GUIDELINES FOR VERIFYING AND VALIDATING SOFTWARE REQUIREMENTS AND DESIGN SPECIFICATIONS

Barry W. Boehm

TRW
Redondo Beach, CA, USA

This paper presents the following guideline information on verification and validation (V&V) of software requirements and design specifications:

- Definitions of the terms "verification" and "validation," and an explanation of their context in the software life-cycle;
- A description of the basic sequence of functions performed during software requirements and design V&V
- An explanation, with examples: of the major software requirements and design V&V criteria: completeness, consistency, feasibility, and testability;
- An evaluation of the relative cost and effectiveness of the major software requirements and design V&V techniques with respect to the above criteria;
- An example V&V checklist for software system reliability and availability.

Based on the above, the paper provides recommendations of the combination of software requirements and design V&V techniques most suitable for small, medium, and large software specifications.

I. OBJECTIVES

The basic objectives in verification and validation (V&V) of software requirements and design specifications are to identify and resolve software problems and high-risk issues early in the software life-cycle. The main reason for doing this is indicated in Figure 1, (1). It shows that savings of up to 100:1 are possible by finding and fixing problems early rather than late in the life-cycle. Besides the major cost savings, there are also significant payoffs in improved reliability, maintainability, and human engineering of the resulting software product.

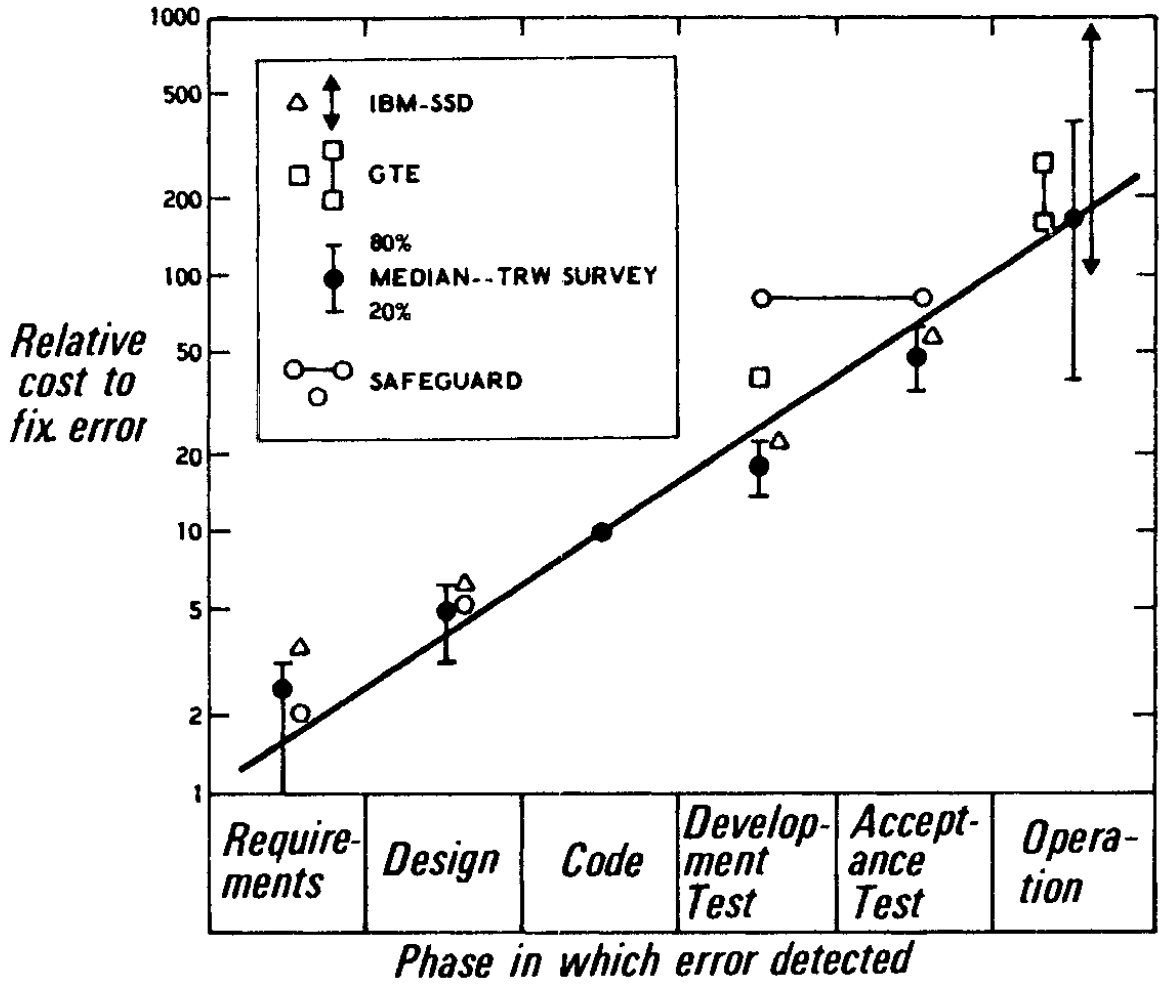


Figure 1: Payoff of Early Software V&V

II. SOFTWARE REQUIREMENTS AND DESIGN V&V FUNCTIONS

There is very little agreement in the software field on the definitions of the terms "verification" and "validation." Therefore, this paper begins by defining these terms rather carefully, and by explaining how each one fits into the software life-cycle. It then discusses the basic V&V criteria for software requirements and design specifications: completeness, consistency, feasibility, and testability.

