Verifying and Validating Software Requirements and Design Specifications

Barry W. Boehm, TRW

These recommendations provide a good starting point for identifying and resolving software problems early in the life cycle—when they're still relatively easy to handle.

"Don't worry about that specification paperwork. We'd better hurry up and start coding, because we're going to have a whole lot of debugging to do."

How many projects start out this way and end up with either a total failure or a tremendously expensive self-fulfilling prophecy? There are still far too many, but more and more projects are turning the self-fulfilling prophecy around. By investing more up-front effort in verifying and validating their software requirements and design specifications, these projects are reaping the benefits of reduced integration and test costs, higher software reliability and maintainability, and more user-responsive software. To help increase their number, this article presents the following guideline information on verification and validation, or V&V, of software requirements and design specifications:

- definitions of the terms "verification" and "validation," an explanation of their context in the software life cycle, and a description of the basic sequence of V&V functions;
- 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 major criteria; and
- an example V&V checklist for software system reliability and availability.

Based on the above, we recommend combinations of software requirements and design V&V techniques that are most suitable for small, medium, and large software specifications.

Verification and validation in the software life cycle

Definitions. The recent *IEEE Standard Glossary of Software Engineering Terminology*[1] defines "verification" and "validation" as follows:

- *Verification.* The process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase.
- *Validation.* The process of evaluating software at the end of the software development process to ensure compliance with software requirements.

In this article we extend the definition of "validation" to include a missing activity at the beginning of the software definition process: determining the fitness or worth of a software product for its operational mission.

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

- *Verification.* "Am I building the product right?"
- *Validation.* "Am I building the right product?"

Objectives. The basic objectives in verification and validation 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.[2] It shows that, for large projects, savings of up to 100:1 are possible by finding and fixing problems early in the

0740-7459/84/0100/0075$01.00

life cycle. For smaller projects, the savings are more on the order of 4-6:1, but this still provides a great deal of leverage for early investment in V&V activities. Besides the major cost savings, there are also significant payoffs in improved reliability, maintainability, and human engineering of the resulting software product.

Early life-cycle specifications and phases. In general, experience has shown that the software development process proceeds most cost-effectively if software specifications are produced in the following order:

(1) a requirements specification, which states the functions the software must perform, the required level of performance (speed, accuracy, etc.), and the nature of the required interfaces between the software product and its environment;

(2) a product design specification, which describes the overall architecture of the software product and its components; and

(3) a detailed design specification, which identifies the inputs, outputs, control logic, algorithms, and data structures of each individual low-level component of the software product.

The typical software life cycle includes requirements, product design, and detailed design phases that involve the development, verification and validation, approval or disapproval, and baselining of each of these specifications (see Figure 2). However, the nature of the V&V process causes intermingling of the activities associated with each phase. For example, one cannot validate the feasibility of a performance-critical requirement without doing some design and analysis of ways to implement the requirement. Similarly, some design and development of code in a working prototype may be necessary to validate the user-interface requirements.

V&V functions in the early phases. Verification and validation activities produce their best results when performed by a V&V agent who operates independently of the developer or specification agent. The basic sequence of functions performed by the V&V agent, the specification agent (the analyst or software system engineer), the project manager, and the customer are shown in Figure 2.

The key portions of Figure 2 are the iterative loops 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 specification;
- the project manager and customer approve any proposed iterations that would perceptibly change the requirements baseline; and
- 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 activities and all problem reports have been either fixed or assigned to a specific agent for resolution within a given time.

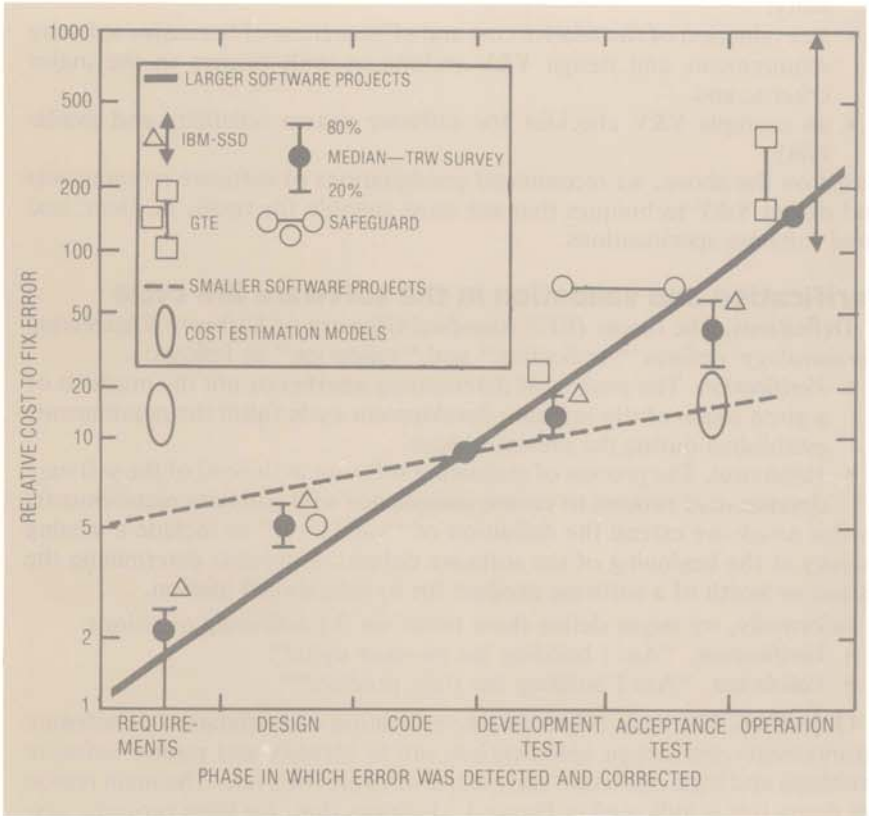

Figure 1. Increase in cost to fix or change software throughout the life cycle.[2]

Verification and validation 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 3, and each is discussed in turn below.

