Test planning

by R. DEAN HARTWICK
Logicon, Inc.
San Pedro, California

ABSTRACT

The test procedures and program verification methods that should be used in planning a software development are presented. Planning considerations cover initial determination of test objectives, test planning criteria, the use of test tools based upon the anticipated design and application error set, and test standards. The presentation is directed to the manager who has had some software experience and wishes to be thorough in preparation. The test methodology stressed is test being performed parallel to program development, starting with an early (prior to code generation) analysis of program requirements and specifications, followed by static analysis of source code, execution analysis of computer program subelements, and integrated system testing. Included is a discussion of automated tools now being used to relieve test analysts of tedious code analysis tasks. Results of a study in which errors from 11 previous verification and validation projects were collected and categorized by severity and functional type are presented.

INTRODUCTION

The testing of computer programs is concerned with finding program errors that will unacceptably degrade program performance. Test planning often consists of organizing an increasingly more complicated series of tests after program development, following the rather simplistic notion that: “Errors will be introduced during program analysis, design, and coding and may be found (so that they may be eliminated) by program testing.” In contrast, a well-planned test approach will start in parallel with program development, making it possible to introduce design and coding constraints that will assist testing, to plan the testing effort properly, and to detect errors at the earliest possible time so that they may be corrected with minimum time and effort. This concept looks at testing as a means of safeguarding program quality rather than as a way of measuring program errors.

This approach is desirable for two reasons: to lower the likelihood of budget and schedule overruns, and to minimize testing costs. Many computer programs have been developed late, cost more than budgeted to complete, and have worked so poorly as to degrade entire system performance; examples are contained in the well-known CCIP-85 study. By catching design errors earlier, major iterations in program development may be reduced, lowering the probability of schedule and budget overruns. An additional advantage is that building quality mechanisms into a program should improve its performance when fielded. With regard to testing costs, these are frequently reported to be in the vicinity of 50 percent of the entire computer program development cost. Considering test requirements early makes it possible to prepare a more efficient plan in which appropriate test tools and personnel are identified. Additionally, finding design errors at an early stage saves time for both program development and testing.

The remainder of this paper discusses how such a well-organized test plan is prepared. Considerations include the testing methodology to be employed as a function of phase of the program development, the selection of test tools to be used, and the organization and control of the test effort.

TEST OBJECTIVES

As software systems have grown in size and complexity, more programmers have been required, more personnel interfaces have been introduced, and more opportunities for error have been created. All of these factors have led to a nonlinear increase in the number of ways a computer program can malfunction, and the resulting software horror stories have received a far better press than the systematic science of computer program testing developed to prevent them. This systematic science has been given several different names, including Independent Verification and Validation (IV&V). The concept of IV&V has become widely accepted within the Defense Department and has proven its value; remarkably few software developments have failed to achieve their desired quality or have violated schedules and budgets when IV&V was used. The essential difference between IV&V and a formal test group is managerial. IV&V is performed by independent personnel employing independent test tools and techniques. Although this distinction will not be maintained hereafter and the remaining discussion will strictly address test methodology, IV&V should always be considered as a technique to be used in any new software development, even though its cost may seem to be high (typically 15 percent to 20 percent
of the total software development cost). The immediate out-
of-pocket expense associated with doing IV&V must be weighed against the assurance of quality and performance gained when it is used.

The fundamental step in preparing test plans that will help achieve high computer program quality is to state the test objectives at the outset of the development. The test objectives, which are independent of program size, significantly influence what test methods are to be applied to a given program, that is, what types of errors must be detected by testing. Test objectives encompass two different purposes: testing to gain a measure of how well the computer program will achieve objectives (performance evaluation) and testing to assure that only intended functions will occur (assurance analysis). These two purposes overlap somewhat, as shown in Figure 1, and the type of application influences the amount of effort devoted to each.