A. Definitions

Verification - to establish the truth of the correspondence between a software product and its specification. (Note: This definition is derived from the Latin word for "truth," veritas. Note also that data bases and documentation are fit subjects for verification, as well as programs.)

Validation - to establish the fitness or worth of a software product for its operational mission. (Note: This definition is derived from the Latin word for "to be worthy," valere.)

Informally, we might define these terms via the following questions:

Verification: "Am I building the product right?"

Validation: "Am I building the right product?"

B. Context of V&V in the Software Life-Cycle

Figure 2 presents a "V-chart" which shows the context of verification and validation activities throughout the software life-cycle (2). This is done in terms of the levels of refinement involved in defining and developing a software product, and the corresponding levels of V&V involved in qualifying the product for operational use.

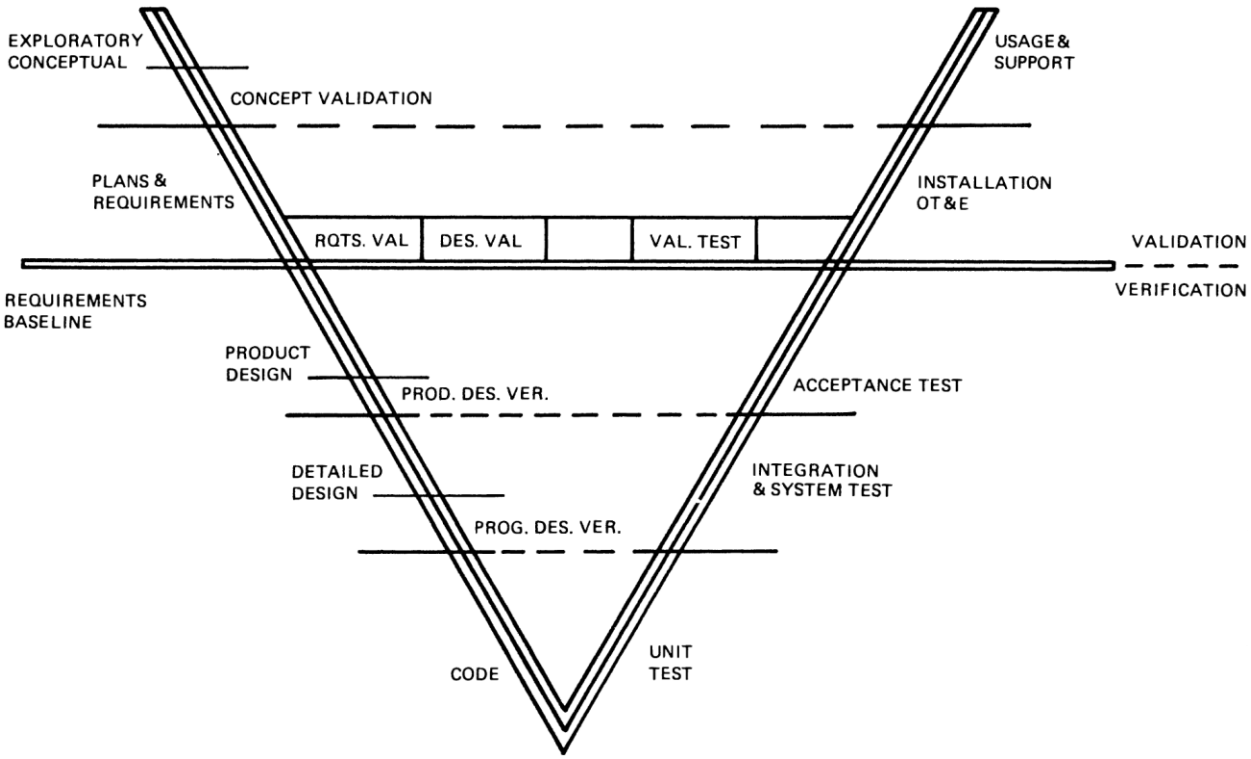


Figure 2: V&V Through the Software Life-Cycle

From the V-chart, it is clear that the key artifact that distinguishes verification activities from validation activities is the software requirements baseline. This refers to the requirements specification which is developed and validated during the Plans and Requirements Phase, accepted by the customer and developer at the Plans and Requirements Review as the basis for the software development contract, and formally change-controlled thereafter.

By definition, verification involves the comparison between the requirements baseline and the successive refinements descending from it—the product design, detailed design, code, database, and documentation--in order to keep these refinements consistent with the requirements baseline. Thus, verification activities begin in the Product Design phase and conclude with the acceptance Test. They do not lead to changes in the requirements baseline; only to changes in the refinements descending from it.