Completeness. A specification is *complete* to the extent that all of its parts are present and each part is fully developed. A software specification must exhibit several properties to assure its completeness.

No TBDs. TBDs are places in the specification where decisions have been postponed by writing "To be Determined" or "TBD." For example:

- "The system shall handle a peak load of (TBD) transactions per second."
- "Update records coming from

the personnel information system shall be in the following format: (TBD)."

No nonexistent references. These are references in the specification to functions, inputs, or outputs (including databases) not defined in the specification. For example:

- "Function 3.1.2.3 Output
  3.1.2.3.a Inputs
  1. Output option flags obtained from the User Output Options functions...," which is undefined in the specification.
- "A record of all transactions is retained in the Transaction File," which is undefined.

No missing specification items. These are items that should be present as part of the standard format of the specification, but are not present. For example:

- No verification provisions
- No 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.

No missing functions. These are functions that should be part of the software product but are not called for in the specification. For example:

- No backup functions
- No recovery functions

No missing products. These are products that should be part of the delivered software but are not called for in the specification. For example:

- Test tools
- Output postprocessors

The first two properties—"no TBDs" and "no nonexistent references"—form a subset of the completeness properties 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.

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.

Internal consistency. Items within the specification do not conflict with

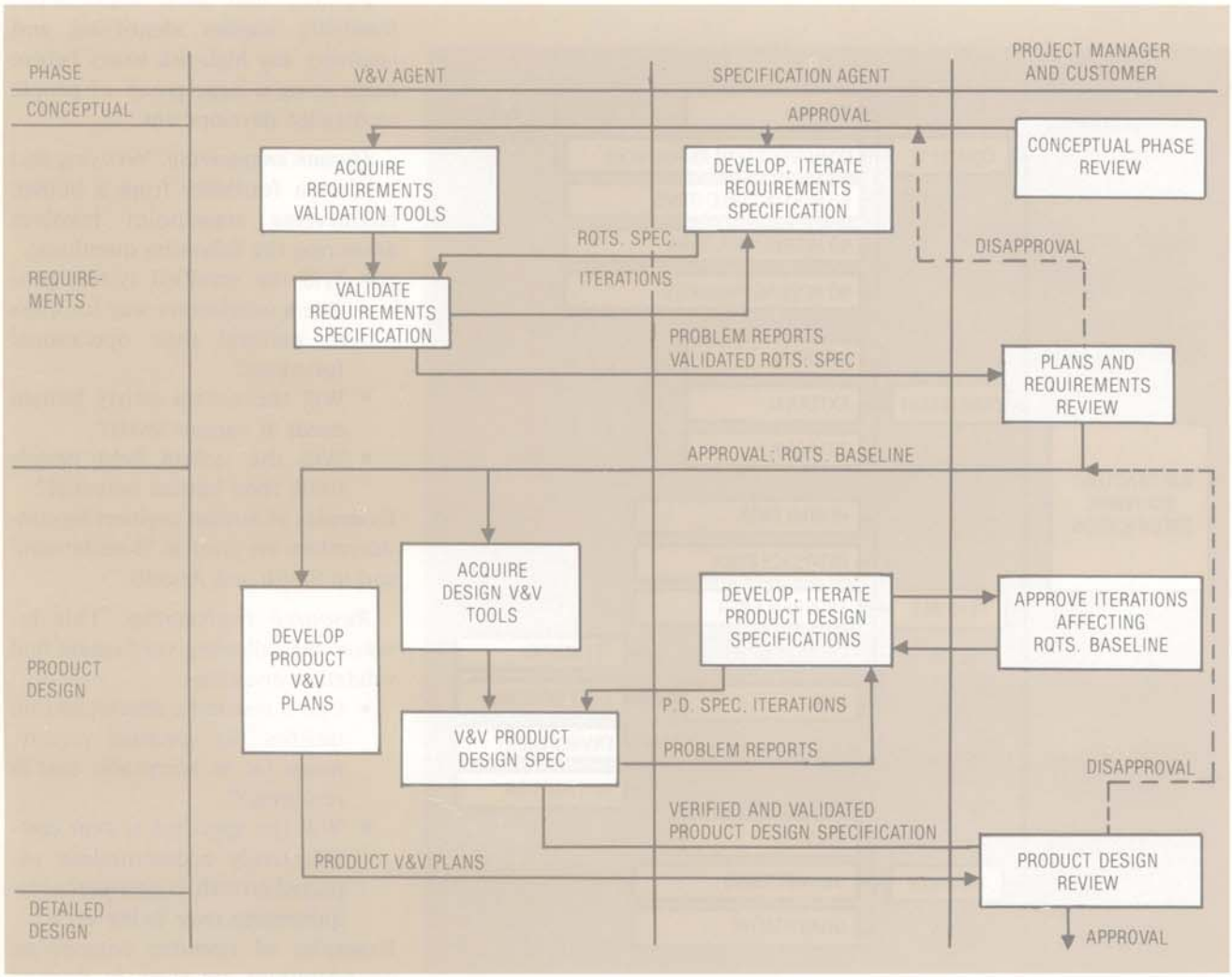

Figure 2. Verification and validation sequences for the requirements and product design phases.

each other, unlike the following counter-examples:

- "Function x
  (1) Inputs: A 4×4 matrix z of reals.
  ⋮
  Function y
  ⋮
  (3) Outputs: A 3×3 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."