Performance evaluation establishes a measure of how well the program performs its intended functions. This measure will be a function of the application, user objectives, and expected program longevity. Some considerations in this respect might be: efficiency (timing and memory), maintainability (ease of modification), accuracy (numerical and logical), compatibility with system and user (convenience, vulnerability), and testability. The following definition of performance evaluation should prove helpful: Performance evaluation is a determination of the extent to which the examined software:

- Satisfies system requirements
- Satisfies program end item requirements
- Is designed and coded efficiently
- Degrades the performance of the system or the subsystem in which it operates

Performance evaluation will be adequate only to the extent that the system and end item requirements are defined. The process of working from system to end item requirements is also subjected to scrutiny by performance evaluation; for the end item requirements, when combined with requirements for other system components, may conflict with the system requirements. It should be pointed out that the third and fourth items above are very subjective and may involve tradeoffs; inefficiency and system degradation are usually demonstrated by counter-example.

Turning now to assurance analysis, the objective is to show that the computer program performs all intended functions and does not perform unintended functions that could degrade or compromise the safety or security of the system to which it belongs. The definition of assurance analysis might be generalized for all types of computer programs as follows. Assurance analysis is the determination of the extent to which the examined software contains coding which could contribute to:

- Unauthorized access to data files or program
- Unauthorized display of confidential data
- Failure to respond in a timely fashion to critical program conditions
- Operations that present a hazard to equipment or personnel

The objectives of performance evaluation and assurance analysis are to some extent interdependent. There will be program errors which have an impact on safety/security as well as on performance.

The management techniques, analysis tools and techniques, personnel qualifications, and configuration control procedures are basically the same for performance evaluation and assurance analysis. Only by assessing the potential impact of a program error can that error be categorized as related to the system’s performance or to its safety or security. The errors are not distinguished by the means of discovery.

TEST PLANNING AND ORGANIZATION

Testing can be divided into six major management phases whose time-phasing relative to a typical system development cycle is illustrated in Figure 2:

- Program and personnel planning
- Test requirements definition
- Tool definition and development
- Test plan/procedures definition
- Testing and analysis (consisting, as will be shown subsequently, of program concept analysis, static code analysis, and code execution testing)
- Final report generation

As shown in the figure, the phases may overlap. Therefore, to prevent significant management problems it is important that the program and personnel planning phase be completed before undertaking any other phase. Detailed schedules for milestones and supporting activities should be established for the entire effort. These schedules should be detailed enough to provide management with a tool for measuring project progress, but also flexible enough to permit reaction to unanticipated problems. If the schedules have been properly established, project management will be
able to detect potential problem areas before they become critical. The schedules should reflect the following types of information:

- Major project design and technical reviews
- Dates and contents of data package deliveries
- Dates of deliverable data items and other major milestones, allowing for rough drafts, internal reviews, editing, final copy preparation, and review and approval
- Dates for the completion of important test support software tools, including definition dates for the test tool requirements, design flows, description document, and user's manual
- Schedule dates and management approaches to ensure the timely review of activities that can only be scheduled upon completion of prerequisite milestones

Once the schedules have been prepared, each supporting activity can be manloaded and the qualifications of people to staff these activities established.

Test reviews, in which test progress is presented using the activity and milestone charts, should also be planned. Near-term milestones, usually those to be accomplished within the next three months, might require an inch-chart review—a weekly or daily breakdown of activities showing how the near-term milestone will be achieved. The inch-chart review ensures that nothing has been overlooked. Also, potential project pitfalls or failure conditions should be reviewed so that the necessary actions (for example, additional project staffing) can be immediately initiated to head them off.

Test requirements should be generated in the second management phase, test requirements definition. For assurance analysis, this entails clearly identifying program requirements whose violation could compromise system safety or security. For performance evaluation, it entails identifying the program requirements to be examined to measure the quality of program performance. Establishing the scope of what is and what is not to be tested is imperative because testing all program combinations is infeasible. As will be discussed, many of the requirements must result from an analysis of the types of errors that may be anticipated and the test methodology required to detect...
them. A high-level test methodology must be established here. It will then be amplified in defining test plans and procedures.

In the third management phase, tool definition and development, the test tools that will best accomplish the test objectives should be identified. The choice that management makes in its selection of test support software tools will be reflected directly in the quality and productivity of the test group. The advantages of automated test techniques and analysis methods are obvious, but frequently the project will lack sufficient time or money. Once the test support software tools have been identified, the test management team must establish the method of qualifying them, the configuration control procedures to be applied to them, and the schedule for time-phasing them into the analysis and testing effort.

