Looking for a test comprehensiveness measure

Robert M. Poston,
Software Standards Editor

Many of us have studied and used the popular cost-estimation models available today: Barry Boehm’s COCOMO, Lawrence Putnam’s SLIM, and RCA’s PRICE S. All these models tell us that more than 40 percent of a project’s total resources should be set aside for testing.

However, these models do not subdivide testing costs into detailed categories so we can see the proportion of resources that should be allocated to (1) demonstrating the product functions as specified under testing conditions, (2) observing deviations from expectations (incidents/fails), and (3) isolating and eliminating the cause of the failures (faults/bugs).

Experience tells us that activity 3, commonly called debugging or rework, will consume most of the testing resources and will be dependent on the number of software failures discovered during testing. The number of failures discovered will depend on the number of faults in the code and the comprehensiveness of the testing.

COCOMO, SLIM, and PRICE S are not modeled to include fault counts or test comprehensiveness. Information from a reliability model (also known as a fault count prediction model) needs to be added to information from a cost-estimation model to provide a more accurate cost picture.

John Musa at AT&T Bell Laboratories has established a reliability model. It uses measurements of failures in a product to predict future failures in that product. If it were used on several similar projects in one organization, the collective information from these projects could be used to estimate failures for a startup project.

Putting together information from these two types of models brings us closer to a satisfactory estimation of project performance, but we still have gaps in the information. The biggest gap concerns the comprehensiveness of test coverage.

Approaches to determining test case comprehensiveness have been successful in the development of safety-oriented, highly critical products where cost was not the primary consideration. Unfortunately, both of these approaches do add considerable time and cost to a project, and, therefore, are not practical in most software environments today.

So we still lack a test comprehensiveness measure that does not add substantially to the cost and schedule of a software project. The IEEE has in the works standards on productivity (cost estimation), reliability, and a unit testing process. But we do not have a working group established to address a test comprehensiveness measure.

Should such a working group be formed? Is such a standard realizable? I have been using a program test comprehensiveness measure.* The measure may be of interest to those of you thinking about the problem. The TC equation I employ includes factors for functional requirements coverage, input domain coverage, output range coverage, and program structure coverage:

\[
TC = \frac{\text{Requirements} \times \text{Inputs} \times \text{Outputs} \times \text{Structure}}{\text{POC/TRC} \times (\text{PIC/TIC}) \times (\text{PAU/TAU})}
\]

To understand the whole equation, let’s look at its factors one by one. For purposes of illustration, please refer to Figures 1-5 as we progress through the equation.

The functional requirements are a description of what the product is supposed to do. The requirements factor in TC is passed requirements (PR) divided by total requirements (TR).

TR is the most important factor in the equation. It is established at a formal review or inspection by the end user, the developer, and the tester of a product. In that review, the three parties must agree that the functional requirements are complete, consistent, and correct—and they must sign a statement to that effect. TR for a sample program called testform can

---

*Editor: Robert M. Poston
Programming Environments, Inc.
68 Monmouth Road
Oakhurst, NJ 07756
Compmail + r.poston

---


---

Robert Poston and Dwayne Knirk of Programming Environments, Inc., have evolved a test comprehensiveness measure in response to project demands from several different companies.

---

Standards bulletin board

Software Life Cycle Process Standard Working Group

The working group has undertaken the task of defining the processes occurring within each software life cycle phase. This includes both the traditional phases and the less traditional phases of concept exploration, operation, maintenance, and retirement.

The goal is to provide a software standard that represents the industry consensus. Interested people are invited to join.

The milestone schedule is:

- First draft produced: Feb. 1986
- Publication of draft standard approved: Oct. 1986
- Draft standard published: Jan. 1987
- Comments on draft standard due: Feb. 1987
- Final draft approved: Sept. 1987
- Balloting letter sent: Jan. 1988
- Approval by IEEE Standards Board: Sept. 1988

The next meeting will be held February 18-20, 1986 in Los Angeles, California, and June 10-12, 1986 in Washington, DC. For further information, contact David J. Schultz, Computer Sciences Corp., 4600 Powder Mill Rd., Beltsville, MD 20705-2666; (301) 937-0760.

---

Standards status reports

The Computer Society publishes a quarterly standards status report that summarizes the standards and project approvals made in each quarter. It also lists information about working groups, publications, memberships, and conferences.

For further information or to be added to the mailing list, contact Fletcher J. Buckley, RCA, MS Plaza Office Complex West, Moorestown, NJ 08057; (609) 866-8832.

---

76 IEEE SOFTWARE
be seen in Figure 1 as having a value of 22 (R22, last requirement in Figure 1).

When a tester confirms that the actual output of a test is within the allowed tolerance of the expected output, that test case passes. When all test cases that exercise one requirement pass, that requirement passes. PR is the simple count of all.*

The input domain factor takes a little more work to derive than does the requirements factor. Because it would be impossible to test all permutations and combinations of inputs, the input domain must be partitioned into testable groups of inputs. Input space can be partitioned into testable groups in two ways: by data organization structure and then by input class.