External consistency. Items in the specification do not conflict with external specifications or entities, unlike the following counter-example:

- "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. . ."

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 is derived to prevent

- misinterpretations, such as assuming that "on-line" storage implies a requirement for random access storage (a dedicated on-line tape unit might be preferable); and
- embellishments, such as adding exotic displays, natural language processing, or adaptive control features that are not needed for the job to be done (and may not work as reliably as simpler approaches).

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

Further, and most importantly, feasibility implies identifying and resolving any high-risk issues before committing a large group of people to detailed development.

Human engineering. Verifying and validation feasibility from a human engineering standpoint involves answering the following questions:

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

Examples of human engineering considerations are given in Shneiderman[3] and in Smith and Aucella.[4]

Resource engineering. This involves the following verification and validation questions:

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

Examples of resource engineering considerations are given in Boehm[1] and Ferrari.[5]

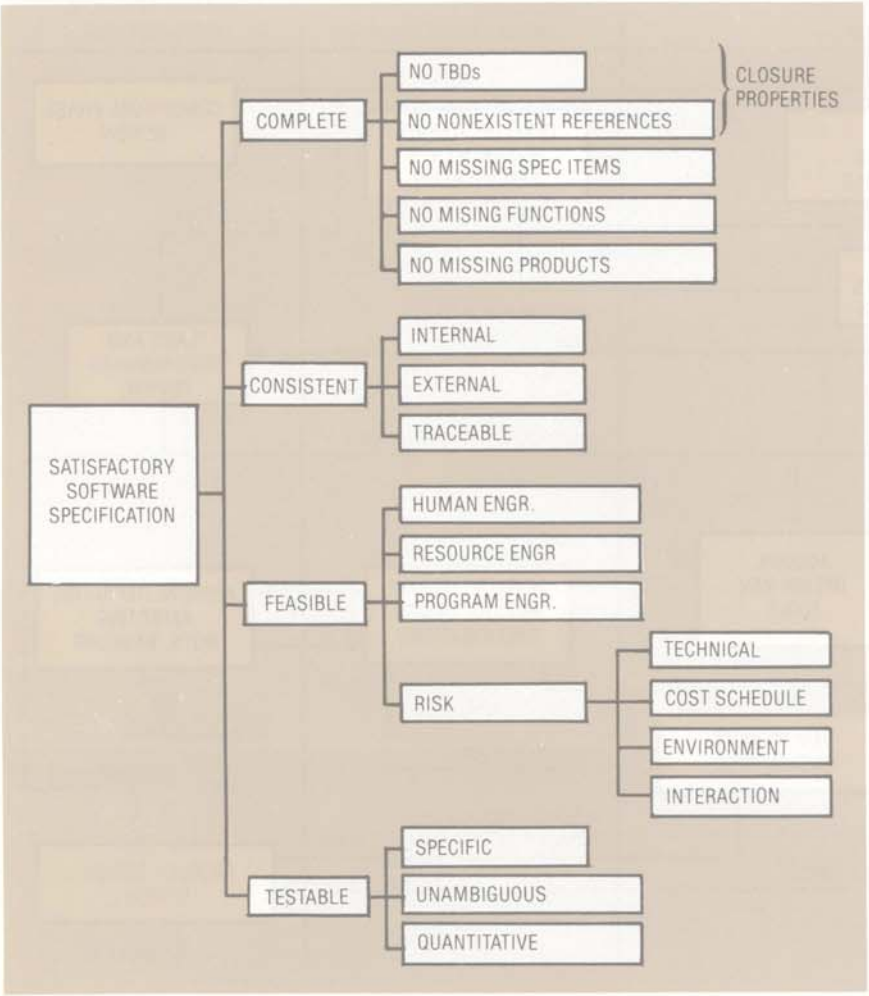

Figure 3. Taxonomy of a satisfactory software specification.

Program engineering. This addresses the following questions regarding a specified system:

- Will it be cost-effective to maintain?
- Will it be cost-effective from a portability standpoint?
- Will it have sufficient accuracy, reliability, and availability to cost-effectively satisfy operational needs over its life cycle?

Examples of these program engineering considerations are given in Lipow, White, and Boehm's[6] checklists on maintainability and portability and in the checklist on reliability and availability on p. 87.

Risk. If the life-cycle cost-effectiveness of a specified system is extremely sensitive to some system aspect that 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 or when this aspect of the system is not realized as expected.

Four major sources of risk in software requirements and design specifications are technical, cost-schedule, environmental, and interaction effects.