In the fourth management phase, test plan/procedures definition, the details of how each test requirement will be tested and analyzed must be documented. The document should include the following types of information:

- Test support software or hardware test bed to be used
- Test scenarios
- Success criteria
- Detailed procedures for test implementation
- Control of data standards and software deliverables

The last item, control of data standards and software deliverables, is especially important. Test management should institute strict procedures to ensure that the data standards are not replaced with others, tampered with, or destroyed. At the completion of testing and analysis, a final check of all the working copies against the data standards should be performed.

In the fifth management phase, testing and analysis, the management task becomes one of monitoring project performance and maintaining control procedures on all input and output items. Test management should review all discrepancies issued by the test group and evaluate each for correctness. Whenever the changes are made to program requirements, program code, or support tools, a decision must be made whether it will be necessary to repeat tests and analyses already completed. The possibility of retest makes it imperative to maintain careful configuration control on the program requirements, program code, and support software.

As Figure 2 shows, the testing and analysis phase is broken down into three subphases: program concept analysis, static code analysis, and code execution testing. The functional procedures and tools used during each are discussed subsequently under selection of test methodology.

The sixth management phase, final report generation, may include certification of the program for operational use integrally or as a separate step. During the planning for the analysis and testing phase, management should establish procedures for recording results in a form usable for final report generation with minimal modifications. This effort allows a final review of activities to ensure that no test has been overlooked. For future efforts, it would be useful to accumulate statistical information about the number of errors detected, their relative importance, and the methods used to detect them.

ERROR SOURCES AND DETECTION METHODS

Software errors cannot be treated within the confines of hardware reliability concepts, nor can it be assumed that the detectability of computer program errors has profound theoretical considerations. A good test plan is based upon sound software error theory. Every application can be expected to have its own unique set of errors. It is the role of the test group to determine this error set; from it, the type and extent of testing to be performed can be determined.

Considerable work has been performed in accumulating and analyzing data to develop a software error theory. One study that will be illustrated here represented the results of analyzing and categorizing 1202 errors discovered in 11 projects. These results are primarily, but not totally, obtained from IV&V activities and hence do not show how earlier development testing would affect results. The results were categorized by 12 different error types and classified by error severity, as shown in Table I. The severity levels were based on program performance, as follows; the greater the performance degradation, the more serious the error:

- **Catastrophic Error:** A coding error that would be fatal to the application in that it would effectively terminate program execution
- **Serious Error:** A coding or specification error that could severely degrade the program performance but would not be fatal to the application; examples are violations of timing, accuracy, safety, or stability requirements
- **Moderate Error:** An error that would not have major impact on program performance and that would probably not result in a system requirements violation
- **Trivial Error:** An error that would have no impact on program performance, e.g., errors within annotations on the program listing

For purposes of test planning, interesting observations can be made from an examination of Table I. For example, even though errors of the "branch & jump" variety occurred infrequently after checkout (4.5 percent of total errors), they tended to be serious (14.3 percent of catastrophic and 6.7 percent of all serious errors). Thus it may be concluded that any testing methodology to be selected should ensure that this type of error should be well analyzed and accounted for. At the other extreme are the "documentation" errors, which accounted for 7.9 percent of the total errors but no catastrophic or serious errors. This then might influence a test planner to reduce effort put into finding documentation errors in favor of intensifying effort elsewhere. One caveat to any such statistical approach can be seen by looking at the "incomplete or
TABLE I—Error Categorization Summary

