Introduction to Software Verification and Validation

SEI Curriculum Module SEI-CM-13-1.1

December 1988

James S. Collofello
Arizona State University

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Education Program is developing a wide range of materials to support software engineering education. A **curriculum module** identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in *designing* a course. A **support materials** package includes materials helpful in *teaching* a course. Other materials under development include model curricula, textbooks, educational software, and a variety of reports and proceedings.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

SEI curriculum modules may be copied or incorporated into other materials, but not for profit, provided that appropriate credit is given to the SEI and to the original author of the materials.

Comments on SEI educational publications, reports concerning their use, and requests for additional information should be addressed to the Director of Education, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

Comments on this curriculum module may also be directed to the module author.

James S. Collofello
Computer Science Department
Arizona State University
Tempe, Arizona 85287

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

Introduction to Software Verification and Validation

Foreword

SEI curriculum modules document and explicate software engineering topics. They are intended to be useful in a variety of situations—in the preparation of courses, in the planning of individual lectures, in the design of curricula, and in professional self-study. Topics do not exist in isolation, however, and the question inevitably arises how one module is related to another. Because modules are written by different authors at different times, the answer could easily be "not at all." In a young field struggling to define itself, this would be an unfortunate situation.

The SEI deliberately employs a number of mechanisms to achieve compatible points of view across curriculum modules and to fill the content gaps between them. Modules such as *Introduction to Software Verification and Validation* is one of the most important devices. In this latest revision, Professor Collofello has more modules to integrate into a coherent picture than when we released his first draft more than a year ago—modules on quality assurance, unit testing, technical reviews, and formal verification, as well as less directly related modules on specification, requirements definition, and design. We believe you will find this curriculum module interesting and useful, both in its own right and by virtue of the understanding of other modules that it facilitates.

— Lionel E. Deimel
Senior Computer Scientist, SEI

Contents

Introduction to Software Verification and Validation

Module Revision History

Version 1.1 (December 1988)	General revision Approved for publication
Version 1.0 (October 1987)	Draft for public review

Introduction to Software Verification and Validation

Capsule Description

Software verification and validation techniques are introduced and their applicability discussed. Approaches to integrating these techniques into comprehensive verification and validation plans are also addressed. This curriculum module provides an overview needed to understand in-depth curriculum modules in the verification and validation area.

Philosophy

This module provides a framework for understanding the application of *software verification* and *validation* (*V&V*) processes throughout the software evolution process. Typical products of this process are identified, along with their possible V&V objectives. The V&V process consists of numerous techniques and tools, often used in combination with one another. Due to the large number of V&V approaches in use, this module cannot address every technique. Instead, it will analyze five categories of V&V approaches. These are:

- technical reviews,
- software testing,
- proof of correctness (program verification),
- simulation and prototyping, and
- requirements tracing.

For each category, some representative techniques will be identified and assessed. Since the traditional focus of software V&V activity has been software testing, this category encompasses by far the largest number of techniques. Thus, in this module, the software testing category will be further refined to expose the major testing techniques. Further depth of coverage of techniques applicable to unit testing is available in the curriculum module *Unit Testing and Analysis* [Morell88]. An additional module is planned addressing integration and system testing issues. With regard to the other V&V approaches, in-depth modules are currently available on technical review techniques (*The Software Technical Review Process* [Collofello87]) and proof of correctness (*Formal Verification of Programs* [Berztiss88]).

This module also addresses planning considerations for V&V processes, including the selection and integration of V&V techniques throughout the software evolution process.

Objectives

This module and its subordinate, in-depth modules, is intended to support the the goal of preparing professional software engineering students to analyze the V&V objectives and concerns of a particular project, examine the project's constraints, plan a comprehensive V&V strategy that includes the selection of techniques, track the progress of the V&V activity, and assess the effectiveness of the techniques used and that of the overall V&V plan. Typically, the educational objectives of the software engineering educator teaching V&V topics will be more modest.

Possible objectives for the material treated in this curriculum module are given below.

The student will be able to

Knowledge

- Define the terminology commonly utilized in the verification and validation area.
- Identify representative techniques for the five categories of V&V approaches.

Comprehension

- Explain the theoretical and practical limitations of V&V approaches.
- Describe the V&V objectives for typical products generated by the software evolution process.

Application

- Perform particular V&V techniques.

Analysis

- Determine the applicability and likely effectiveness of V&V approaches for particular products of the software evolution process.

Synthesis

- Develop an outline for a V&V plan for a project that reflects understanding of V&V objectives, integration of techniques, problem tracking, and assessment issues.

Evaluation

- Assess the effectiveness of a V&V plan with respect to its objectives.

Prerequisite Knowledge

This module assumes that students have had experience as members of a team working on a substantial software development project. This experience is essential to understanding the importance of software review techniques and integration approaches for verification and validation. Students should also understand the purpose of specifications and how to interpret them. A one-semester software engineering course can provide the necessary background.

Additional knowledge that enhances the students' learning experience is that resulting from exposure to a variety of types of systems: real-time systems, expert systems, etc. This knowledge may help students appreciate the complexity of V&V issues, as well as provide them with additional insights into the applicability of various V&V approaches.

Depending upon the level of coverage to be accorded proof of correctness, some background in formal logic may be helpful.

Module Content

Outline

I. Introduction
  1. Terminology
  2. Evolving Nature of Area
II. V&V Limitations
  1. Theoretical Foundations
  2. Impracticality of Testing All Data
  3. Impracticality of Testing All Paths
  4. No Absolute Proof of Correctness
III. The Role of V&V in Software Evolution
  1. Types of Products
    a. Requirements
    b. Specifications
    c. Designs
    d. Implementations
    e. Changes
  2. V&V Objectives
    a. Correctness
    b. Consistency
    c. Necessity
    d. Sufficiency
    e. Performance
IV. Software V&V Approaches and their Applicability
  1. Software Technical Reviews
  2. Software Testing
    a. Levels of Testing
      (i) Module Testing
      (ii) Integration Testing
      (iii) System Testing
      (iv) Regression Testing
    b. Testing Techniques and their Applicability
      (i) Functional Testing and Analysis
      (ii) Structural Testing and Analysis
      (iii) Error-Oriented Testing and Analysis
      (iv) Hybrid Approaches
      (v) Integration Strategies
      (vi) Transaction Flow Analysis
      (vii) Stress Analysis
      (viii) Failure Analysis
      (ix) Concurrency Analysis
      (x) Performance Analysis
  3. Proof of Correctness
  4. Simulation and Prototyping
  5. Requirements Tracing
V. Software V&V Planning
  1. Identification of V&V Goals
  2. Selection of V&V Techniques
    a. Requirements
    b. Specifications
    c. Designs
    d. Implementations
    e. Changes
  3. Organizational Responsibilities
    a. Development Organization
    b. Independent Test Organization
    c. Software Quality Assurance
    d. Independent V&V Contractor
  4. Integrating V&V Approaches
  5. Problem Tracking
  6. Tracking Test Activities
  7. Assessment

Annotated Outline

I. Introduction
  1. Terminology

The evolution of software that satisfies its user expectations is a necessary goal of a successful software development organization. To achieve this goal, software engineering practices must be applied throughout the evolution of the software product. Most of these software engineering practices attempt to create and modify software in a manner that maximizes the probability of satisfying its user expectations. Other practices, addressed in this module, actually attempt to insure that the product will meet these user expectations. These practices are collectively referred to as *software verification* and *validation* (*V&V*). The reader is cautioned that terminology in this area is often confusing and conflict-

ing. The glossary of this module contains complete definitions of many of the terms often used to discuss V&V practices. This section attempts to clarify terminology as it will be used in the remainder of the module.