We know that software will handle different data organizations such as primitives and structures differently. So we will test those organizations as unique entities. Inputs to our sample program, testform, are itemized in Figure 2. A separate entry is provided for every primitive and structure. In addition, each primitive has been given a data type such as an integer, enumerated value, or character.*

We will break each input's domain into five classes: normal, boundary, adjacent, troublesome, and invalid. The normal input values are those that are valid and do not lie on a boundary.

*This concept was partially derived from the work of Glenford Meyers, especially in his The Art of Software Testing (John Wiley and Sons, New York, 1979).

---

**Figure 1. Functional requirements for testform.**

<table>
<thead>
<tr>
<th>R01. Initiate processing on the Unix Command:</th>
<th>R11. Eliminate O-BREAKs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXPOS, B LIMIT</td>
<td>R12. Replace all R-BREAKs with SPACE.</td>
</tr>
<tr>
<td>Provide a default value for MAXPOS if a value is not included with the Command.</td>
<td>R13. Preserve the relative order of input CHARACTERS and WORDS.</td>
</tr>
<tr>
<td>Perform the ERROR POLICY if MAXPOS is not acceptable.</td>
<td>R14. Preserve the structure of WORDs (do not break words).</td>
</tr>
<tr>
<td>Provide a default value for B LIMIT if a value is not included with the Command.</td>
<td>R15. Produce LINE(s) that have as many WORDS as possible and no more than MAXPOS CHARACTERS.</td>
</tr>
<tr>
<td>Perform the ERROR POLICY if B LIMIT is not acceptable.</td>
<td>R16. Produce O-STREAM with as few LINEs as possible.</td>
</tr>
<tr>
<td>Accept I-STREAM from standard input (keyboard).</td>
<td>R17. Produce O-STREAM to standard output (CRT).</td>
</tr>
<tr>
<td>Perform the ERROR POLICY if I-STREAM is not present.</td>
<td>R18. Return a COMPLETION CODE to standard error output (CRT) for normal completion if all inputs were acceptable and terminate processing of testform.</td>
</tr>
<tr>
<td>Perform the ERROR POLICY if I-STREAM is empty (&lt;EOT&gt; only).</td>
<td>R19. ERROR POLICY If any input values are not acceptable:</td>
</tr>
<tr>
<td>Perform the ERROR POLICY if a BREAK has more than B LIMIT CHARACTERS.</td>
<td>R20. Produce O-STREAM to standard output (CRT) for all input up to and including the unacceptable input.</td>
</tr>
<tr>
<td>Perform the ERROR POLICY if a WORD has more than MAXPOS CHARACTERS.</td>
<td>R21. Return to standard error output (CRT) a COMPLETION CODE corresponding to the unacceptable input.</td>
</tr>
<tr>
<td>Perform the ERROR POLICY if a BREAK has more than B LIMIT CHARACTERS.</td>
<td>R22. Terminate processing of testform.</td>
</tr>
</tbody>
</table>

---

**Figure 2. Input data dictionary for testform.**

```
MAXPOS = primitive
data type = integer
acceptable values are from 1 to 132, inclusive
default value = 80

I-STREAM = structure
  = (O-BREAK) + WORD + 0[R-BREAK + WORD]n
  + (O-BREAK) + <EOT>
shortest accepted structure = WORD + <EOT>
longest accepted structure = a total of 5,000,000 CHARACTERS

WORD = structure
  = 1[WORD-CHARACTER |value(MAXPOS)]

WORD-CHARACTER = primitive
data type = CHARACTER
acceptable values are limited to the CHARACTERS with ASCII values between 033 and 126 inclusive (printable ASCII characters).

O-BREAK = Optional BREAK
R-BREAK = Required BREAK

BREAK = structure
  = 1[BREAK-CHARACTER |value(B LIMIT)]

BREAK-CHARACTER = primitive
data type = enumerated
acceptable values = [ <HT> | <SP> | <NL> ]

B LIMIT = primitive
data type = integer
acceptable values are from 1 to 5,000,000 inclusive
default value = 2000

<HT> = primitive, ASCII value = 009 (tab character)
<SP> = primitive, ASCII value = 032 (space character)
<NL> = primitive, ASCII value = 010 (new-line character)
<EOT> = primitive, end-of-text delimiter, provided by the operating system

CHARACTER = ASCII standard character set codes. Values are between 033 and 126 inclusive, plus 009, 010, and 032. All other character values are trapped or filtered by the operating system and will not be input to testform.
```

---

November 1985
expected to be processed in exactly the same way by the code.

Sample input class lists are provided for a primitive data element and a data structure for the testform program inputs in Figures 3 and 4. The input domain factor is calculated by first counting all the class entries for each input and then by totaling all the counts to derive total input classes (TIC).

Just as the end user, the developer, and the tester established agreement about the list of requirements in TR, so must those three parties agree that input classes included in TIC are complete, consistent, correct, and sufficient for determining test cases.