On the other hand, validation identifies problems which must be resolved by a change of the requirements specification. Thus, there are validation activities which occur throughout the software life-cycle, including the development phase. For example, a simulation of the product design may establish not only that the design cannot meet the baseline performance requirements (verification), but also that the performance requirements are too stringent for any cost-effective product designs, and therefore need to be changed (validation).

C. V&V Functions in the Early Phases

The basic sequence of functions performed during the Plans and Requirements phase and the Product design phase by the V&V agent, the specification agent (the analyst or software system engineer), the project manager, and the customer are shown in Figure 3.

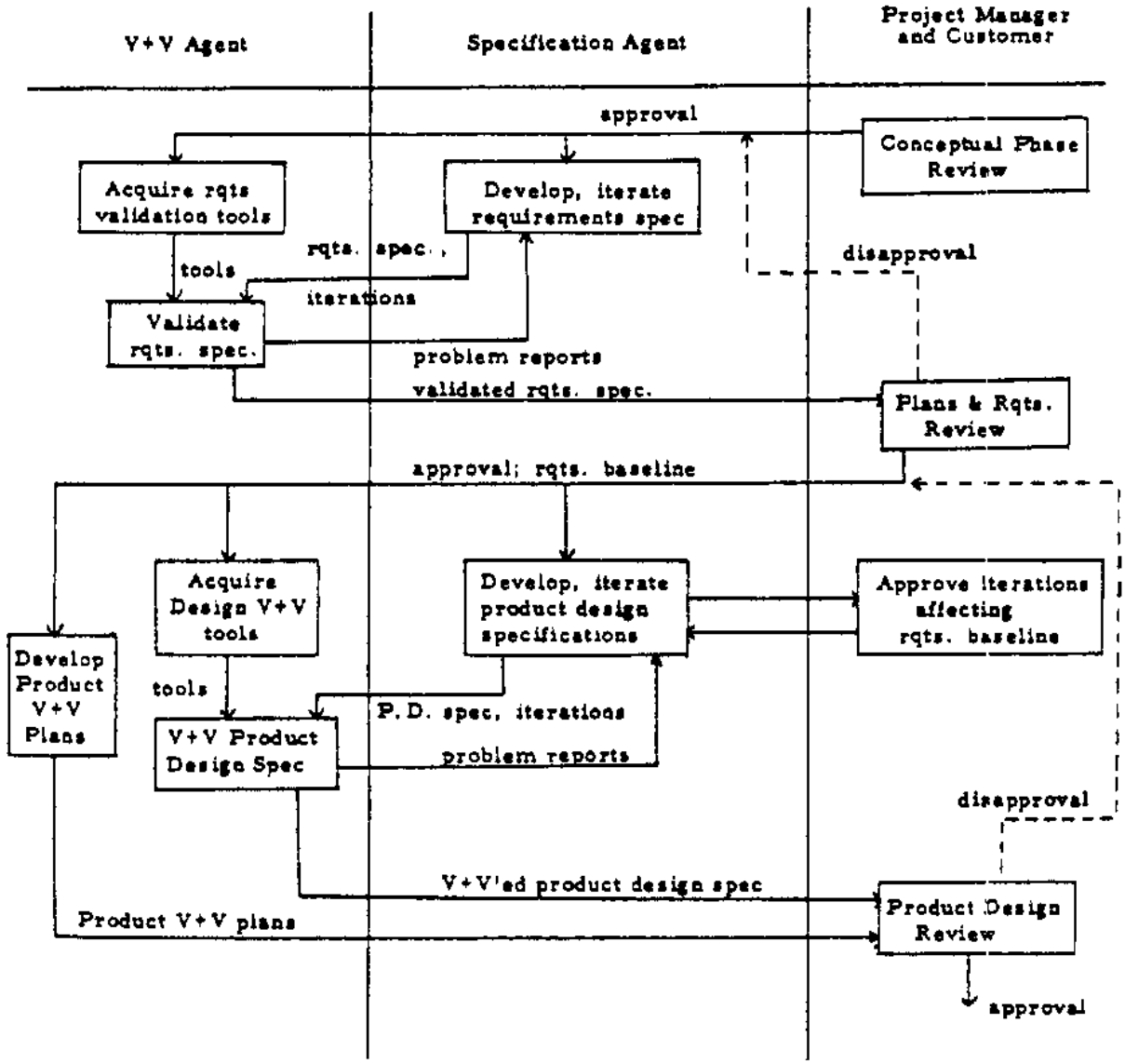


Figure 3: Requirements and Design V&V Sequences

The key portions of Figure 3 are the iterative loops involved in the V&V and iteration of the requirements and design specifications, in which:

The V&V agent analyzes the specifications and issues problem reports to the specification agent;

The specification agent isolates the source of the problem, and develops a solution resulting in an iteration of the spec;

The Project Manager and Customer approve any proposed iterations which would perceptibly change the requirements baseline;

The V&V agent analyzes the iterated specification, and issues further problem reports if necessary;

The process continues until the V&V agent completes his planned V&V activities, and all problem reports have been either fixed or assigned to a specific agent for resolution within a given time.

D. V&V Criteria

The four basic V&V criteria for requirements and design specifications are completeness, consistency, feasibility, and testability. An overall taxonomy of their components is given in Figure 4, and each is discussed in turn.