Technical risk, for example, can involve

- 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 "artifical intelligence" domains (e.g., pattern recognition, natural language processing); and
- achievable levels of man-machine performance (e.g., air traffic control).

Cost-schedule risks include the sensitivity to cost and schedule constraints of such items as

- availability and reliability of the underlying virtual machine (hardware, operating system, database management system, compiler) upon which the specified software will be built;
- stability of the underlying virtual machine;
- availability of key personnel; and
- strain on available main memory and execution time.

Some environmental risk issues are

- expected volume and quality of input data;
- availability and performance of interfacing systems; and
- expected sophistication, flexibility, and degree of cooperation of system users.

A particular concern here is the assessment of second-order effects caused by introduction of the new system. For example, several airline reservation systems experienced overloads because new capabilities stimulated additional customer requests and transactions. Of course, this sort of reaction can't be predicted precisely. The important thing is to determine where system performance is highly sensitive and to concentrate risk-avoidance efforts in those areas.

If the development is high-risk in several areas, the risks tend to interact exponentially. Unless you resolve the high-risk issues in advance, you may find yourself in the company of some of the supreme disasters in the software business.

For 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 databases;
- continually changing external interfaces; and
- 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 seven years and $250 million, the project failed to provide any significant operational capability and was cut off by Congress.

Testability. A specification is *testable* to the extent that one can identify an economically feasible technique for determining whether or not the developed software will satisfy the specification. To be testable, specifications must be specific, unambiguous, and quantitative wherever possible. Below are some examples of specifications which are *not* 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 percent 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 seconds;
  Type B queries in ⩽10 seconds;

Type C queries in ≤2 minutes; where Type A, B, and 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 feet" or "≤2 seconds" refer to root-mean-square performance, 90-percent 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 often requires a good deal of added effort to eliminate a specification's vagueness and ambiguity and make it testable. But such effort is generally well worthwhile: It would have to be done eventually for the test phase anyway, and doing it early eliminates a great deal of expense, controversy, and possible bitterness in later stages.

Verification and validation techniques

A number of techniques, outlined in Figure 4 and evaluated below, are effective in performing software requirements and design verification and validation.

Simple manual techniques. Five relatively simple and easily implemented manual techniques—reading, cross-referencing, interviews, checklists, and models—can provide much valuable information for meeting verification and validation criteria.

Simple manual techniques:
- Reading
- Manual cross-referencing
- Interviews
- Checklists
- Manual models
- Simple scenarios

Simple automated techniques:
- Automated cross-referencing
- Simple automated models

Detailed manual techniques:
- Detailed scenarios
- Mathematical proofs

Detailed automated techniques:
- Detailed automated models
- Prototypes

Figure 4. Verification and validation techniques.

Reading. Having someone other than the originator read the specification to identify potential problems is often referred to as "reviewing." Here, however, we call it "reading" and reserve the term "reviewing" for a more formal activity.

Since reading subjects the specification to another point of view, it is very good for picking up any blind spots or misconceptions that the specification developer might have. This is particularly true if the reader is going to be one of the product's testers, users, maintainers, interfacers, or program developers; a tester can, for example, verify that the specification is testable and unambiguous. Another strength of reading is that it requires little preparation. It is also extremely flexible with respect to when, to where, and to what level of detail it is done.

Reading's strong points can turn into weak points if the "little preparation" it does require is not carried out. Readers can waste a lot of time—looking for the wrong things or looking for nothing in paticular—that could have been spent bringing a valuable perspective to focus on a set of significant issues. This is a particular danger on large projects. For detailed designs, the design inspection or walkthrough described by Fagan[7] can be particularly effective. Still, reading is fundamentally limited in the extent to which it can be used to verify a detailed specification's completeness and consistency, or the feasibility of a complex system's performance requirements.

Manual cross-referencing. Cross-referencing goes beyond reading; it involves constructing cross-reference tables and various diagrams—for example, state transition, data flow, control flow, and data structure diagrams—to clarify interactions among specified entities. These entities include functions, databases, and interfacing hardware, software, or personnel.

Manual cross-referencing is effective for the consistency (internal, external, and traceability) and closure properties of a specification, particularly for small to medium specifications. For large specifications, it can be quite cumbersome, leading to the suggested use of automated aids. If these are not available, manual methods are still recommended; the payoff will outweigh the cost and time expended.

Manual cross-referencing will not do much to verify the feasibility of performance requirements or to validate the subjective aspects of human engineering or maintainability provisions.

Interviews. Discussing a specification with its originator will identify potential problems. With minimum effort, you can find out a great deal about its strengths and weaknesses; this will allow you to deploy your V&V resources most effectively by concentrating on the weak points. Interviews are particularly good for identifying potential blind spots, misunderstandings, and high-risk issues. But, like spot-checking, they only identify and scope the specification's major problem areas; the detailed V&V work remains to be done.

Checklists. Specialized lists, based on experience, of significant issues for assuring successful software development can be used effectively with any of the manual methods described above.

Checklists are excellent for catching omissions, such as the missing items, functions, and products discussed under "Completeness." They are also valuable aids in addressing some of the life-cycle feasibility considerations: human engineering, maintainability, reliability and availability, portability, security and privacy, and life-cycle efficiency. But they are not much help in verifying the feasibility of performance re-

quirements or in dealing with detailed consistency and closure questions.