Passed input classes (PIC) is the count of all distinct input classes for which all associated test cases have passed. The input domain factor is simply PIC divided by TIC.

Figure 5 provides a description of the output for our sample program. Partitioning the output range is performed exactly like partitioning the input domain. Total output classes (TOC) is the count of all distinct output classes. Passed output classes (POC) is the count of all distinct output classes for which all associated test cases have passed. The output range factor is POC divided by TOC. TOC is formalized during a review or inspection just as TR and TIC were formalized.

Before we discuss the structure factor in the equation, we should sidestep here to consider several important points. The factors covered so far can be derived from the requirements for the product. A very complete set of test cases can be synthesized from this information alone. On several projects, I have seen a developer generate a highly comprehensive set of requirements-driven test cases—before the design of the program was initiated. If a
designer knows how a product will be tested, he is very likely to design the product to pass the test.

Our input class categories were pulled from the most probable errors list. Therefore, the designer can design out all of the most probable errors. In IEEE Software Standards Seminars, in exercises on our sample program textform (Figures 1-3), class participants created test cases without seeing code. When the test cases were exercised, more than 95 percent of the decision to decision paths were covered.

Structure is the final factor in the equation (see Figure 6 for notation standard). There are many layers of software in any software product. For our purposes, let’s use the term architectural unit, where the architectural unit is the next lower level building block in the layering of software.

For example, in a system, the architectural unit is a subsystem. In a module, the architectural unit is a decision-to-decision path. In a decision-to-decision path, it is an executable line of code.

Total architectural units (TAU) is the count of all distinct building blocks of the item under test. Passed architectural units (PAU) is the count of all distinct building blocks that have passed their associated tests. The structure coverage factor is PAU divided by TAU.

Although I’ve had success with this measurement, its evolution certainly should not stop here. Mathematically, the parameters that go into the factors in TC are very subjective and are highly dependent on the agreement of the end user, developer, and tester of the product. Test comprehensiveness is not the same as test completeness.

The input domain factor and the output range factor represent a minor sampling of the total possible input and output values. The structure coverage says merely that each lower level building block was exercised once, not that it was exercised in all permutations and combinations.

As now used, the structure coverage factor does not compensate for data flow through the product. It only addresses control flow. But, even with these shortcomings, many software engineers find TC a useful measure in determining when sufficient testing has been performed.

Prior to using TC, “Testing is complete” usually meant the money was used up and the schedule had run out. After applying TC, the simple statement, “TC equals one,” was meaningful and definitive. Because of my experience with this homegrown metric, I believe a standardized test comprehensiveness measure is realizable in our industry now.

What do you think?

---

**A C compiler for the Macintosh**

Abbas Birjandi, Oregon State University

AztecC by Manx Software Systems is a C language compiler for developing programs on the Apple Macintosh. It is offered in three versions: the C68K-c commercial system, the C68K-d development system, and the C68K-p personal system. Since the versions differ mainly in the availability of some tools and license rights, I have chosen to review only the C68K-c, version 1.06F, as representative of the group.

---

**The AztecC compiler supports a nearly complete version of C.**

AztecC provides the user with a Unix shell-like environment for developing programs. Version 1.06F supports a limited version of Macintosh menus and desktop capabilities. It runs on both 128K and 512K systems. For serious development, an extra drive is needed.

The C68K-c comes in three disks. The first contains a Manx-supplied shell, C compiler, assembler, linker, and Vi-like text editor called Z. (Vi is a popular page- or line-oriented editor that runs under Unix.) The first disk also contains the support libraries and C header files. The second disk contains a rich set of utility programs like source archiver, diff, make, grep, and a set of various examples. The third disk contains an Apple-supplied debugger, a resource maker, Edit (an Apple-supplied editor), and a program to support a serial driver. It also contains a source archive of the tools in the system. Manx provides its own resource maker in its latest version.

The system employs a nonoffensive copy-protection mechanism. At startup, the shell checks for a key disk. Afterwards, there is no need for such a check unless a cold restart is done. I found that this feature is really not bothersome compared with other copy-protection mechanisms.

The AztecC compiler supports a complete version of C except for bit fields, enumerated data types, and structure passing. The data types and size in bits are as follows:

- char: 8
- int or short: 16
- long: 32
- float: 32
- double: 64
- pointer: 32

Float and double use IEEE formats for floating and double-precision numbers. With 32 bits, the pointers provide the full addressability of Macintosh memory.

Compiling is done in two stages. The C program is first translated to assembly language and then to object code by the assembler. The assembler, which is a full implementation of the 68000 instruction set, can be used to write stand-alone assembly programs as well. The compiler also allows one to save assembled code generated along with the C source code for possible manual optimization and inspection.

---

**Development environment**

As mentioned earlier, AztecC has its own shell and does not really provide a complete Macintosh-style user interface. Those used to the Unix environment find themselves at home, but for fanatics of the Macintosh interface, it is bad news—almost. The shell does provide