<table>
<thead>
<tr>
<th>Error Category</th>
<th>Total Errors</th>
<th>Catastrophic Errors</th>
<th>Serious Errors</th>
<th>Moderate Errors</th>
<th>Trivial Errors</th>
<th>% of Catastrophic</th>
<th>% of Serious</th>
<th>% of Moderate</th>
<th>% of Trivial</th>
<th>% of Total (excluding trivial errors)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data/instruction access &amp; storing</td>
<td>120</td>
<td>6</td>
<td>30</td>
<td>72</td>
<td>12</td>
<td>28.6</td>
<td>20.1</td>
<td>15.0</td>
<td>2.2</td>
<td>16.6</td>
</tr>
<tr>
<td>Equation computation &amp; arithmetic</td>
<td>113</td>
<td>0</td>
<td>22</td>
<td>73</td>
<td>18</td>
<td>0</td>
<td>14.8</td>
<td>15.3</td>
<td>3.3</td>
<td>14.6</td>
</tr>
<tr>
<td>Branch &amp; jump</td>
<td>32</td>
<td>3</td>
<td>10</td>
<td>16</td>
<td>3</td>
<td>14.3</td>
<td>6.7</td>
<td>3.3</td>
<td>0.5</td>
<td>4.5</td>
</tr>
<tr>
<td>Incorrect constant value &amp; data formats</td>
<td>41</td>
<td>2</td>
<td>12</td>
<td>19</td>
<td>8</td>
<td>9.5</td>
<td>8.1</td>
<td>3.9</td>
<td>1.5</td>
<td>5.1</td>
</tr>
<tr>
<td>Violation of programming practices</td>
<td>118</td>
<td>0</td>
<td>2</td>
<td>22</td>
<td>94</td>
<td>0</td>
<td>1.3</td>
<td>4.6</td>
<td>17.0</td>
<td>3.7</td>
</tr>
<tr>
<td>Specification violation due to incorrect implementation</td>
<td>145</td>
<td>0</td>
<td>9</td>
<td>61</td>
<td>75</td>
<td>0</td>
<td>6.0</td>
<td>12.7</td>
<td>13.6</td>
<td>10.8</td>
</tr>
<tr>
<td>Timing &amp; process allocation</td>
<td>44</td>
<td>0</td>
<td>14</td>
<td>25</td>
<td>5</td>
<td>0</td>
<td>9.4</td>
<td>5.2</td>
<td>0.9</td>
<td>6.0</td>
</tr>
<tr>
<td>Interruption &amp; data coherency</td>
<td>45</td>
<td>2</td>
<td>10</td>
<td>32</td>
<td>1</td>
<td>9.5</td>
<td>6.7</td>
<td>6.7</td>
<td>0.2</td>
<td>6.8</td>
</tr>
<tr>
<td>Incomplete or erroneous specifications</td>
<td>340</td>
<td>1</td>
<td>18</td>
<td>82</td>
<td>239</td>
<td>4.8</td>
<td>12.1</td>
<td>17.1</td>
<td>45.2</td>
<td>15.6</td>
</tr>
<tr>
<td>Logic &amp; sequencing</td>
<td>107</td>
<td>6</td>
<td>22</td>
<td>67</td>
<td>12</td>
<td>28.6</td>
<td>14.8</td>
<td>14.0</td>
<td>2.2</td>
<td>14.6</td>
</tr>
<tr>
<td>Documentation</td>
<td>96</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>86</td>
<td>0</td>
<td>2.1</td>
<td>15.6</td>
<td>1.5</td>
<td>7.8</td>
</tr>
<tr>
<td>Erroneous use of system hardware/software</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4.8</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Total                                                                 1202 21 149 479 553

Total % (excluding trivial errors)                                     3.2 23.0 73.8

erroneous specifications” category. Although this class contributed 43.2 percent of all trivial errors, one catastrophic error was found. It takes only one such error to totally degrade the usefulness of the program.

Once the test planner has a feeling for the types of errors that may be found within a particular program and understands the test objectives, he next must consider the test methodology to be used. It is useful to think of this methodology in terms of testable logical groupings within the computer program and the methods that can be used to detect the class of errors.

Table II gives the general test methods used to detect each category of error in Table I. Referring to Table II, it is clear that a variety of techniques may be needed to test even a single module. At a higher level of program complexity, the executive routine, the various functional modules are interconnected. Generally, the executable itself is modular; i.e., portions can readily be separated so that they can be used with the functional modules to form a functional subprocess. The modularity of the executable can generally be demonstrated by use of a flowchart to construct a module-level flow diagram from the actual