One danger with checklists is that items might be considered absolute requirements rather than suggestions. For example, several features in a portability checklist will not be necessary if the software will never be used on another machine; adding them blindly will just incur unnecessary expense.

Manual models. Mathematical formulas can be used to represent and analyze certain aspects of the system being specified. These manual models are very good for analyzing some life-cycle feasibility issues, particularly accuracy, real-time performance, and life-cycle cost. They are also useful for risk analysis. They are not, however, much help in verifying the details of a specification's consistency and completeness or in assessing subjective factors. Their manual nature makes them inefficient for detailed feasibility analysis of large, complex systems, but they are good for top-level analysis of large systems.

Simple scenarios. Scenarios describe how the system will work once it is in operation. Man-computer dialogues are the most common form of simple scenarios, which are very good for clarifying misunderstandings or mismatches in the specification's human engineering aspects but not for checking completeness and consistency details or for validating performance speed and accuracy.

Simple automated techniques. Automation extends the power of two manual techniques—cross-referencing and simple modeling.

Automated cross-referencing. Automated cross-referencing involves the use of a machine-analyzable specification language—for example, SREM-RSL, Software Requirements Engineering Methodology-Requirements Statement Language;[8,9] PSL/PSA, Problem Statement Language/Problem Statement Analyzer;[10] or PDL, Program Design Language.[11] Once a specification is expressed in such a language, it can be automatically analyzed for consistency, closure properties, or presentation of cross-reference information for manual analysis. Some further automated aids in this category are discussed in the DoD "Methodman" document[12] and its references.

Current automated specification aids have only limited capabilities in addressing accuracy and dynamic performance issues.

Automated cross-referencing is excellent for verifying the detailed consistency and closure properties of both small and large specifications. Using a formatted specification language also eliminates many testability and ambiguity problems because the language forms help prevent ambiguous terms and vague generalities. The automated systems also have less of a problem in checking for additional clerical errors introduced in iterating and retyping a specification.

Current automated specification aids have only limited capabilities in addressing accuracy and dynamic performance issues, and some, particularly SREM-RSL, are not available on many computers. Although their performance on small and medium specifications has improved considerably, they are still somewhat inefficient in performing consistency and completeness checks on very large specifications. Even so, the costs of using them on large specifications are more than repaid by the savings involved in early error detection.

Simple automated models. Mathematical formulas implemented in a small computer program provide more powerful representations than manual models for analyzing such life-cycle feasibility issues as accuracy, real-time performance, and life-cycle costs. Simple automated models are especially good for risk and sensitivity analysis, but, like manual models, are not much help in verifying detailed consistency and completeness, in assessing subjective factors, or in performing detailed feasibility analysis of large, complex systems.

Detailed manual techniques. Detailed scenarios and mathematical proofs are especially effective for clarifying human engineering needs and for verifying finite-mathematics programs, respectively.

Detailed scenarios. Detailed scenarios, which provide more elaborate—and thus more expensive—operational descriptions, are even more effective than simple scenarios in clarifying a system's human engineering aspects.

Mathematical proofs. Mathematical transformation rules can be applied to a set of statements, expressed in a precise mathematical specification language, to prove desired properties of the set of statements. These properties include internal consistency, preservation of "invariant" relations, and equivalence of two alternate sets of statements (e.g., requirements and design specifications). Automated aids to formal specification and verification such as Special/HDM, Gypsy, Affirm, and Ina Jo, are now available and have been compared by Cheheyl et al.[13]

For certain classes of problems—small problems involving the use of finite mathematics—mathematical proofs offer a near-certain guarantee of the properties proved. But mathematical proofs cannot deal with nonformalized inputs (e.g., noisy sensor data, natural language); cannot deal conveniently with "real-variable" calculations or with many issues in synchronizing concurrent processes; and cannot deal efficiently with large specifications.

Detailed automated techniques. Two final techniques—detailed automated models and prototypes—provide the most complete information.

Detailed automated models. Detailed automated models typically involve large event simulations of the system. While more expensive than simple automated models, they are much more effective in analyzing such issues as accuracy, dynamic consistency, real-time performance, and life-cycle cost. The process of generating such models also serves as a good way to catch specification inconsistencies and ambiguities.

Prototypes. In many situations, the feasibility of a specification cannot be conclusively demonstrated without developing some working software. Examples include some accuracy and real-time performance issues, as well as user interface suitability. In such cases, it is often valuable to build a throwaway prototype of key portions of the software: a program that works well enough to resolve the feasibility issues, but lacks the extensive features, error-proofing, and documentation of the final product. The process of building the prototype will also expose and eliminate a number of ambiguities, inconsistencies, blind spots, and misunderstandings incorporated in the specification.

Prototypes can be expensive and do not shed much light on maintainability, but they are often the only way to resolve the critical feasibility issues. (See References 14 and 15 for experiences in prototyping and Chapter 20 of Reference 1 for guidelines on when to prototype.)