Validation refers to the process of evaluating software at the end of its development to insure that it is free from *failures* and complies with its requirements. A failure is defined as incorrect product behavior. Often this validation occurs through the utilization of various testing approaches. Other intermediate software products may also be validated, such as the validation of a requirements description through the utilization of a prototype.

Verification refers to the process of determining whether or not the products of a given phase of a software development process fulfill the requirements established during the previous phase. Software technical reviews represent one common approach for verifying various products. For example, a specifications review will normally attempt to verify the specifications description against a requirements description (what Rombach has called "D-requirements" and "C-requirements," respectively [Rombach87]). Proof of correctness is another technique for verifying programs to formal specifications. Verification approaches attempt to identify product *faults* or *errors*, which give rise to failures.

2. Evolving Nature of Area

As the complexity and diversity of software products continue to increase, the challenge to develop new and more effective V&V strategies continues. The V&V approaches that were reasonably effective on small batch-oriented products are not sufficient for concurrent, distributed, or embedded products. Thus, this area will continue to evolve as new research results emerge in response to new V&V challenges.

II. V&V Limitations

The overall objective of software V&V approaches is to insure that the product is free from failures and meets its user's expectations. There are several theoretical and practical limitations that make this objective impossible to obtain for many products.

1. Theoretical Foundations

Some of the initial theoretical foundations for testing were presented by Goodenough and Gerhart in their classic paper [Goodenough75]. This paper provides definitions for reliability and validity, in an attempt to characterize the properties of a test selection strategy. A mathematical framework for investigating testing that enables comparisons of the power of testing methods is described in [Gourlay83]. Howden claims the most important theoretical result in program testing and analysis is that no general purpose testing or analysis procedure can be used to prove program correctness. A proof of this result is contained in his text [Howden87].

2. Impracticality of Testing All Data

For most programs, it is impractical to attempt to test the program with all possible inputs, due to a combinatorial explosion [Beizer83, Howden87]. For those inputs selected, a testing oracle is needed to determine the correctness of the output for a particular test input [Howden87].

3. Impracticality of Testing All Paths

For most programs, it is impractical to attempt to test all execution paths through the product, due to a combinatorial explosion [Beizer83]. It is also not possible to develop an algorithm for generating test data for paths in an arbitrary product, due to the inability to determine path feasibility [Adrion86].

4. No Absolute Proof of Correctness

Howden claims that there is no such thing as an absolute proof of correctness [Howden87]. Instead, he suggests that there are proofs of equivalency, *i.e.*, proofs that one description of a product is equivalent to another description. Hence, unless a formal specification can be shown to be correct and, indeed, reflects exactly the user's expectations, no claim of product correctness can be made [Beizer83, Howden87].

III. The Role of V&V in Software Evolution

The evolution of a software product can proceed in many ways, depending upon the development approach used. The development approach determines the specific intermediate products to be created. For any given project, V&V objectives must be identified for each of the products created.

1. Types of Products

To simplify the discussion of V&V objectives, five types of products are considered in this module. These types are not meant to be a partitioning of all software documents and will not be rigorously defined. Within each product type, many different representational forms are possible. Each representational form determines, to a large extent, the applicability of particular V&V approaches. The intent here is not to identify V&V approaches applicable to all products in any form, but instead to describe V&V approaches for representative forms of products. References are provided to other sources that treat particular approaches in depth.

a. Requirements

The requirements document (Rombach [Rombach87]: "customer/user-oriented requirements" or C-requirements) provides an informal statement of the user's needs.

b. Specifications

The specifications document (Rombach: "design-oriented requirements" or D-requirements) provides a refinement of the user's needs, which must be satisfied by the product. There are many approaches for representing specifications, both formal and informal [Berztiss87, Rombach87].

c. Designs

The product design describes how the specifications will be satisfied. Depending upon the development approach applied in the project, there may be multiple levels of designs. Numerous possible design representation approaches are described in *Introduction to Software Design* [Budgen88].

d. Implementations

"Implementation" normally refers to the source code for the product. It can, however, refer to other implementation-level products, such as decision tables [Beizer83].

e. Changes

Changes describe modifications made to the product. Modifications are normally the result of error corrections or additions of new capabilities to the product.

2. V&V Objectives

The specific V&V objectives for each product must be determined on a project-by-project basis. This determination will be influenced by the criticality of the product, its constraints, and its complexity. In general, the objective of the V&V function is to insure that the product satisfies the user needs. Thus, everything in the product's requirements and specifications must be the target of some V&V activity. In order to limit the scope of this module, however, the V&V approaches described will concentrate on the functional and performance portions of the requirements and specifications for the product. Approaches for determining whether a product satisfies its requirements and specifications with respect to safety, portability, usability, maintainability, serviceability, security, etc., although very important for many systems, will not be addressed here. This is consistent with the V&V approaches normally described in the literature. The broader picture of "assurance of software quality" is addressed elsewhere [Brown87].

Limiting the scope of the V&V activities to functionality and performance, five general V&V objectives can be identified [Howden81, Powell86a]. These objectives provide a framework within which it is possible to determine the applicability of various V&V approaches and techniques.

a. Correctness

The extent to which the product is fault free.

b. Consistency

The extent to which the product is consistent within itself and with other products.

c. Necessity

The extent to which everything in the product is necessary.

d. Sufficiency

The extent to which the product is complete.

e. Performance

The extent to which the product satisfies its performance requirements.

IV. Software V&V Approaches and their Applicability

Software V&V activities occur throughout the evolution of the product. There are numerous techniques and tools that may be used in isolation or in combination with each other. In an effort to organize these V&V activities, five broad classifications of approaches are presented. These categories are not meant to provide a partitioning, since there are some techniques that span categories. Instead, the categories represent a practical view that reflects the way most of the V&V approaches are described in the literature and used in practice. Possible combinations of these approaches are discussed in the next section.

1. Software Technical Reviews

The software technical review process includes techniques such as walk-throughs, inspections, and audits. Most of these approaches involve a group meeting to assess a work product. A comprehensive examination of the technical review process and its effectiveness for software products is presented in *The Software Technical Review Process* [Collofello-88].

Software technical reviews can be used to examine all the products of the software evolution process. In particular, they are especially applicable and necessary for those products not yet in machine-processable form, such as requirements or specifications written in natural language.

2. Software Testing

Software testing is the process of exercising a product to verify that it satisfies specified requirements or to identify differences between expected and actual results [IEEE83a].

a. Levels of Testing

In this section, various levels of testing activities, each with its own specific goals, are identified and described. This listing of levels is not meant to be complete, but will illustrate the notion of levels of testing with particular goals. Other possible levels of testing not addressed here include acceptance testing, alpha testing, beta testing, etc. [Beizer84].