TABLE II—Examples of Specific Detection Methods by Error Category

| Error Category                                      | Document Computations | Independent Coding & Simulation | Equation/Algorithm Derivation | Logic Analysis | Edit Source Code | Code Analysis | Execute Code on Real Computer | Exec Code on Simulation | Execute Process Allocation | Perform Branch Analysis | Analyze Test Case Results | Correctness Proof | Flowcharting | Accuracy Study Presentation | % indicates at least one error discovered | Number indicates percentage of total errors in category found by this method |
|-----------------------------------------------------|-----------------------|---------------------------------|--------------------------------|----------------|------------------|--------------|--------------------------------|----------------------------|-------------------------|--------------------------|-------------------------|--------------------------|-----------------|-------------|---------------------------------------------------------------|---------------------------------------------------------------|----------------------------------------------------------------|
| Data/instruction access & storing                   | 90% x                | x                               | x                             | x              | x                | x            | x                              | x                          | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |
| Equation computation & arithmetic                   | x                     | x                               | x                             | x              | 90% x            | x            | x                              |   x                        | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |
| Branch & jump                                       | x                     | x                               | x                             | x              | x                | 70% x        | x                              |   x                        | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |
| Incorrect constant value & data formats             | 50% x                | x                               | 20% x                         | x              | x                | x            | x                              |   x                        | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |
| Violation of programming practices                  | x                     | x                               | 90% x                         | x              | x                | x            | x                              |   x                        | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |
| Specification violation due to incorrect implementation | x                    |                               |                               |                | x                | x            | x                              |   x                        | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |
| Timing & process allocation                         | x                     |                               |                               |                | x                | 90% x        | x                              |   x                        | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |
| Interruption & data coherency                       | x                     |                               | x                             | 70% x          | x                | x            | x                              |   x                        | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |
| Incomplete or erroneous specifications               | x                     | x                               | x                             | x              | x                | x            | x                              |   x                        | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |
| Logic & sequencing                                   | x                     | x                               | x                             | x              | x                | x            | x                              |   x                        | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |
| Documentation                                        | x                     | x                               | x                             | x              | x                | x            | x                              |   x                        | x                       | x                        | x                       | x                        | x               | x          |                                                             |                                                             |                                                                |

x indicates at least one error discovered
Number indicates percentage of total errors in category found by this method
code. Such a diagram helps to demonstrate the correctness of both functional interrelationships (connection by data flow) and physical relationships (overlay of common memory).

Another effect becomes clear from an examination of Table II. It is extremely unlikely that any particular test method will assure a complete assessment of all errors. Even where a particular method is most suited to detect a particular class of error, it cannot be assumed sufficient. For example, even though a source code editor detected 90 percent of all access and storing errors, 12 errors were found by a combination of six other methods. Therefore, the test planner must consider all test methods available to him and select the set of the methodology that will provide adequate test coverage—we have not yet reached the nirvana of having the one magical tool that will find all errors.

SELECTION OF TEST METHODOLOGY

It has often been suggested that the most cost-effective way to test a program is to load it into its host computer, exercise it black-box fashion, using a myriad of inputs, and determine whether its behavior is reasonable. While this approach seems enticing, it is overly simplistic in that it ignores the complexity existing in even very small programs. Further, black-box testing necessarily means that errors will be found only at the completion of coding. As stated earlier, the most cost-effective way to develop software is to find and fix errors as early as possible. Cost-effective testing therefore requires that some of the testing be conducted before any code has been developed. Such testing can be conceptualized in terms of three types (as shown on Figure 2):

- **Program Concept Analysis:** Achieving satisfaction that the program has been adequately and accurately designed
- **Static Code Analysis:** Analyzing code to detect errors before its execution
- **Code Execution Testing:** Testing code by executing it in successive builds to verify correctness

The process may be likened to building a house. Each level of testing builds upon the foundation and structure established by the previous step until the house is ready for occupancy. As is true of the construction process, each phase has its own set of tools and methods.

Program concept analysis

Of the many test-related functions that can be performed before code generation, one of the more important is to assist the program designer in specifying the standards to be followed in designing and writing the program. Consistent coding techniques greatly simplify the problems of learning code and setting up test tools that can be adapted to specific coding standards, for example, the use of standard subroutine entry-exit conditions. Another pre-coding function is to investigate design tradeoffs that might facilitate testing without introducing substantial programming inefficiency. An example is designing a program for a real-time application on a variable-instruction-time computer in such a way that “dead time” occurs long enough before an interrupt to guarantee that the program location where the interrupt will occur can always be precisely determined in testing. By doing this it is possible to ease greatly a very difficult test program with an insignificant loss of computation time.