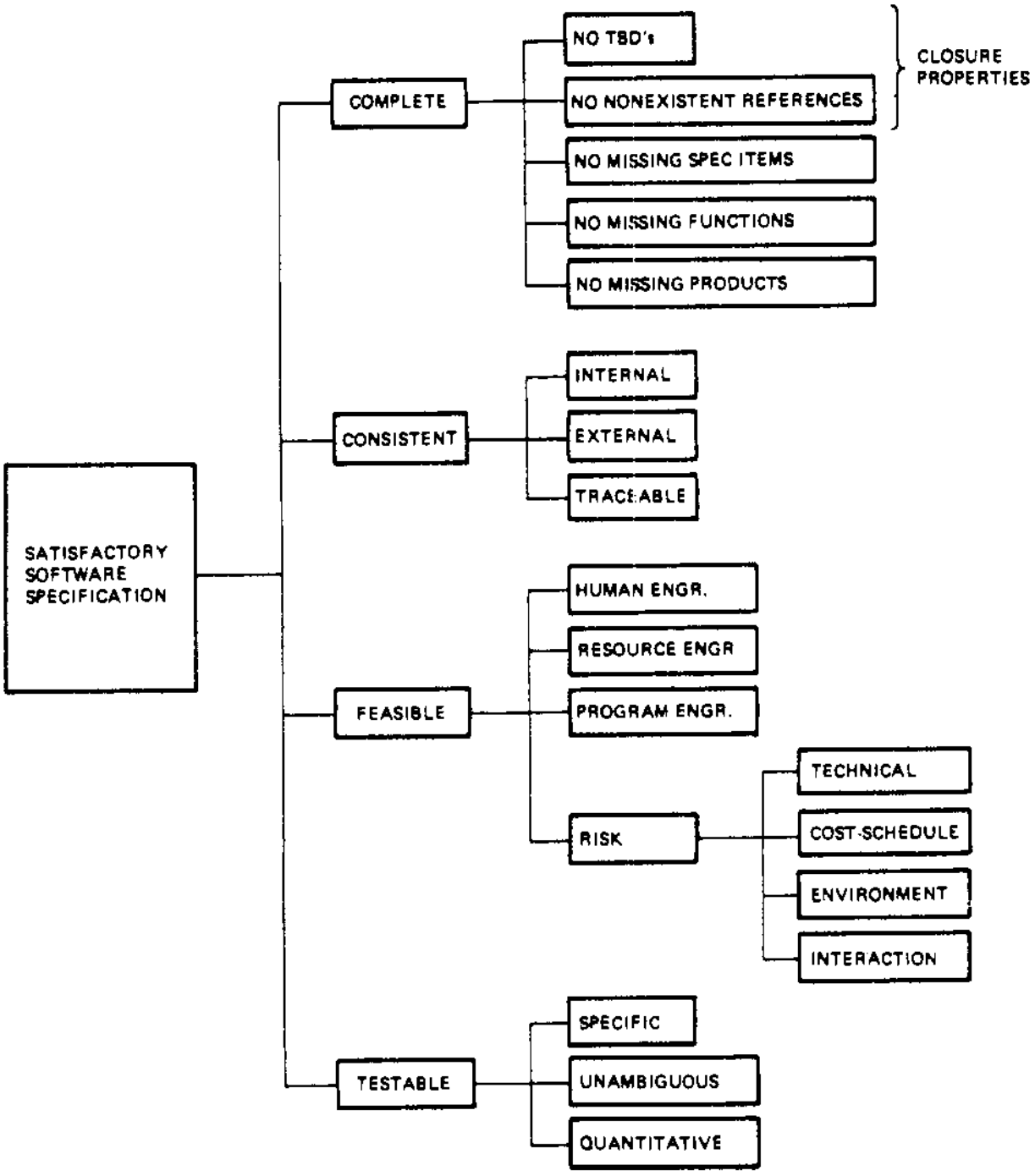


Figure 4: Taxonomy of a Satisfactory Software Specification

1. Completeness.

A specification is complete to the extent that all of its parts are present and each part is fully developed. There are several properties which a software specification must exhibit in order to assure its completeness. These are:

a. No TBDs. TBDs are places in the specification where decisions have been postponed by writing "To Be Determined," or "TBD."

   Examples: "The system shall handle a peak load of (TBD) transactions per second."

   "Update records coming from personnel information system shall be in the following format: TBD."

b. No reference to nonexistent functions, inputs, or outputs. These are references made in the specification to functions, inputs, or outputs (including data bases) which are not defined in the specification.

   Examples: "Function 3.1.2.3 Output 3.1.2.3.a Inputs 1. Output option flags obtained from the "User Output Options" function ..." – which is undefined in the specification.

   "A record of all transactions is retained in the Transaction File" – which is undefined.

c. No missing specification items. These are items which should be present as part of the standard format of the specification, but are not present.

   Examples: Missing verification provisions

   Missing interface specifications

   Note that verification of this property often involves a human judgment call: a small, stand-alone system may have no interfaces to specify.

d. No missing functions. These are functions which should have been part of the software product, but which are not called for in the specification.

e. No missing products. These are products which should be part of the delivered software, but which are not called for in the specification.