(i) Module Testing

Module (or unit) testing is the lowest level of testing and involves the testing of a software module or unit. The goal of module-level testing is to insure that the component being tested conforms to its specifications and is ready to be integrated with other components of the product. Module testing is treated in depth in the curriculum module *Unit Testing and Analysis* [Morell88].

(ii) Integration Testing

Integration testing consists of the systematic combination and execution of product components. Multiple levels of integration testing are possible with a combination of hardware and software components at several different levels. The goal of integration testing is to insure that the interfaces between the components are correct and that the product components combine to execute the product's functionality correctly.

(iii) System Testing

System testing is the process of testing the integrated hardware and software system to verify that the system meets its specified requirements [IEEE83a]. Practical priorities must be established to complete this task effectively. One general priority is that system testing must concentrate more on system capabilities rather than component capabilities [Beizer84, McCabe85, Petschenik85]. This suggests that system tests concentrate on insuring the use and interaction of functions rather than testing the details of their implementations. Another priority is that testing typical situations is more important that testing special cases [Petschenik85, Sum86]. This suggests that test cases be constructed corresponding to high-probability user scenarios. This facilitates early detection of critical problems that would greatly disrupt a user.

There are also several key principles to adhere to during system testing:

- System tests should be developed and performed by a group independent of the people who developed the code.
- System test plans must be developed and inspected with the same rigor as other elements of the project.
- System test progress must be planned and tracked similarly to other elements of the project.
- System tests must be repeatable.

(iv) Regression Testing

Regression testing can be defined as the process of executing previously defined test cases on a modified program to assure that the software changes have not adversely affected the program's previously existing functions. The error-prone nature of software modification demands that regression testing be performed. Some examples of the types of errors targeted by regression testing include:

- **Data corruption errors.** These errors are side effects due to shared data.
- **Inappropriate control sequencing errors.** These errors are side effects due to changes in execution sequences. An example of this type of error is the attempt to remove an item from a queue before it is placed into the queue.
- **Resource contention.** Examples of these types of errors are potential bottlenecks and deadlocks.
- **Performance deficiencies.** These include timing and storage utilization errors.

An important regression testing strategy is to place a higher priority on testing the older capabilities of the product than on testing the new capabilities provided by the modification [Petschenik85]. This insures that capabilities the user has become dependent upon are still intact. This is especially important when we consider that a recent study found that half of all failures detected by users after a modification were failures of old capabilities, as a result of side effects of implementation of new functionality [Collofello87].

Regression testing strategies are not well-defined in the literature. They differ from development tests in that development tests tend to be smaller and diagnostic in nature, whereas regression tests tend to be long and complex scenarios testing many capabilities, yet possibly proving unhelpful in isolating a problem, should one be encountered. Most regression testing strategies require that some baseline of product tests be rerun. These tests must be

supplemented with specific tests for the recent modifications. Strategies for testing modifications usually involve some sort of systematic execution of the modification and related areas. At a module level, this may involve retesting module execution paths traversing the modification. At a product level, this activity may involve retesting functions that execute the modified area [Fisher77]. The effectiveness of these strategies is highly dependent upon the utilization of test matrices (see below), which enable identification of coverage provided by particular test cases.

b. Testing Techniques and their Applicability

(i) Functional Testing and Analysis

Functional testing develops test data based upon documents specifying the behavior of the software. The goal of functional testing is to exercise each aspect of the software's specified behavior over some subset of its input. Howden has developed an integrated approach to testing based upon this notion of testing each aspect of specified behavior [Howden86, Howden87]. A classification of functional testing approaches and a description of representative techniques is presented in [Morell88].

Functional testing and analysis techniques are applicable for all levels of testing. However, the level of specified behavior to be tested will normally be at a higher level for integration and system-level testing. Thus, at a module level, it is appropriate to test boundary conditions and low-level functions, such as the correct production of a particular type of error message. At the integration and system level, the types of functions tested are normally those involving some combination of lower-level functions. Testing combinations of functions involves selection of specific sequences of inputs that may reveal sequencing errors due to:

- race conditions
- resource contention
- deadlock
- interrupts
- synchronization issues

Functional testing and analysis techniques are effective in detecting failures during all levels of testing. They must, however, be used in combination with other strategies to improve failure detection effectiveness [Beizer84, Girgis86, Howden80, Selby86].

The automation of functional testing techniques has been hampered by the informality of commonly used specification techniques. The difficulty lies in the identification of the functions to be tested. Some limited success in automating this process has been obtained for some more rigorous specification techniques. These results are described in [Morell88].

(ii) Structural Testing and Analysis

Structural testing develops test data based upon the implementation of the product. Usually this testing occurs on source code. However, it is possible to do structural testing on other representations of the program's logic. Structural testing and analysis techniques include data flow anomaly detection, data flow coverage assessment, and various levels of path coverage. A classification of structural testing approaches and a description of representative techniques is presented in [Morell88] and in Glenford Myers' text [Myers79].

Structural testing and analysis are applicable to module testing, integration testing, and regression testing. At the system test level, structural testing is normally not applicable, due to the size of the system to be tested. For example, a paper discussing the analysis of a product consisting of 1.8 million lines of code, suggests that over 250,000 test cases would be needed to satisfy coverage criteria [Petschenik85]. At the module level, all of the structural techniques are applicable. As the level of testing increases to the integration level, the focus of the structural techniques is on the area of interface analysis [Howden87]. This interface analysis may involve module interfaces, as well as interfaces to other system components. Structural testing and analysis can also be performed on designs using manual walk-throughs or design simulations [Powell86a].

Structural testing and analysis techniques are very effective in detecting failures during the module and integration testing levels. Beizer reports that path testing catches 50% of all errors during module testing and a total of one-third of all of the errors [Beizer84]. Structural testing is very cumbersome to perform without tools, and even with tools requires considerable effort to achieve desirable levels of coverage. Since structural testing and analysis techniques cannot detect missing functions (nor some other types of errors), they must be used in combination with other strategies to improve failure detection effectiveness [Beizer84, Girgis86, Howden80, Selby86].

There are numerous automated techniques to support structural testing and analysis. Most of the automated approaches provide statement and branch coverage. Tools for automating several structural testing techniques are described in the papers cited in [Morell88].

(iii) Error-Oriented Testing and Analysis

Error-oriented testing and analysis techniques are those that focus on the presence or absense of errors in the programming process. A classification of these approaches and a description of representative techniques is presented in [Morell88].

Error-oriented testing and analysis techniques are, in general, applicable to all levels of testing. Some techniques, such as statistical methods [Currit86], error seeding [Mills83], and mutation testing [DeMillo78], are particularly suited to application during the integration and system levels of testing.

(iv) Hybrid Approaches

Combinations of the functional, structural, and error-oriented techniques have been investigated and are described in [Morell88]. These hybrid approaches involve integration of techniques, rather than their composition. Hybrid approaches, particularly those involving structural testing, are normally applicable at the module level.

(v) Integration Strategies

Integration consists of the systematic combination and analysis of product components. It is assumed that the components being integrated have already been individually examined for correctness. This insures that the emphasis of the integration activity is on examining the interaction of the components [Beizer84, Howden87]. Although integration strategies are normally discussed for implementations, they are also applicable for integrating the components of any product, such as designs.