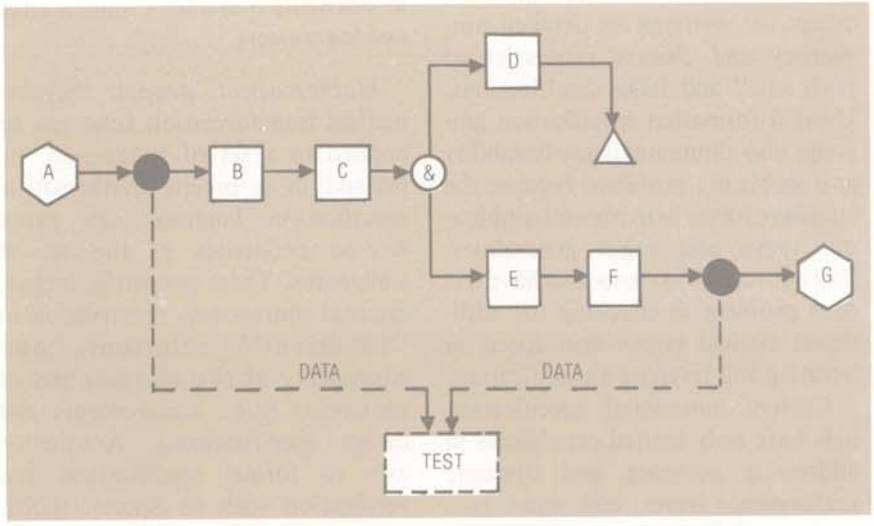

Figure 5. An example of a path analysis structure. Validation points (dark circles) are added to the paths close to the interfaces (hexagons).

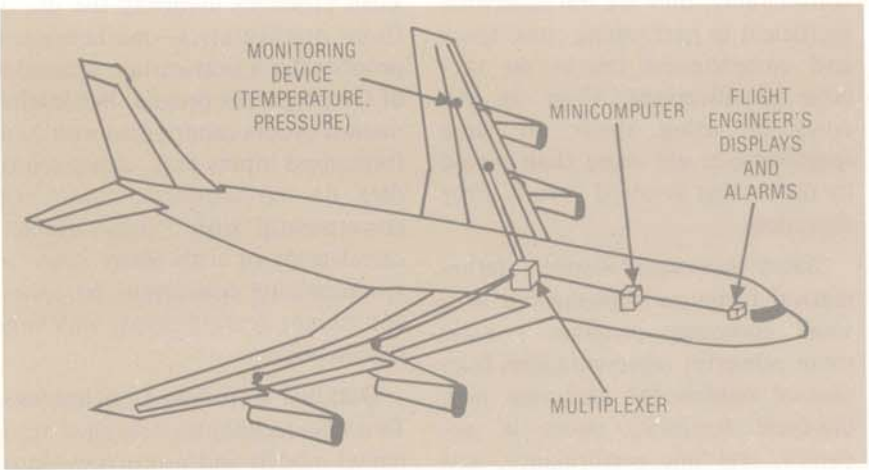

Figure 6. An aircraft engine monitoring system.

An automated aid for requirements V&V

Several available systems—PSL/PSA, SREM, PDL, Special/HDM—provide automated aids to requirements and design verification and validation. To get a feel for what these systems do, let us look at how one of them—the Software Requirements Engineering Methodology, or SREM—handles a simple example, an aircraft engine monitoring system.[16]

Requirements networks. The SREM Requirements Statement Language expresses software requirements in terms of processing paths—that is, the sequences of data processing required to operate on an input stimulus to the software and produce an output response.

The first step in developing an SREM specification is to write it as a description of the processing paths. Thus, if some data is input to a processing path, and a response is required at some other point, we write this in the specification as a fundamental component. To produce several thousand paths for a complex system, a method was developed to integrate simple paths into requirements networks called R_NETS. Thus, all paths initiated by the same input interface are integrated into a common network.

Figure 5 illustrates a network with data coming across an interface (represented by a hexagon). All data coming across the interface have common processing, as indicated by the first two boxes. The AND node is next encountered to indicate a "don't care" parallelism. That is, either branch may be processed first after the AND node. That decision is left as an option for the designer. Validation points (the dark circles) are used to specify performance re-

quirements in an explicitly testable fashion and are added to the paths close to the interfaces. Data is specified that must be available for testing purposes at those points, and an explicit procedure is described for analyzing those data to give a pass/fail relationship.

One advantage of this type of network is that it is integrated from the collection of paths. Therefore, it can be automatically analyzed, using the rules established for R_NETs. Another advantage is that it is similar to classic logic diagrams and flowcharts used by engineers, making it a natural form of communication between engineers. Finally, it possesses natural test points for assuring that the requirement expressed on a path is expressly addressed for testing.

R_NETs, then, are the basic tool used by SREM to define functional requirements unambiguously by showing each path of processing required by the input stimuli to the system.

Engine monitoring system. This simple example for an engine monitoring system, or EMS, will show how the R_NET concept is used in describing requirements.

An airplane with multiple engines has a device that is connected to each engine (see Figure 6). This device measures temperatures, pressure, and the state of two switches. All of the devices are connected to the multiplexer, which is interrogated by an on-board computer. The computer senses when a temperature or pressure goes out of an allowed range and gives an alarm in the cockpit.

A partial system specification for the EMS capabilities might look like this:

(1) Monitor 1 to 10 engines.

(2) Monitor (a) 3 temperatures (0 to 1000°C) (b) 3 pressures (0 to 4000 PSIA), and (c) 2 switches (off, on).

(3) Monitor each engine at a specified rate (1 to 10 times per second).

(4) Output a warning message if any parameter falls outside prescribed limits, and an alarm if outside danger limits.

(5) Record history of each engine.

