wall' is likely.

(4) Co-operation with high quality U.S. software producers will be actively sought, both for inward investment and R & D collaboration.

Therefore '1992' should be seen as an opportunity for high quality U.S. Software producers.

BIBLIOGRAPHY

COLCHESTER, Nicholas & Buchan, David. EUROPE RELAUNCHED

Hutchinson Business Book London 1990