There are several types of errors targeted by integration testing:

- **Import/export range errors** This type of error occurs when the source of input parameters falls outside of the range of their destination. For example, assume module **A** calls module **B** with table pointer *X*. If **A** assumes a maximum table size of 10 and **B** assumes a maximum table size of 8, an import/export range error occurs. The detection of this type of error requires careful boundary-value testing of parameters.
- **Import/export type compatibility errors.** This type of error is attributed to a mismatch of user-defined types. These errors are normally detected by compilers or code inspections.
- **Import/export representation errors.** This type of error occurs when parameters are of the same type, but the meaning of the parameters is different in the calling and called modules. For example, assume module **A** passes a parameter *Elapsed_Time*, of type real, to module **B**. Module **A** might pass the value as seconds, while module **B** is assuming the value is passed as milliseconds. These types of errors are difficult to detect, although range checks and inspections provide some assistance.
- **Parameter utilization errors.** Dangerous assumptions are often made concerning whether a module called will alter the information passed to it. Although support for detecting such errors is provided by some compilers, careful testing and/or inspections may be necessary to insure that values have not been unexpectedly corrupted.
- **Integration time domain/ computation errors.** A domain error occurs when a specific input follows the wrong path due to an error in the control flow. A computation error exists when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed. Although domain and computation errors are normally addressed during module testing, the concepts apply across module boundaries. In fact, some domain and computation errors in the integrated program might be masked during integration testing if the module being integrated is assumed to be correct and is treated as a black box. Examples of these types of errors and an approach for detecting them is presented in [Haley84].

Measures of integration coverage can be defined in an analogous way to those defined by Miller [Miller77] for module-level coverage. Whereas Miller's "C1" measure requires every statement to be executed, an "I1" measure for integration coverage might require every module to be invoked during the integration test. The "C2" measure, which requires each branch to be executed, might have an integration coverage counterpart "I2" that requires each module to be invoked by all possible callers. An "I3" measure might require that each call in every module be executed.

In addition to module-level coverage, various interface coverage measures can be defined. An "X0" measure requires each I/O interface be utilized. This implies that passed parameters are referenced, as well as returned, parameters. An "X1" measure requires that each output variable of a module be set in all possible assignments and that each input variable be used at all possible reference points.

Several strategies for integration testing exist. These strategies may be used independently or in combination. The primary techniques are top-down, bottom-up, big-bang, and threaded integration, although terminology used in the literature varies. Top-down integration attempts to combine incrementally the components of the program, starting with the topmost element and simulating lower level elements with stubs. Each stub is then replaced with an actual program component as the integration process proceeds in a top-down fashion. Top-down integration is useful for those components of the program with complicated control structures [Beizer84]. It also provides visibility into the integration process by demonstrating a potentially useful product early.

Bottom-up integration attempts to combine incrementally components of the program starting with those components that do not invoke other components. Test drivers must be constructed to invoke these components. As bottom-up integration proceeds, test drivers are replaced with the actual program components that perform the invocation, and new test drivers are constructed until the "top" of the program is reached. Bottom-up integration is consistent with the notion of developing software as a series of building blocks. Bottom-up integration should proceed wherever the driving control structure is not too complicated [Beizer84].

Big-bang integration is not an incremental strategy and involves combining and testing all modules at once. Except for small programs, big-bang integration is not a cost-effective technique because of the difficulty of isolating integration testing failures [Beizer84].

Threaded integration is an incremental technique that identifies major processing functions that the product is to perform and maps these functions to modules implementing them. Each processing function is called a thread. A collection of related threads is often called a build. Builds may serve as a basis for test management. To test a thread, the group of modules corresponding to the thread is combined. For those modules in the thread with interfaces to other modules not supporting the thread, stubs are used. The initial threads to be tested normally correspond to the "backbone" or "skeleton" of the product under test. (These terms are also used to refer to this type of integration strategy.) The addition of new threads for the product undergoing integration proceeds incrementally in a planned fashion. The use of system verification diagrams for "threading" requirements is described in [Deutsch82].

(vi) Transaction Flow Analysis

Transaction flow analysis develops test data to execute sequences of tasks that correspond to a transaction, where a "transaction" is defined as a unit of work seen from a system user's point of view [Beizer84, McCabe85, Petschenik85]. An example of a transaction for an operating system might be a request to print a file. The execution of this transaction requires several tasks, such as checking the existence of the file, validating permission to read the file, etc.

The first step of transaction flow analysis is to identify the transactions. McCabe suggests the drawing of data flow diagrams after integration testing to model the logical flow of the system. Each transaction can then be identified as a path through the data flow diagram, with each data flow process corresponding to a task that must be tested in combination with other tasks on the transaction flow [McCabe85]. Information about transaction flows may also be obtained from HIPO diagrams, Petri nets, or other similar system-level documentation [Beizer84].

Once the transaction flows have been identified, black-box testing techniques can be utilized to generate test data for selected paths through the transaction flow diagram. Some possible guidelines for selecting paths follow:

- Test every link/decision in the transaction flow graph.
- Test each loop with a single, double, typical, maximum, and maximum-less-one number of iterations.
- Test combinations of paths within and between transaction flows.
- Test that the system does not do things that it is not supposed to do, by watching for unexpected sequences of paths within and between transaction flows.

Transaction flow analysis is a very effective technique for identifying errors corresponding to interface problems with functional tasks. It is most applicable to integration and system-

level testing. The technique is also appropriate for addressing completeness and correctness issues for requirements, specifications, and designs.

(vii) Stress Analysis

Stress analysis involves analyzing the behavior of the system when its resources are saturated, in order to assess whether or not the system will continue to satisfy its specifications. Some examples of errors targeted by stress tests include:

- potential race conditions
- errors in processing sequences
- errors in limits, thresholds, or controls designed to deal with overload situations
- resource contention and depletion

For example, one typical stress test for an operating system would be a program that requests as much memory as the system has available.

The first step in performing a stress analysis is identifying those resources that can and should be stressed. This identification is very system-dependent, but often includes resources such as file space, memory, I/O buffers, processing time, and interrupt handlers. Once these resources have been identified, test cases must be designed to stress them. These tests often require large amounts of data, for which automated support in the form of test-case generators is needed [Beizer84, Sum86].

Although stress analysis is often viewed as one of the last tasks to be performed during system testing, it is most effective if it is applied during each of the product's V&V activities. Many of the errors detected during a stress analysis correspond to serious design flaws. For example, a stress analysis of a design may involve an identification of potential bottlenecks that may prevent the product from satisfying its specifications under extreme loads [Beizer84].

Stress analysis is a necessary complement to the previously described testing and analysis techniques for resource-critical applications. Whereas the foregoing techniques primarily view the product under normal operating conditions, stress analysis views the product under conditions that may not have been anticipated. Stress analysis techniques can also be combined with other approaches during V&V activities to insure that the product's specifications for such attributes as performance, safety, security, etc., are met.

(viii) Failure Analysis

Failure analysis is the examination of the product's reaction to failures of hardware or software. The product's specifications must be examined to determine precisely which types of failures must be analyzed and what the product's reaction must be. Failure analysis is sometimes referred to as "recovery testing" [Beizer84].

Failure analysis must be performed during each of the product's V&V activities. It is essential during requirement and specification V&V activities that a clear statement of the product's response to various types of failures be addressed in terms that allow analysis. The design must also be analyzed to show that the product's reaction to failures satisfies its specifications. The failure analysis of implementations often occurs during system testing. This testing may take the form of simulating hardware or software errors or actual introduction of these types of errors.