Other standards that can be applied include:

- **Compatible Numbering System:** Each function performed by a section of code should be traceable to a specific, documented requirement.
- **Consistent Use of Specification Language:** A common and consistent nomenclature should be defined that minimizes the possibility of ambiguous interpretations.
- **Flowcharting Standards:** All flowcharts should use common formats and be consistent through different levels of detail.
- **Programming Standards:** All dos and don’ts for the application should be specified, including parameter-passing techniques, annotation conventions, and ways of specifying symbolic names.
- **Program Change Procedures:** Means of making changes should be developed to communicate changes properly both before and after the program is put under configuration management.

A more directly test-oriented function that can be performed during concept analysis is to evaluate the evolving design to find problems. This activity, sometimes called requirements and specification analysis, essentially duplicates the design process in the way it is done but differs in the goals that are being met. This type of analysis is directed to answering questions such as those contained in Table III.

The methods generally used to perform concept analysis testing are:

- **Documentation Research and Analysis:** Verifying that program requirements are genuine and related to real system requirements, that variables and parameters are consistent across all documents, and that all data can be traced to consistent, coherent sources.
- **Algorithm Analysis:** Verifying that the design will work, is accurate, and truly reflects program requirements. Discrete and continuous simulations are often used to check out concepts and algorithms, particularly where very complex models are being designed. Particularly important for real-time programs is performing independent sizing and timing analyses that will detect significant design problems before the design is frozen.

From the collection of the Computer History Museum (www.computerhistory.org)
TABLE III—Program Concept Analysis Questions

**Requirements Analysis**
- Are the requirements logical?
- Are the applicable physical constraints of the complete system and each subsystem clearly specified?
- Are all the performance and design requirements contained within the document? If so, are they correct?
- Have the input data and output requirements been identified?
- Are the system and all subsystems clearly defined?
- Are the human and software interfaces specified?
- Are all the requirements stated without ambiguity?
- Are the requirements sufficient to realize the system objectives?

**Specifications Analysis**
- Are the program functional flows a true representation of the logical and mathematical operation of the software?
- Are the model interactions and interfaces compatible?
- Are the equations, algorithms, and data that make up the model properly ordered?
- Are the equations, algorithms, models, and modules correct?
- Is the data base accurate?
- Have the correct assumptions been made in equation/algorithmd derivation?
- Are the equations and algorithms mechanized in the program flowcharts the same as those developed in the requirements documentation?
- Have all the program requirements been correctly translated into the equations, data, algorithms, models, flows, logic, and rationale that make up the program specifications?

**Static code analysis**

The first test function always performed is to examine the code for error before actually executing it. This is useful because it is very productive (many errors are found) and also because it can be done quickly (no startup time is needed to make the program flow in a computer). A large body of test tools has been developed to aid in analyzing computer code without actually executing it. In some form or another, these tools all replace the analysis that is done by a skilled programmer in examining the code manually. This manual analysis has been replaced by automated analysis for several reasons:

- It is one of the more tedious tasks that can be placed on programmers.
- Human analysts make errors when doing tedious things.
- Computer time is much cheaper than analyst time.
- Tedious processes can be easily and cheaply repeated to check the effects of minor code changes.
- Code too complex to be retained and understood by the human mind can be automatically decomposed to its simplest representation to facilitate analysis.

The tools that are used are numerous and subject to many variations of definitions depending upon their developer. The following describes a few of the static code analysis tools implemented by the author's organization.

**Comparators** are used to compare the code of one program version to that of another and reveal the differences. The simple one-to-one check provided by use of a comparator is used both to bring out the modifications between an updated program and its baseline version and to demonstrate that all physically different versions of a single program (cards, magnetic tape, and Mylar tape) are identical. Conceptually very simple, these programs are complex. They work very well in setting baseline standards.

**Editors** are used to analyze source code for coding errors and to extract information that can be used to check relationships between sections of code. Among other functions, the error-detection capability determines whether the code: sets and clears flags properly, uses error-prone instruction sequences, sets up calling sequences properly, modifies instructions, attempts to reference or modify restricted data, uses restricted instructions, or contains inaccessible instructions. The second capability of an editor provides a comprehensive cross-reference listing giving information pertaining to references to program data and to the program/subroutine calling structures. Editors work very well in finding mechanical violations of programming standards. They can also be used to flag coding techniques determined to be risky for the application.

**Flowcharters** show, in detail, the logical structure of a program, an aspect not readily apparent from the code. Flowcharters can be used to reconstruct the logic flow of both higher-level source language programs and assembly language programs. The flow is determined from the actual operations specified by the executable statements, not from comments. The flowcharts generated are adaptable for comparison with the flowcharts provided in the developer's program documentation.