   Examples: Test tools

   Output postprocessors

The first two of these properties form a subset of the completeness properties which are called "closure" properties. Closure is distinguished by the fact that it can be verified by mechanical means; the last three properties generally require some human intuition to verify or validate.

2. Consistency.

A specification is consistent to the extent that its provisions do not conflict with each other or with governing specifications and objectives. Specifications require consistency in several ways. These are:

a. Internal Consistency. Items within the specification do not conflict with each other.

Counterexamples:

"Function x
1. Inputs: A 4 x 4 matrix z of reals.
…
Function y
…
3. Outputs: A 3 x 3 matrix z of integers."

"Page 14: Master Real-Time Control interrupts shall have top priority at all times ...
Page 37: A critical-level interrupt from the Security subsystem shall take precedence over all other processes and interrupts."

b. External Consistency. Items in the specification do not conflict with external specifications or entities.

Counterexamples:

"Spec: All safety parameters are provided by the Preprocessor System, as follows: ... "

"Preprocessor System Spec: The preprocessor initializes all safety parameters except for real-time control safety parameters, which are self-initialized ... "

c . Traceability. Items in the specification have clear antecedents in earlier specifications or statements of system objectives. Particularly on large specifications, each item should indicate the item or items in earlier specifications from which it derives.

Counterexamples:

Misinterpretations. Assuming that "on-line" storage implies a requirement for random access storage. (A dedicated on-line tape unit might be preferable.)

Embellishments. Adding exotic displays, natural language processing, or adaptive control features which are not needed for the job to be done (and may not work as reliably as simpler approaches).

d. Clerical errors and typos generally result in consistency problems.

3. Feasibility.

A specification is feasible to the extent that the life-cycle benefits of the system specified will exceed the life-cycle costs. Thus, feasibility involves more than verifying that a system can be developed which satisfies the functional and performance requirements. It also implies validating that the specified system will be sufficiently maintainable, reliable, and well human-engineered to keep the system's life-cycle balance sheet positive.

Further, and most importantly, it implies the identification and resolution of any high-risk issues involved, before the commitment of large numbers of people to detailed development.

Specifications require feasibility with respect to their human engineering, resource engineering, program engineering, and risk aspects, as discussed below.

a. Human engineering. Verifying and validating feasibility from a human engineering standpoint involves the following issues:

   - Will the specified system provide a satisfactory way for users to perform their operational functions?
   - Will the system satisfy human needs at appropriate levels of their needs hierarchy?
   - Will the system help people fulfill their human potential?

   Examples of human engineering considerations are given in the Human Engineering checklist in Ref. (3).

b. Resource engineering. This involves the following V&V issues:

   - Can a system be developed which satisfies the specified requirements (at an acceptable cost in resources)?
   - Will the specified system cost-effectively accommodate the expected growth in operational requirements over the system life-cycle?

   Examples of resource engineering considerations are given in the Efficiency checklist in Ref. (3).

c. Program engineering. This involves the following V&V issues:

   - Will the specified system be cost-effective to maintain?
   - Will the specified system be cost-effective from a portability standpoint?

- Will the specified system have sufficient accuracy, reliability, and availability to cost-effectively satisfy operational needs over the life-cycle?

Examples of these program engineering considerations are given in the checklists in Ref. (3) on Maintainability and Portability, and in the checklist on Reliability and Availability given in the Appendix.

d. Risk. If the 1ife-cycle cost-effectiveness of a specified system is extremely sensitive to some aspect of the system which is not well known or understood, there is a high risk involved in the system. If such high-risk issues are not identified and resolved in advance, there is a strong likelihood of disaster if and when this aspect of the system does not realize itself as expected.

Below are the four major sources of risk in software requirements and design specifications:

(1) Technical risk. Example issues are:

- Achievable levels of overhead in a multiprocessor operating system;
- Achievable levels of computer security protection;
- Achievable speed and accuracy of new algorithms;
- Achievable performance in "artificial intelligence" domains (e.g., pattern recognition, natural language processing);
- Achievable levels of man-machine performance (e.g., air traffic control).

(2) Cost-schedule risk. Examples include the sensitivity to cost and schedule constraints of such items as:

- Availability and reliability of the underlying virtual machine (hardware, operating system, data base management system, compiler) upon which the specified software will be built;
- Stability of the underlying virtual machine;
- Availability of key personnel;
- Strain on available main memory and execution time.

(3) Environment risk. Example issues are:

- Expected volume and quality of input data;

- Availability and performance of interfacing systems;
- Expected sophistication, flexibility, and degree of cooperation of system users.

A particular concern here is the assessment of second-order effects due to the introduction of the new system. For example, several airline reservation systems experienced overloads when they found that their new computer system capabilities stimulated many more requests and transactions per customer than were previously experienced. Of course, one can't expect to do this sort of prediction precisely. The important thing is to determine where your system performance is highly sensitive to such factors, and to concentrate your effort on risk-avoidance in those areas.

(4) Interaction effects. If your development is high-risk in several areas, you will find that the risks tend to interact exponentially. Unless you resolve the high-risk issues in advance, you will find yourself in the company of some of the supreme disasters in the software business.