Failure analysis is essential to detecting product recovery errors. These errors can lead to lost files, lost data, duplicate transactions, etc. Failure analysis techniques can also be combined with other approaches during V&V activities to insure that the product's specifications for such attributes as performance, security, safety, usability, etc., are met.

(ix) Concurrency Analysis

Concurrency analysis examines the interaction of tasks being executed simultaneously within the product to insure that the overall specifications are being met. Concurrent tasks may be executed in parallel or have their execution interleaved. Concurrency analysis is sometimes referred to as "background testing" [Beizer84].

For products with tasks that may execute in parallel, concurrency analysis must be performed during each of the product's V&V activities. During design, concurrency analysis should be performed to identify such issues as potential contention for resources, deadlock, and priorities. A concurrency analysis for implementations normally takes place during system testing. Tests must be designed, executed, and analyzed to exploit the parallelism in the system and insure that the specifications are met.

(x) Performance Analysis

The goal of performance analysis is to insure that the product meets its specified perfor-

mance objectives. These objectives must be stated in measurable terms, so far as possible. Typical performance objectives relate to response time and system throughput [Beizer84].

A performance analysis should be applied during each of the product's V&V activities. During requirement and specification V&V activities, performance objectives must be analyzed to insure completeness, feasibility, and testability. Prototyping, simulation, or other modeling approaches may be used to insure feasibility. For designs, the performance requirements must be allocated to individual components. These components can then be analyzed to determine if the performance requirements can be met. Prototyping, simulation, and other modeling approaches again are techniques applicable to this task. For implementations, a performance analysis can take place during each level of testing. Test data must be carefully constructed to correspond to the scenarios for which the performance requirements were specified.

3. Proof of Correctness

Proof of correctness is a collection of techniques that apply the formality and rigor of mathematics to the task of proving the consistency between an algorithmic solution and a rigorous, complete specification of the intent of the solution [Adrion86, Powell86b]. This technique is also often referred to as "formal verification." The usual proof technique follows Floyd's Method of Inductive Assertions or some variant [Floyd67, Hantler76].

Proof of correctness techniques are normally presented in the context of verifying an implementation against a specification. The techniques are also applicable in verifying the correctness of other products, as long as they possess a formal representation [Ambler78, Korelsky87].

There are several limitations to proof of correctness techniques. One limitation is the dependence of the technique upon a correct formal specification that reflects the user's needs. Current specification approaches cannot always capture these needs in a formal way, especially when product aspects such as performance, reliability, quality, etc., are considered [Berztiss87, Rombach87]. Another limitation has to do with the complexity of rigorously specifying the execution behavior of the computing environment. For large programs, the amount of detail to handle, combined with the lack of powerful tools may make the proof technique impractical [Beizer83, Korelsky-87, Howden87, Powell86b].

More information on proof of correctness approaches is contained in the curriculum module *Formal Verification of Programs* [Berztiss88].

4. Simulation and Prototyping

Simulation and prototyping are techniques for analyzing the expected behavior of a product. There are many approaches to constructing simulations and prototypes that are well-documented in the literature. For V&V purposes, simulations and prototypes are normally used to analyze requirements and specifications to insure that they reflect the user's needs [Brackett88]. Since they are executable, they offer additional insight into the completeness and correctness of these documents. Simulations and prototypes can also be used to analyze predicted product performance, especially for candidate product designs, to insure that they conform to the requirements. It is important to note that the utilization of simulation and prototyping as V&V techniques requires that the simulations and prototypes themselves be correct. Thus, the utilization of these techniques requires an additional level of V&V activity.

5. Requirements Tracing

Requirements tracing is a technique for insuring that the product, as well as the testing of the product, addresses each of its requirements. The usual approach to performing requirements tracing uses matrices. One type of matrix maps requirements to software modules. Construction and analysis of this matrix can help insure that all requirements are properly addressed by the product and that the product does not have any superfluous capabilities [Powell86b]. System Verification Diagrams are another way of analyzing requirements/modules traceability [Deutsch82]. Another type of matrix maps requirements to test cases. Construction and analysis of this matrix can help insure that all requirements are properly tested. A third type of matrix maps requirements to their evaluation approach. The evaluation approaches may consist of various levels of testing, reviews, simulations, etc. The requirements/evaluation matrix insures that all requirements will undergo some form of V&V [Deutsch82, Powell86b]. Requirements tracing can be applied for all of the products of the software evolution process.

V. Software V&V Planning

The development of a comprehensive V&V plan is essential to the success of a project. This plan must be developed early in the project. Depending on the development approach followed, multiple levels of test plans may be developed, corresponding to various levels of V&V activities. Guidelines for the contents of system, software, build, and module test plans have been documented in the literature [Deutsch82, DoD87, Evans84, NBS76, IEEE83b]. These references also contain suggestions about how to document other information, such as test procedures and test cases. The formulation of an effective V&V plan requires many

considerations that are defined in the remainder of this section.

1. Identification of V&V Goals

V&V goals must be identified from the requirements and specifications. These goals must address those attributes of the product that correspond to its user expectations. These goals must be achievable, taking into account both theoretical and practical limitations [Evans84, Powell86a, Sum86].

2. Selection of V&V Techniques

Once a set of V&V objectives has been identified, specific techniques must be selected for each of the project's evolving products. A methodology for the selection of techniques and tools is presented in [Powell86b]. More specific guidelines for the selection of techniques applicable at the unit level of testing are presented in [Morell88]. A mapping of some of the approaches presented in Section IV of this module to the products in Section III follows.

a. Requirements

The applicable techniques for accomplishing the V&V objectives for requirements are technical reviews, prototyping, and simulation. The review process is often called a System Requirements Review (SRR). Depending upon the representation of the requirements, consistency analyzers may be used to support the SRR.

b. Specifications

The applicable techniques for accomplishing the V&V objectives for specifications are technical reviews, requirements tracing, prototyping, and simulation. The specifications review is sometimes combined with a review of the product's high-level design. The requirements must be traced to the specifications.

c. Designs

The applicable techniques for accomplishing the V&V objectives for designs are technical reviews, requirements tracing, prototyping, simulation, and proof of correctness. High-level designs that correspond to an architectural view of the product are often reviewed in a Preliminary Design Review. Detailed designs are addressed by a Critical Design Review. Depending upon the representation of the design, static analyzers may be used to assist these review processes. Requirements must be traced to modules in the architectural design; matrices can be used to facilitate this process [Powell86b]. Prototyping and simulation can be used to assess feasibility and adherence to performance requirements. Proofs of correctness, where applicable, are normally performed at the detailed design level [Dyer87].

d. Implementations

The applicable techniques for accomplishing the V&V objectives for implementations are technical reviews, requirements tracing, testing, and proof of correctness. Various code review techniques such as walk-throughs and inspections exist. At the source-code level, several static analysis techniques are available for detecting implementation errors. The requirements tracing activity is here concerned with tracing requirements to source-code modules. The bulk of the V&V activity for source code consists of testing. Multiple levels of testing are usually performed. Where applicable, proof-of-correctness techniques may be applied, usually at the module level.

e. Changes

Since changes describe modifications to products, the same techniques used for V&V during development may be applied during modification. Changes to implementations require regression testing.