**Logic/Equation Generators** automatically reconstruct arithmetic and flowchart assembly language programs. Such a program translates assembly language instructions into a machine-independent microprogramming language and builds the microprogramming statements into a network in which the flow of control is analyzed and arithmetic equations are reconstructed.

**Program Structure Analyzers** are used to analyze the program paths under input control. They allow different types or sequences of code to be specified (e.g., all subroutine calls, all interrupt-related instructions, all extension register operations) and obtain information such as estimates on timing, paths followed, and entry conditions for specific paths.

**Correctness Proofs** establish, in a mathematical fashion, that a given program performs a desired function and halts. The proof technique does this by determining the correspondence between a function in its coded form (in FORTRAN, for example) and the same function presented in the pertinent specification in mathematical and English language descriptions. Operational use of this technique has been limited mostly to manual proofs. Considerable work is being done to automate it.

**Symbolic Program Executors** decompose source code by logically executing it. They provide a capability to express paths in terms of both all necessary conditions to be satisfied in selecting the path and the result of transversing the path. Some symbolic program executors can also be
used to produce a structured representation of an unstructured program.  

**Code execution testing**

In the traditional mode of software testing, the program is tested by actually executing the object instructions, generally on the computer it is being built for, sometimes on emulators or simulators. Code execution testing closely follows the test philosophy normally used for complex hardware, namely, to test in pyramid fashion, emphasizing the thorough testing of subassemblies, then successively larger groupings of these subassemblies until the total end item has been tested. Provided that the prior testing has been adequate and, moreover, that it has not been subverted by the incorporation of insufficiently tested changes, the testing done on the top level is essentially reduced to demonstrating that all of the pieces work together properly.

Testing methodology depends to some extent on the development methodology. Traditional developments employ bottom-up testing, in which increasing aggregates of small programs are tested in succession, generally using program drivers. Top-down developments replace program “stubs” with analytical data to simulate the effect of having executed the program element that replaced the stub. The “build” concept requires that stimulus-response patterns be determined to test out constructs of program elements that implement particular stimulus-response paths.  

The code execution test methodology is applicable to all of these forms of testing. For simplicity, discussion here is limited to the more common bottom-up form.

Unit-level testing is performed at the subroutine level by starting code execution of subroutines as they are verified. Each subroutine is exercised by test drivers that furnish input and parameters, including representative queues. Testing at the subroutine level is performed open-loop and, in general, does not require simulation of the modules or environment with which the subroutine interfaces. Completion of subroutine testing with “live” data should demonstrate that:

- All instructions have been executed at least once
- All error conditions have been tested
- All logic branches have been traced
- All classes of input will be accepted and all outputs can be produced
- Arithmetic results are correct for nominal conditions

Test cases are generated based on an analysis of the program specification. Input data are selected to verify proper handling of the full range of acceptable data and to verify correct action in processing abnormal data. A test driver program typically sets up the input data, calls the module to be tested, and, upon return of control, lists the module output. For routines that are time-critical, the driver is augmented to check timing; the timer can be set either to determine the time to calculate one set of input data or to evaluate average, minimum, or maximum time.

Whenever anomalies are suspected, the driver is patched to call upon system routines that will allow step-by-step monitoring of a process.

The unit level interface tests are designed to verify that subroutines can be loaded and executed together in the computer system and that they properly assume and relinquish computer control. Of particular importance are tests that execute all possible branching conditions between subroutines and modules.

A supplement to actual or simulated code execution is to independently perform each operation intended by a section of code. This procedure consists of following through the calculations of an equation or algorithm. This can be done by hand, but is better done by an automated tool which can swiftly and accurately run many check cases. Using either a listing or a flowchart of a module and the data captured from a specific run, the mathematical calculations or logical processes indicated in the flowcharts (or the listings) are performed and the results compared with the code execution results. The importance of this technique is that subtle coding errors not caught during flowcharting or code analysis procedures are quickly identified.