Example: One large Government agency attempted to build a huge real-time inventory control system involving a nationwide network of supercomputers with:

- Extremely ambitious real-time performance requirements;
- A lack of qualified techniques for the operating system and networking aspects;
- Integration of huge, incompatible data bases;
- Continually changing external interfaces;
- A lack of qualified development personnel.

Although some of these were pointed out as high-risk items early, they were not resolved in advance. After spending roughly 7 years and $250M, the project failed to provide any significant operational capability and was cut off by Congress.

4. Testability.

A specification is <u>testable</u> to the extent that one can identify an economically feasible technique for determining whether or not the developed software will satisfy the specification. In order to be testable, specifications must be specific, unambiguous, and quantitative wherever possible. Below are some examples of specifications which are <u>not</u> testable:

- The software shall provide interfaces with the appropriate subsystems;
- The software shall degrade gracefully under stress;
- The software shall be developed in accordance with good development standards;
- The software shall provide the necessary processing under all modes of operation;
- Computer memory utilization shall be optimized to accommodate future growth;
- The software shall provide a 99.9999% assurance of information privacy (or "reliability," "availability," or "human safety," when these terms are undefined);
- The software shall provide accuracy sufficient to support effective flight control;
- The software shall provide real-time response to sales activity queries.

These statements are good as goals and objectives, but they are not precise enough to serve as the basis of a pass-fail acceptance test.

Below are some more testable versions of the last two requirements:

- The software shall compute aircraft position within the following accuracies:
  ± 50 feet in the horizontal;
  ± 20 feet in the vertical.
- The system shall respond to:
  Type A queries in ≤ 2 sec;
  Type B queries in ≤ 10 sec;
  Type C queries in ≤ 2 min;
  where Type A, B, C queries are defined in detail in the specification.

In many cases, even these versions will not be sufficiently testable without further definition. For example:

- Do the terms "± 50 ft" or "≤ 2 sec" refer to root-mean-square performance, 90% confidence limits, or never-to-exceed constraints?
- Does "response" time include terminal delays, communications delays, or just the time involved in computer processing?

Thus, it will often require a good deal of added effort to eliminate the vagueness and ambiguity in a specification and make it testable. But much effort is generally well worthwhile, for the following reasons:

- It would have to be done eventually for the test phase anyway;

- Doing it early eliminates a great deal of expense, controversy, and possible bitterness in later stages.

III. SOFTWARE REQUIREMENTS AND DESIGN V&V TECHNIQUES

This Section evaluates a number of techniques effective in performing software requirements and design V&V. They are not discussed in detail, but an evaluation is provided.

Simple Manual Techniques

Reading
Manual Crossreferencing
Interviews
Checklists
Manual Models
Simple Scenarios

Simple Automated Techniques

Automated Crossreferencing (4,5,6,7)
Simple Automated Models

Detailed Manual Techniques

Detailed Scenarios (8)
Mathematical Proofs (9,10)

Detailed Automated Techniques

Detailed Automated Mode1s (11,12)
Prototypes (13,14)

Evaluation of V&V Techniques

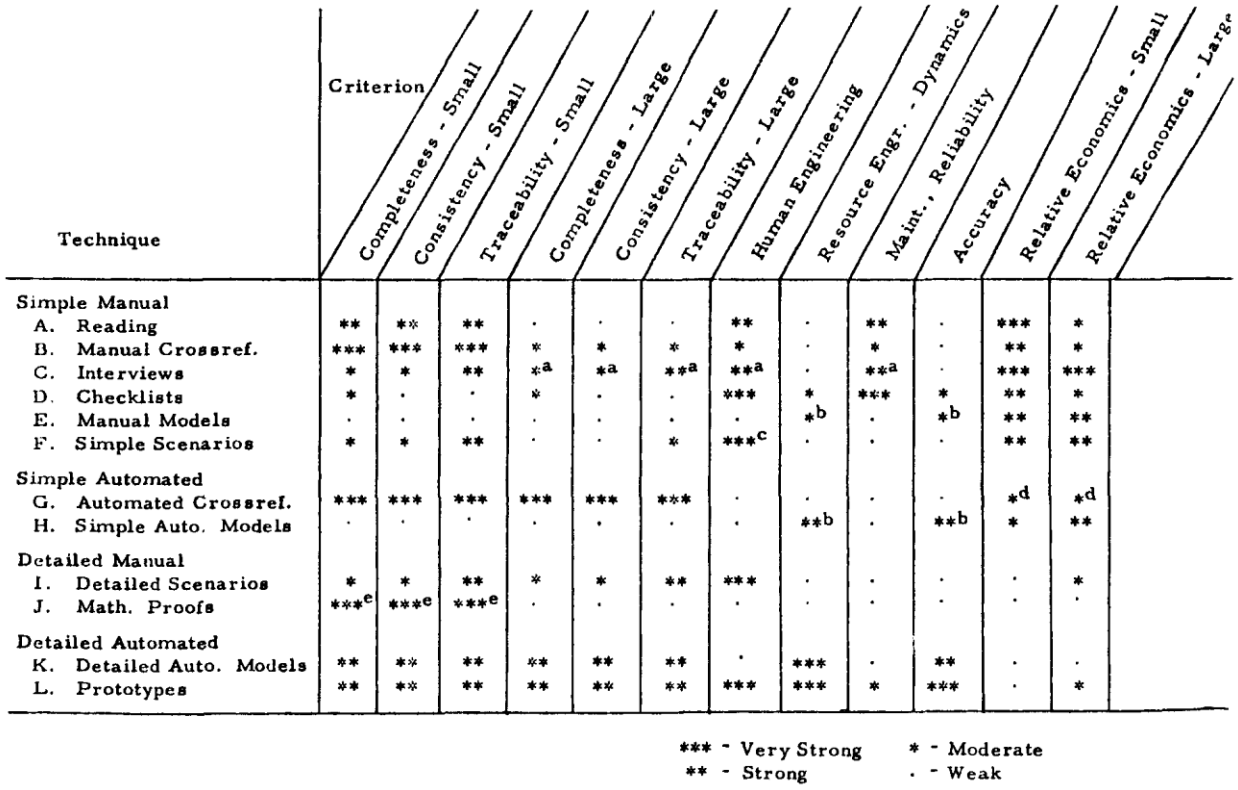
Figure 5 evaluates techniques discussed above with respect to their ability to help verify and validate specifications. This is done in terms of the basic V&V criteria discussed in Section II, with added assessments given for large and small specifications. Ratings in Figure 5 are on the following scale:

*** - Very Strong (technique very strong for V&V of this criterion)

** - Strong

* - Moderate

. - Weak



Technique / Criterion	Completeness - Small	Consistency - Small	Traceability - Small	Completeness - Large	Consistency - Large	Traceability - Large	Human Engineering	Resource Engr. - Dynamics	Maint., Reliability	Accuracy	Relative Economics - Small	Relative Economics - Large
Simple Manual												
A. Reading	**	**	**	.	.	.	**	.	**	.	***	*
B. Manual Crossref.	***	***	***	*	*	*	*	.	*	.	**	*
C. Interviews	*	*	**	*[a]	*[a]	**[a]	**[a]	.	**[a]	.	***	***
D. Checklists	*	.	.	*	.	.	***	*	***	*	**	*
E. Manual Models	.	.	.	.	.	.	.	*[b]	.	*[b]	**	**
F. Simple Scenarios	*	*	**	.	.	*	***[c]	.	.	.	**	**
Simple Automated												
G. Automated Crossref.	***	***	***	***	***	***	.	.	.	.	*[d]	*[d]
H. Simple Auto. Models	.	.	.	.	.	.	.	**[b]	.	**[b]	*	**
Detailed Manual												
I. Detailed Scenarios	*	*	**	*	*	**	***	.	.	.	.	*
J. Math. Proofs	***[e]	***[e]	***[e]	.	.	.	.	.	.	.	.	.
Detailed Automated												
K. Detailed Auto. Models	**	**	**	**	**	**	.	***	.	**	.	.
L. Prototypes	**	**	**	**	**	**	***	***	*	***	.	*

*** - Very Strong * - Moderate
** - Strong . - Weak

Figure 5: Summary of V&V Techniques

Notes in Figure 5 are explained below:

a. Interviews are good for identifying potential large-scale problems, but are not so good for detail.

b. Simple models are good for full analysis of small systems and top-level analysis of large systems.

c. Simple scenarios are very strong (***) for user aspects of small systems; strong (**) for large systems.

d. Economy rating for automated crossreferencing is higher (**) for medium systems. If the cross reference tool needs to be developed, reduce rating to weak (.).

e. Proofs provide near-certain correctness for the finite-mathematics aspects of the software.

IV. RECOMMENDATIONS FOR SMALL, MEDIUM, AND LARGE SPECIFICATIONS

A. Small System V&V

Customer and/or Project Manager

- Outline specification before opening
- Read specification for critical issues
- Interview with specification developer(s)
- Determine best allocation of V&V resources

  V&V Agent

  - Read specification; use checklists and manual crossreferencing
    - use simple models if accuracy or real-time performance are critical
    - use simple scenarios if user interface is critical
    - use mathematical proofs on portions if their reliability is extremely critical

  Users

  - Read specification from user point of view, using human engineering checklist

  Maintainers

  - Read specification from maintainer point of view, using maintainability and reliability/availability checklists. Use portability checklist if portability is a consideration

  Interfacers

  - Read specification; perform manual crossreferencing with respect to interfaces

Customer and/or Project Manager

- Coordinate efforts; resolve conflicts

B. Medium System V&V

Use same general approach as with small systems, with the following differences:

- Use automated crossreferencing if a suitable aid is available; otherwise use manual crossreferencing
- Use simple-to-medium manual and automated models for critical performance analyses
- Use simple-to-medium scenarios for critical user interfaces
- Prototype high-risk items which cannot get adequate V&V otherwise

C. Large System V&V

Use same general approach as with medium systems, except for use of simple-to-detailed models and scenarios instead of simple-to medium.

V. APPENDIX

RELIABILITY AND AVAILABILITY CHECKLIST

Reliable Input Handling

a. Are input codes engineered for reliable data entry?