3. Organizational Responsibilities

The organizational structure of a project is a key planning consideration for project managers. An important aspect of this structure is delegation of V&V activities to various organizations [Deutsch82, Evans84, Petschenik85, Sum86]. This decision is often based upon the size, complexity, and criticality of the product. In this module, four types of organizations are addressed. These organizations reflect typical strategies for partitioning tasks to achieve V&V goals for the product. It is, of course, possible to delegate these V&V activities in many other ways.

a. Development Organization

The development organization has responsibility for participating in technical reviews for all of the evolution products. These reviews must insure that the requirements can be traced throughout the class of products. The development organization may also construct prototypes and simulations. For code, the development organization has responsibility for preparing and executing test plans for unit and integration levels of testing. In some environments, this is referred to as Preliminary Qualification Testing. The development organization also constructs any applicable proofs of correctness at the module level.

b. Independent Test Organization

An independent test organization (ITO) may be established, due to the magnitude of the testing effort or the need for objectivity. An ITO enables the preparation for test activities to occur in paral-

lel with those of development. The ITO normally participates in all of the product's technical reviews and monitors the preliminary qualification testing effort. The primary responsibility of the ITO is the preparation and execution of the product's system test plan. This is sometimes referred to as the Formal Qualification Test. The plan for this must contain the equivalent of a requirements/evaluation matrix that defines the V&V approach to be applied for each requirement [Deutsch82]. If the product must be integrated with other products, this integration activity is normally the responsibility of the ITO as well.

c. Software Quality Assurance

Although software quality assurance may exist as a separate organization, the intent here is to identify some activities for assuring software quality that may be distributed using any of a number of organizational structures [Brown87]. Evaluations are the primary avenue for assuring software quality. Some typical types of evaluations to be performed where appropriate throughout the product life cycle are identified below. Other types can be found in *Assurance of Software Quality* [Brown87]. Evaluation types:

- internal consistency of product
- understandability of product
- traceability to indicated documents
- consistency with indicated documents
- appropriate allocation of sizing, timing resources
- adequate test coverage of requirements
- consistency between data definitions and use
- adequacy of test cases and test procedures
- completeness of testing
- completeness of regression testing

d. Independent V&V Contractor

An independent V&V contractor may sometimes be used to insure independent objectivity and evaluation for the customer. The scope of activities for this contractor varies, including any or all of the activities addressed for the Independent Test and Software Quality Assurance organizations [Deutsch82].

4. Integrating V&V Approaches

Once a set of V&V objectives has been identified, an overall integrated V&V approach must be determined. This approach involves integration of techniques applicable to the various life cycle phases as well as delegation of these tasks among the project's organizations. The planning of this integrated V&V approach is very dependent upon the nature of the product and the process used to develop it. Traditional integrated V&V approaches have followed the "waterfall model" with various V&V functions allocated to the project's development phases [Deutsch82, DoD87, Evans84, Powell86a]. Alternatives to this approach exist, such as the Cleanroom software development process developed by IBM. This approach is based on a software development process that produces incremental product releases, each of which undergoes a combination of formal verification and statistical testing techniques [Currit86, Dyer87]. Regardless of the approach selected, V&V progress must be tracked. Requirements/evaluation matrices play a key role in this tracking by providing a means of insuring that each requirement of the product is addressed [Powell86b, Sum86].

5. Problem Tracking

Other critical aspects of a software V&V plan are developing a mechanism for documenting problems encountered during the V&V effort, routing problems identified to appropriate individuals for correction, and insuring that the corrections have been performed satisfactorily. Typical information to be collected includes:

- when the problem occurred
- where the problem occurred
- state of the system before occurrence
- evidence of the problem
- actions or inputs that appear to have led to occurrence
- description of how the system should work; reference to relevant requirements
- priority for solving problem
- technical contact for additional information

Problem tracking is an aspect of configuration management that is addressed in detail in the curriculum module *Software Configuration Management* [Tomayko87]. A practical application of problem tracking for operating system testing is presented in [Sum86].

6. Tracking Test Activities

The software V&V plans must provide a mechanism for tracking the testing effort. Data must be collected that enable project management to assess both the quality and the cost of testing activities. Typical data to collect include:

- number of tests executed
- number of tests remaining
- time used
- resources used

- number of problems found and the time spent finding them

These data can then be used to track actual test progress against scheduled progress. The tracking information is also important for future test scheduling.

7. Assessment

It is important that the software V&V plan provide for the ability to collect data that can be used to assess both the product and the techniques used to develop it. Often this involves careful collection of error and failure data, as well as analysis and classification of these data. More information on assessment approaches and the data needed to perform them is contained in [Brown87].

Glossary

acceptance testing
The testing done to enable a customer to determine whether or not to accept a system [IEEE83a].

correctness
The extent to which software is free from faults. The extent to which software meets user expectations [IEEE83a].

coverage
Used in conjunction with a software feature or characteristic, the degree to which that feature or characteristic is tested or analyzed. Examples include input domain coverage, statement coverage, branch coverage, and path coverage [Morell88].

error
Human action that results in software containing a fault [IEEE83a].

failure
Incorrect behavior of a program induced by a fault [Howden87].

fault
An accidental condition that causes a product to fail. [IEEE83a].

integration testing
An orderly progression of testing in which software elements, hardware elements, or both are combined and tested until the entire system has been integrated [IEEE83a].

proof of correctness
A formal technique to prove mathematically that a program satisfies its specifications [IEEE83a].

regression testing
Selective retesting to detect faults introduced during modification of a system or system component. Retesting to verify that modifications have not caused unintended adverse effects and that the modified system or system component still meets its specified requirements [IEEE83a].

software technical review process
A critical evaluation of an object. Walkthroughs, inspections and audits can be viewed as forms of technical reviews [Collofello88].

system testing
The process of testing an integrated hardware and software system to verify that the system meets its specified requirements [IEEE83a].

testing
The process of exercising a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results [IEEE83a].

unit
Code that is meaningful to treat as a whole. It may be as small as a single statement or as large as a set of coupled subroutines [Morell88].

unit testing
The testing of a software unit.

validation
The process of evaluating software at the end of its development process to ensure compliance with its requirements [IEEE83a]. Other products besides code can be validated, such as requirements and designs, through the use of prototypes or simulation [Howden87].

verification
The process of determining whether or not the products of a given phase of a software development process fulfill the requirements established during the previous phase [IEEE83a]. Of-

ten equated with proof of correctness, which proves equivalency of programs to formal specifications [Howden87].

Teaching Considerations

Suggested Schedules

The nature of this module lends itself to several possible uses, depending upon the topics of interest and the depth of coverage desired.

Semester Course. This module can provide a framework for developing a graduate one-semester course on software verification and validation. The outline of the module can be used to structure the syllabus for the course. The amount of time to be spent on each topic will, of course, depend upon the background and interests of the students and instructor. It is recommended, however, that each V&V approach in the outline be addressed in the course, to insure that the students have sufficient breadth to understand the context for applying each approach and for combining approaches where appropriate.

Overview Lectures. This module can also be used as a basis for an overview of the software V&V area. Such an overview could be presented in 2 to 3 one-hour lectures. The overview could serve as the first week's lecture material for a course developed from one of the more advanced curriculum modules in this area, such as *Unit Testing and Analysis*. This overview would also be valuable for management in an industrial environment.