The R_NET approach (see Figure 7) for defining the EMS functional

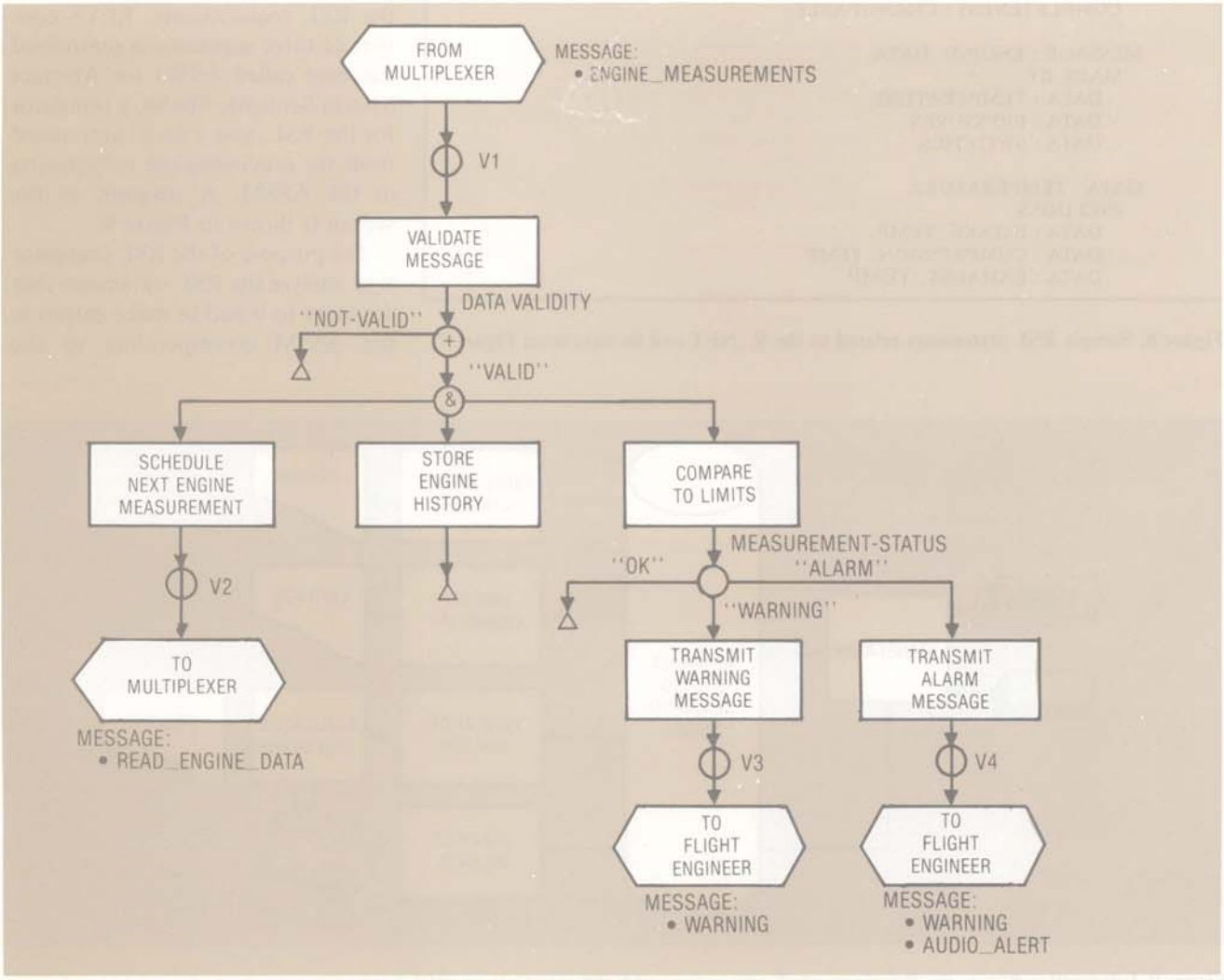

Figure 7. The R_NET approach for the engine monitoring system shown in Figure 6.

requirement presents a clear, unambiguous definition of how each stimulus (called a MESSAGE in SREM terminology) is processed, and where the VALIDATION_POINTs (V1, V2, V3, V4) reside that allow testing of the PERFORMANCE_REQUIREMENTs specified for selected functional processes (paths).

As a result of preparing the R_NET and accomplishing automated error checks, many questions arise. These are typical types of ambiguities resident in system specifications, and the SREM methodology brings them out early and quite clearly. Answers must be attained before completion of the software specification. Examples for the EMS are

(1) Does "output warning" mean each time or just the first time?

(2) How are "prescribed limits" defined?

(3) How quickly must the system output a "warning" or "alarm?"

(4) What does a "warning message" contain?

(5) What is to be done with the history data for each engine?

```
INPUT_INTERFACE : MULTI.
  DESCRIPTION : 3 TEMPERATURES 3 PRESSURES 2 SWITCHES.
  ENABLES : R_NET MONITOR_ENGINES.
  PASSES : MESSAGE ENGINE_DATA.
  DOCUMENTED BY
    SOURCE : TRW_CLASS_NOTES.
  COMPLETENESS : CHANGEABLE.

MESSAGE : ENGINE_DATA.
  MADE BY
    DATA : TEMPERATURE
    DATA : PRESSURES
    DATA : SWITCHES.

DATA : TEMPERATURE.
  INCLUDES
    DATA : INTAKE_TEMP
    DATA : COMPRESSION_TEMP
    DATA : EXHAUST_TEMP.
```

Figure 8. Sample RSL statements related to the R_NET and its data from Figure 7.