   Comment: Examples are use of meaningful codes; increasing the "distance" between codes (NMEX, NYRK vs NEWM, NEWY); frequency-optimized codes (reducing the number of keystrokes as error sources).

b. Do inputs include some appropriate redundancy as a basis for error checking?

   Comment: Examples are checksums; error correcting codes; input boundary identifiers (e.g., for synchronization); check words, (e.g., name and employee number)

c. Will the software properly handle unspecified inputs?

   Comment: Options include use of default values, error responses, and fallback processing options.

d. Are all off-nominal input values properly handled?

   Comment: Options are similar to those in c. above. "Off-nominal" checking may refer not just to the range of values but also to the mode, form, volume, order, or timing of the input.

e. Are man-machine dialogues easy to understand and use for the expected class of users? Are they hard to misunderstand and misuse?

f. Are records kept of the inputs for later failure analysis or recovery needs?

Reliable Execution

g. Are there provisions for proper initialization of control options, data values, and peripheral devices?

h. Are there provisions for protection against singularities?

Comment: Examples are division by zero, singular matrix operations, proper handling of empty sets.

i. Are there design standards for internal error checking? Are they being followed?

Comment: These should tell under what conditions the responsibility for checking lies with the producer of outputs, the consumer of inputs, or a separate checking routine.

j. Are there design standards for synchronization of concurrent processes? Are they being followed?

k. Are the numerical methods—algorithms, tolerances, tables, constants—sufficiently accurate for operational needs?

l. Is the data structured to avoid harmful side effects?

Comment: Techniques include "information hiding," minimum essential use of global variables, and use of data cluster (data-procedure-binding) concepts.

m. Do the programming standards support reliable execution?

Comment: This topic is covered well in the Maintainability checklist. (l).

Error Messages and Diagnostics

n. Are there requirements for clear, helpful error messages?

Comment: These should be correct, understandable by maintenance personnel, expressed in user language, and accompanied by useful information for error isolation.

o. Are there requirements for diagnostic and debugging aids?

Comment: Examples are labeled. selective traces and dumps; labeled display of program

inputs; special hardware and software diagnostic routines; timers; self-checks; post-processors; control parameters for easily specifying diagnostic and debugging options.

p. Are measurement and validation points explicitly identified in the software specifications?

q. Are data base diagnostics specified for checking the consistency, reasonableness, standards compliance, completeness. and access control of the data base?

Backup and Recovery

r. Are there adequate provisions for backup and recovery capabilities?

Comment: These include capabilities for:

- Archiving of programs and data;
- Diagnosis of failure conditions;
- Interim fallback operation in degraded mode;
- Timely reconstitution of programs, data. and/or hardware;
- Operational cutover to reconstituted system.

s. Have these provisions been validated by means of a Failure Modes and Effects Analysis?

Comment: Potential failure modes include:

- Hardware outages and failure conditions (CPUs, memory, peripherals. communications);
- Loss of data (data base, pointers, catalogs, input data);
- Failure of programs (deadlockso unending loops, nonconvergence of algorithms).

VI. REFERENCES

(1) Boehm, B. W., "Software Engineering," IEEE Trans. Computers, Dec. 1976, pp. 1226-1241.

(2) personal communication from J. B. Munson, System Development Corporation. 1977.

(3) Lipow. M., B. B. White, and B. W. Boehm. "Software Quality Assurance: An Acquisition Guidebook," TRW-SS-77-07, November 1977 (IEEE Computer Society Repository No. R78-155).

(4) Boehm, B.W. R. K. McClean, and D. B. Urfrig, "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software," IEEE Trans. Software Engr., Mar. 1975, pp. 125-133

(5) Bell, T. E., D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. Software Engr., Jan 1977, pp. 49-59.

(6) Alford, M. W., "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Trans. Software Engr., Jan. 1977, pp. 60-68.

(7) Teichroew, D., and E. A. Hershey, III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. Software Engr., Jan. 1977, pp. 41-48.

(8) Myers, G., Software Reliability, Wiley-Interscience, New York, 1976.

(9) London, R. L., "A View of Program Verification." Proceedings, 1975 IEEE/ACM International Conference on Reliable Software, April 1975.

(10) Zelkowitz, M., ed., Proceedings, 1979 Conference on Specifications of Reliable Software, IEEE, Cambridge, MA, April 1979.

(11) Dreyfus. J. M., "Use of Simulation in the BMD Systems Technology Program Software Development," Proceedings, 1976 Summer Computer Simulation Conference, IEEE, Washington, D.C. July 1976.

(12) Dreyfus, J. M., and P. Karacsony, "The Preliminary Design as a Key to Successful Software Development," Proceedings, Second International Conference on Software Engineering, IEEE/ACM, October 1976.

(13) Royce, W. W., "Software Requirements Analysis, Sizing and Costing," in Practical Strategies for Developing Large Software Systems, E. Horowitz. ed., Addison-Wesley, 1975

(14) Boehm, B. W., "Software Design and Structuring," in Practical Strategies for Developing Large Software Systems, E. Horowitz, ed., Addison-Wesley. 1975.