Exercises and Worked Examples

Any course on software V&V requires worked examples and exercises illustrating the techniques being taught. Some suggestions can be found in *The Software Technical Review Process* [Collofello88], *Unit Testing and Analysis* [Morell88], and *Formal Verification of Programs* [Berztiss88].

A useful exercise for students taking an introductory V&V course based on this module is the development of an integrated V&V plan for a project. The instructor can assign teams of students to projects. Projects can address different application areas, possibly using different development approaches. For example, one team might address V&V issues for an expert system; another team might be concerned with a robotics project developed with incremental product releases. Presentations at the end of the course can further enhance the learning experience for all involved.

Bibliography

Adrion86

Adrion, W. R., M. A. Branstad, and J. C. Cheriavsky. "Validation, Verification and Testing of Computer Software." In *Software Validation, Verification, Testing, and Documentation*, S. J. Andriole, ed. Princeton, N. J.: Petrocelli, 1986, 81-123.

This survey describes and categorizes V&V techniques applicable throughout the product's development cycle. Various testing techniques, as well as technical reviews and proof of correctness approaches, are addressed.

Ambler78

Ambler, A. L., *et al*. "Gypsy: A Language for Specification and Implementation of Verifiable Programs." *Proc. ACM Conf. on Language Design for Reliable Software.* New York: ACM, 1978, 1-10.

Beizer83

Beizer, B. *Software Testing Techniques*. New York: Van Nostrand, 1983.

This text provides comprehensive coverage of several testing techniques, with an emphasis on structural approaches. Information is also presented on database-driven test design and state-based testing. The text collects practical approaches and demonstrates them well with examples.

Beizer84

Beizer, B. *Software System Testing and Quality Assurance*. New York: Van Nostrand, 1984.

This text begins with an introduction to general testing techniques and then proceeds to integration and system testing approaches. Techniques for designing security, recovery, configuration, stress, and performance tests are presented in detail.

Berztiss87

Berztiss, A. *Formal Specification of Software.* Curriculum Module SEI-CM-8-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

Berztiss88

Berztiss, A., M. A. Ardis. *Formal Verification of Programs.* Curriculum Module SEI-CM-20-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

Brackett88

Brackett, J. W. *Software Requirements.* Curriculum Module SEI-CM-19-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

Brown87

Brown, B. J. *Assurance of Software Quality.* Curriculum Module SEI-CM-7-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., July 1987.

Budgen88

Budgen, D. *Introduction to Software Design.* Curriculum Module SEI-CM-2-2.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Nov. 1988.

Collofello87

Collofello, J. S. and J. Buck. "The Need for Software Quality Assurance during the Maintenance Phase." *IEEE Software 4*, 5 (Sept. 1987), 46-51.

This paper describes the need for thorough regression testing during software maintenance. Results of an extensive analysis of failures detected in a new release of a large system are presented. The results suggest strongly that existing features must undergo careful regression testing, since almost half of all failures in the new release occurred on existing features of the system that worked fine before the modification.

Collofello88

Collofello, J. S. *The Software Technical Review Process.* Curriculum Module SEI-CM-3-1.5, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., June 1988.

Currit86

Currit, P. A., M. Dyer, and H. D. Mills. "Certifying the Reliability of Software." *IEEE Trans. Software Eng. SE-12*, 1 (Jan. 1986), 3-11.

This paper describes a statistical approach to reliability projection. It presents a procedure for estimating the mean-time-to-failure for software systems. The approach is based on selecting test cases that reflect statistical samples of user operations. This paper provides another perspective on testing that should be addressed after the students have studied both structural and functional testing

methods. The paper also is targeted to an incremental development methodology that provides a nice contrast to the typical waterfall testing approaches.

DeMillo78

DeMillo, R. A., R. J. Lipton and F. G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." *Computer 11* (April 1978).

This paper is also included in [Miller81]. It introduces the idea of mutation testing. It can be used as an example of error-oriented testing and analysis. Significant other work in the mutation analysis area at a more advanced level can be found in other papers by DeMillo.

Deutsch82

Deutsch, M. S. *Software Verification and Validation: Realistic Project Approaches*. Englewood Cliffs, N. J.: Prentice-Hall, 1982.

This text presents an integrated approach to software V&V. A systematic approach to software integration utilizing threads is also presented in detail. Example outlines for test plans at each level of testing are also provided. The book also contains chapters addressing organizational issues such as the role of configuration management, software quality assurance, and independent test organizations.

DoD87

DoD. *Military Standard for Defense System Software Development.* DOD-STD-2167A, U. S. Department of Defense, 1987.

Included in this standard are descriptions of the V&V requirements that must be followed for defense system software development. This standard follows a waterfall model and can thus provide a framework for integrating V&V approaches.

Dyer87

Dyer, M. "A Formal Approach to Software Error Removal." *J. Syst. and Software 7*, 2 (June 1987), 109-114.

This paper describes the Cleanroom software development process developed by the IBM Federal Systems Division. This process replaces traditional testing models with a new process that combines program verification techniques with statistical testing. Data are presented that indicate that the method may be more effective than structural unit testing.

Evans84

Evans, M. W. *Productive Software Test Management*. New York: John Wiley, 1984.

This text is written for managers to provide them with a framework for managing software testing. Test planning is emphasized and described for the software development process. Other managerial issues, such as how to motivate the work force, are also presented. The text concludes with a fictional account of what can go wrong on a project if the test planning is poorly done.

Fisher77

Fisher, K. F. "A Test Case Selection Method for the Validation of Software Maintenance Modifications." *Proc. IEEE COMPSAC.* Long Beach, Calif.: IEEE Computer Society Press, 1977.

This paper describes a strategy for performing regression testing of a software component. The strategy is based upon retesting execution paths through the changed area of the component. Other types of possible regression testing strategies are also described.

Floyd67

Floyd, R. W. "Assigning Meanings to Programs." *Proc. Symp Applied Math.* Providence, R. I.: American Math. Society, 1967, 19-32.

A classic paper in the program verification area that is heavily referenced.

Girgis86

Girgis, M. R. and M. R. Woodward. "An Experimental Comparison of the Error Exposing Ability of Program Testing Criteria." *Proc. Workshop on Software Testing.* Washington, D. C.: IEEE Computer Society Press, 1986, 64-73.

This paper describes an experiment in which weak mutation testing, data flow testing, and control flow testing were compared in terms of their failure detection ability for FORTRAN programs. The paper can be used to reinforce the idea that testing strategies should be thought of as complementary rather than competing methods.

Goodenough75

Goodenough, J. B. and S. L. Gerhart. "Toward a Theory of Test Data Selection." *IEEE Trans. Software Eng. SE-1*, 2 (June 1975), 156-173.

This paper is also included in [Miller81]. The paper examines the theoretical and practical role of testing in software development. Definitions for reliability and validity are presented, in an attempt to characterize the properties of a test selection strategy.

Gourlay83
Gourlay, J. S. "A Mathematical Framework for the Investigation of Testing." *IEEE Trans. Software Eng. SE-9*, 6 (Nov. 1983), 686-709.