The most common tools used to test software at module levels are test drivers, simulations, and execution instruments. Many examples of this last class of program exist. They operate by instrumenting software “probes” within the source code. When the program is subsequently executed, statistics showing execution frequency of paths and code are produced. This gives insight into the program behavior and allows analysts to prepare test cases to more fully exercise the program. A more recent trend in many evaluation efforts is to use hardware monitors to gather these data. Inserting probes into computer back panels is somewhat laborious; hardware monitors allow the same statistics to be collected without modifying the computer program. Further, since they operate in a passive mode, they can be used after the program becomes operational to evaluate it and provide data to optimize it.

The sequence of testing individual modules proceeds in order through tests such as the following:

- **Initialization Tests**: The performance of all initialization operations is tested to assure that all indicated initializations are performed and that correct values for all initialized quantities results.
- **Interaction Tests**: All quantities, variables, and system conditions obtained from other modules are examined to determine the sensitivity of the module under test to their possible values or states.
- **Arithmetic Tests**: The precision of arithmetic calculations is checked to discover where insufficient precision is maintained or incorrect arithmetic calculations are performed.
- **Timing Analysis**: The longest and shortest possible execution times for all tests are determined to establish the execution time requirements for the module and to identify potential timing problems.
- **Branch Logic Tests**: The correct branching decision paths for each branch and each closed-loop test case...
are checked. The branch decision paths that are not exercised in any of the normal test cases are identified and their correctness demonstrated by special test cases.

Once the modules have completed individual code execution testing, they are integrated into the complete software component package and tested as a group. At this level, the testing is primarily functional: testing the collection of modules to show that the aggregate satisfies the stated problem. For large-scale software or weapon systems in which software is an integral part of the system, this testing very often takes place in a laboratory containing a copy of the computer(s) and enough equipment to simulate the application with considerable fidelity.

TEST STANDARDS

To assure the success of the test effort, the test group itself must be subject to quality standards. All hardware and software used in the test effort should be qualified and controlled.

The test tools can be qualified by certifying they have been previously qualified and not changed; by calibration against actual system data, other operational standards, or other qualified test tools; or by a simplified form of the more elaborate test procedures described earlier. Whatever the qualification procedure selected, it should be formally documented and approved as a part of the test plan/procedures. Satisfaction that formal qualification has been performed should be obtained before the tools are used and should be documented in the final test report. After qualification, the tools should be placed under configuration control using conventional techniques.

Where complex simulation testing is required, there is a natural tendency to modify the test hardware and software to adapt them to the peculiarities of the particular program segment under test. Any such changes cannot be allowed to compromise test integrity, such as by using different versions of test tools to test different program modules. All modifications to the test environment must be first communicated and approved. Where a test bed is to be established, it must be provided with a usable inventory of test support tools. These tools should also be formally qualified, and controlled to provide certainty as to their content.

One of the more critical aspects of testing is ensuring the correspondence of the test results obtained at different times or by different personnel. At a minimum, test results should contain the following information to allow comparison of results:

- Test designation
- Test purpose
- Specification, option, or feature being tested
- Range of parameter being tested
- Method of test
- Inputs required for test
- Output expected from test
- Estimated time for test
- Criteria for satisfactory completion of test
- Identification of test bed configuration

The final test results and supporting documentation should be bound as a single document and labeled with the name of the routine(s) checked, the test completion date, and the names of the programmer(s) and test personnel responsible for reviewing and accepting the results. Results should be accompanied by such flowcharts, diagrams, equations, and verbal descriptions as necessary to identify and clearly describe each test, including what was done, why it was done, and what specifically was demonstrated. A summary, if appropriate, should preface the test results package.

SUMMARY

The success of a software test program is determined at the outset of the development. If testing is accorded the effort due it because of its cost and ultimate importance, it is possible to achieve working software of high quality within cost and schedule. By contrast, letting a test program evolve subject to the caprices of development problems and schedules gives very little confidence that the test program will be successful in all dimensions.

At the outset of the development, the test objectives should be set and agreed to by the project manager, the development group, and the test group. These objectives should have the concurrence of the user. Test requirements should then be developed based upon the objectives, the application, and the anticipated design approach. A plan to use, and acquire as necessary, appropriate test tools based on the requirements and the predicted error set should be prepared. An arsenal of tools and test methodology exists; test planning should meticulously select that which is most appropriate. A test plan that provides appropriate interfaces throughout the development effort should be prepared. Finally, good practices pertaining to tool qualification and configuration control should be followed.

REFERENCES