This paper develops a mathematical framework for testing that enables comparisons of the power of testing methods. This paper should be read after the Goodenough and Gerhart paper, since it attempts to build upon and clarify some of their results.

Haley84
Haley, A. and S. Zweben. "Development and Application of a White Box Approach to Integration Testing." *J. Syst. and Software 4*, 4 (Nov. 1984), 309-315.

This paper describes how the concept of domain and computation errors can be applied during integration testing. Some examples of these types of errors and approaches for detecting them are presented.

Hantler76
Hantler S. L. and J. C. King. "An Introduction to Proving the Correctness of Programs." *ACM Computing Surveys 8*, 3 (Sept. 1976), 331-53.

This paper is also included in [Miller81]. The paper presents, in an introductory fashion, a technique for showing the correctness of a program. The paper also helps define the relationship between proofs of correctness and symbolic execution.

Howden80
Howden, W. E. "Applicability of Software Validation Techniques to Scientific Programs." *ACM Trans. Prog. Lang. and Syst. 2*, 3 (July 1980), 307-320.

This paper is also included in [Miller81]. It describes an analysis of a collection of programs whose faults were known, in order to identify which testing techniques would have detected those faults. The paper provides a good example of error analysis, as well as motivation for integrated V&V approaches.

Howden81
Howden, W. E. "A Survey of Static Analysis Methods." In *Tutorial: Software Testing and Validation Techniques*, E. Miller and W. E. Howden, eds. Los Alamitos, Calif.: IEEE Computer Society Press, 1981, 101-115.

This survey describes a variety of static analysis techniques applicable to requirements, designs, and code. Formal and informal analysis techniques are presented. Symbolic execution is also introduced, along with some examples.

Howden86
Howden, W. E. "A Functional Approach to Program Testing and Analysis." *IEEE Trans. Software Eng. SE-12*, 10 (Oct. 1986), 997-1005.

This paper provides a summary of some of the major research contributions contained in Howden's text [Howden87]. An integrated approach to testing combining both static and dynamic analysis methods is introduced, along with a theoretical foundation for proving both its effectiveness and efficiency. This paper should be read after both static and dynamic analysis test methods have been studied.

Howden87
Howden, W. E. *Functional Program Testing and Analysis*. New York: McGraw-Hill, 1987.

This text expands upon the author's paper [Howden86], providing details and proofs of his integrated approach to testing.

IEEE83a
IEEE. *Standard Glossary of Software Engineering Terminology.* ANSI/IEEEStd729-1983, Institute of Electrical and Electronics Engineers, 1983.

IEEE83b
IEEE. *Standard for Software Test Documentation.* ANSI/IEEEStd829-1983, Institute of Electrical and Electronics Engineers, 1983.

Korelsky87
Korelsky, T., M. Shoji, R. A. Platek and C. Shilepsky. *Verification Methodology Evaluation.* RADC-TR-86-239, Rome Air Development Center, 1987.

McCabe85
McCabe, T. J. and G. G. Schulmeyer. "System Testing Aided by Structured Analysis: A Practical Experience." *IEEE Trans. Software Eng. SE-11*, 9 (Sept. 1985), 917-921.

This paper extends the ideas of Structured Analysis to system acceptance testing. Data flow diagrams are used to form the basis for integrating modules together to form transaction flows. A test traceability matrix is then defined, which maps test cases to transaction flows and their corresponding functionality.

Miller77
Miller, E. F., Jr. "Program Testing: Art Meets Theory." *Computer 10*, 7 (July 1977), 42-51.

This paper describes some of the history of software testing and how the field continues to evolve. Existing test methodologies are briefly addressed. A hierarchy of testing measures is also presented.

Miller81
E. Miller and W. E. Howden, eds. *Tutorial: Software Testing and Validation Techniques, 2nd Ed*. Los Alamitos, Calif.: IEEE Computer Society Press, 1981.

Mills83
Mills, H. D. *Software Productivity*. Boston: Little, Brown, 1983.

Morell88
Morell, L. J. *Unit Testing and Analysis.* Curriculum Module SEI-CM-9-1.1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Dec. 1988.

Myers79
Myers, G. J. *The Art of Software Testing*. New York: John Wiley, 1979.

A well written text that carefully explains practical approaches to testing modules utilizing functional and structural techniques.

NBS76
NBS. *Guidelines for Documentation of Computer Programs and Automated Data Systems.* Federal Information Processing Standards Publication FIPS PUB 38, National Bureau of Standards, Feb. 1976.

Petschenik85
Petschenik, N. H. "Practical Priorities in System Testing." *IEEE Software 2*, 5 (Sept. 1985), 18-23.

This paper describes the system testing priorities followed on a large Bell Communications Research product. The product continuously undergoes new releases. The system testing methodology is based on a set of priority rules. The first rule suggests that testing the system's capabilities is more important than testing its components. The second rule states that testing old capabilities is more important than testing new capabilities. The third rule implies that testing typical situations is more important than testing boundary value cases. The rationale for these rules is developed in the paper, along with discussion of practical experiences.

Powell86a
Powell, P. B. "Planning for Software Validation, Verification, and Testing." In *Software Validation, Verification, Testing and Documentation*, S. J. Andriole, ed. Princeton, N. J.: Petrocelli, 1986, 3-77. Also available from National Bureau of Standards, as NBS Publication 500-98, Nov. 1982.

This paper begins with an overview of a waterfall model for software development. Within the model, V&V activities are identified and described for each development phase. A framework for integrating V&V techniques is then presented. The paper concludes with an in-depth analysis of V&V planning issues, including several example plans for various levels of V&V technology.

Powell86b
Powell, P. B. "Software Validation, Verification and Testing Technique and Tool Reference Guide." In *Software Validation, Verification, Testing, and Documentation*, S. J. Andriole, ed. Princeton, N. J.: Petrocelli, 1986, 189-310.

This paper describes thirty techniques and tools for performing V&V activities. Each description includes the basic features of the technique or tool, an example, an assessment of its applicability, and the time required to learn it. Program verification is described in detail, along with various review and testing approaches. Requirements tracing is also presented as an important V&V technique.

Rombach87
Rombach, H. D. *Software Specification: A Framework.* Curriculum Module SEI-CM-11-1.0, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., Oct. 1987.

Selby86
Selby, R. W. "Combining Software Testing Strategies: An Empirical Evaluation." *Proc. Workshop on Software Testing.* Washington, D. C.: IEEE Computer Society Press, 1986, 82-90.

This paper presents the results of a study comparing code reading, functional testing, and structural testing methods, along with their six pairwise combinations. The results suggest that combined approaches are more effective than individual approaches. This paper should be read by students after the individual testing methods have been studied, to reinforce the idea that various combinations of techniques must be employed to detect faults.

Sum86
Sum, R. N., R. H. Campbell, and W. J. Kubitz. "An Approach to Operating System Testing." *J. Syst. and Software 6*, 3 (Aug. 1986), 273-284.

This paper describes a practical approach that was used to perform the system testing of an operating system. A framework for systematic testing is described that can be generalized to other types of systems. Data are also presented suggesting the effectiveness of the approach. Sample test definition forms, test matrices, and problem tracking memos are included in the appendix.

Tomayko87
Tomayko, J. E. *Software Configuration Management.* Curriculum Module SEI-CM-4-1.3, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., July 1987.