Later, we will see how some of these questions were identified by SREM procedures.

Requirements statement language. The SREM approach to attaining explicitness throughout a requirement specification is grounded in the use of the Requirements Statement Language. RSL is a machine-processible, artificial language which overcomes the shortcomings of English in stating requrements. RSL is based on the entity-attribute-relationship model of representing information. Figure 8 illustrates RSL statements related to the R_NET and its data from Figure 7. This figure illustrates the English-like nature of RSL.

Requirements tools and database. The Requirements Engineering and Validation System is an integrated set of tools to support development of the RSL requirements. REVS consists of three segments: a centralized database called ASSM for Abstract System Semantic Model, a translator for the RSL, and a set of automated tools for processing the information in the ASSM. A diagram of the system is shown in Figure 9.

The purpose of the RSL translator is to analyze the RSL statements that are input to it and to make entries in the ASSM corresponding to the

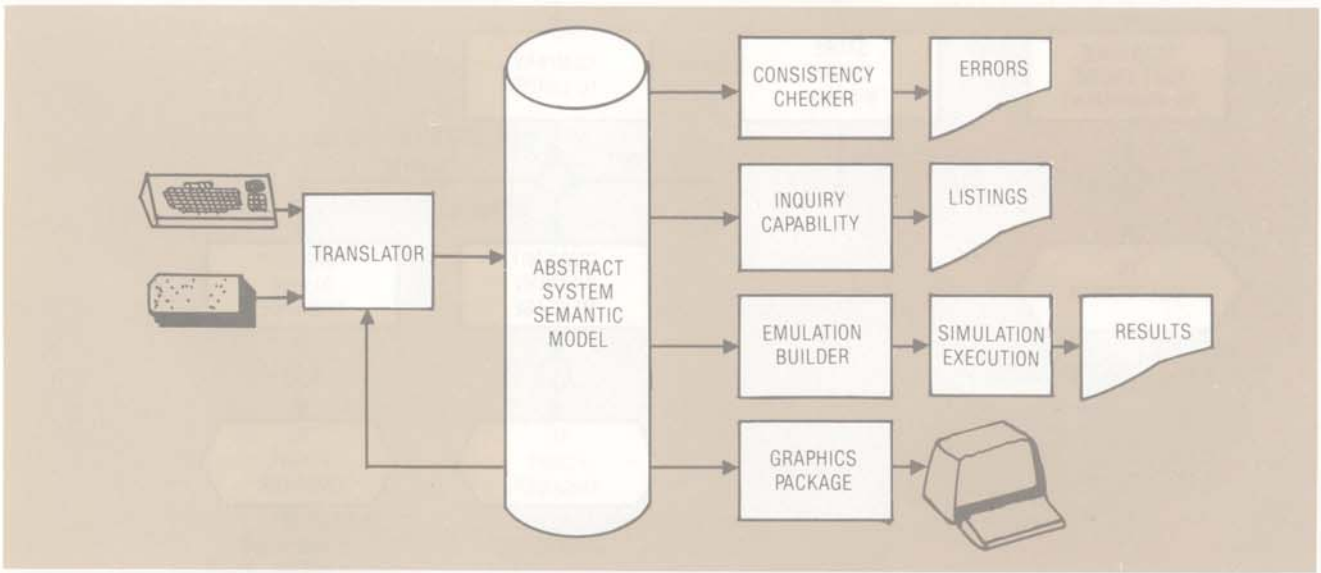

Figure 9. Information flow in the Requirements Engineering and Validation System. REVS is an integrated set of tools that supports the Requirements Statement Language.

meaning of the statements. The translator references the ASSM to do simple consistency checks on the input. This prevents the occurrence of disastrous errors such as the introduction of an element with the same name as a previously existing element or an instance of a relationship that is tied to an illegal type of element. This type of checking catches, at an early stage, some of the simple types of inconsistencies that are often found in requirements specifications.

The information available in the ASSM will support a wide variety of requirements analysis tools. Among these are an interactive graphics package to aid in the specification of the flow paths; static consistency checkers, which check for consistency of information throughout the system; and an automated simulation generator and execution package, which aids in the study of dynamic interactions of the various requirements. Situation-specific reports and analyses, which a particular user or manager may need, are generated through the use of the Requirements Analysis and Data Extraction system. The RADX system is independent of the extensions to RSL so that new concepts added to the language may be included in queries to the database.

A key consideration is that all steps in the SREM approach, including simulations, utilize a common requirements database, which is necessary since many individuals are continually adding, deleting, and changing information about requirements for the data processing system. Centralizaton allows both the requirements engineers and the analysis tools to work from a common baseline and enables implementation of management controls on changes to this baseline.

Automated consistency and completeness checking. Figure 10, a RADX printout for the engine monitoring system, illustrates the RADX query-response capability to test adequacy of engineer inputs to the database.

First, RADX identified that accuracy and response-time limits have not yet been specified. This feature is designed to assure that all appropriate paths are eventually covered by performance requirements.

Next, RADX identified data with no sink—that is, data produced but

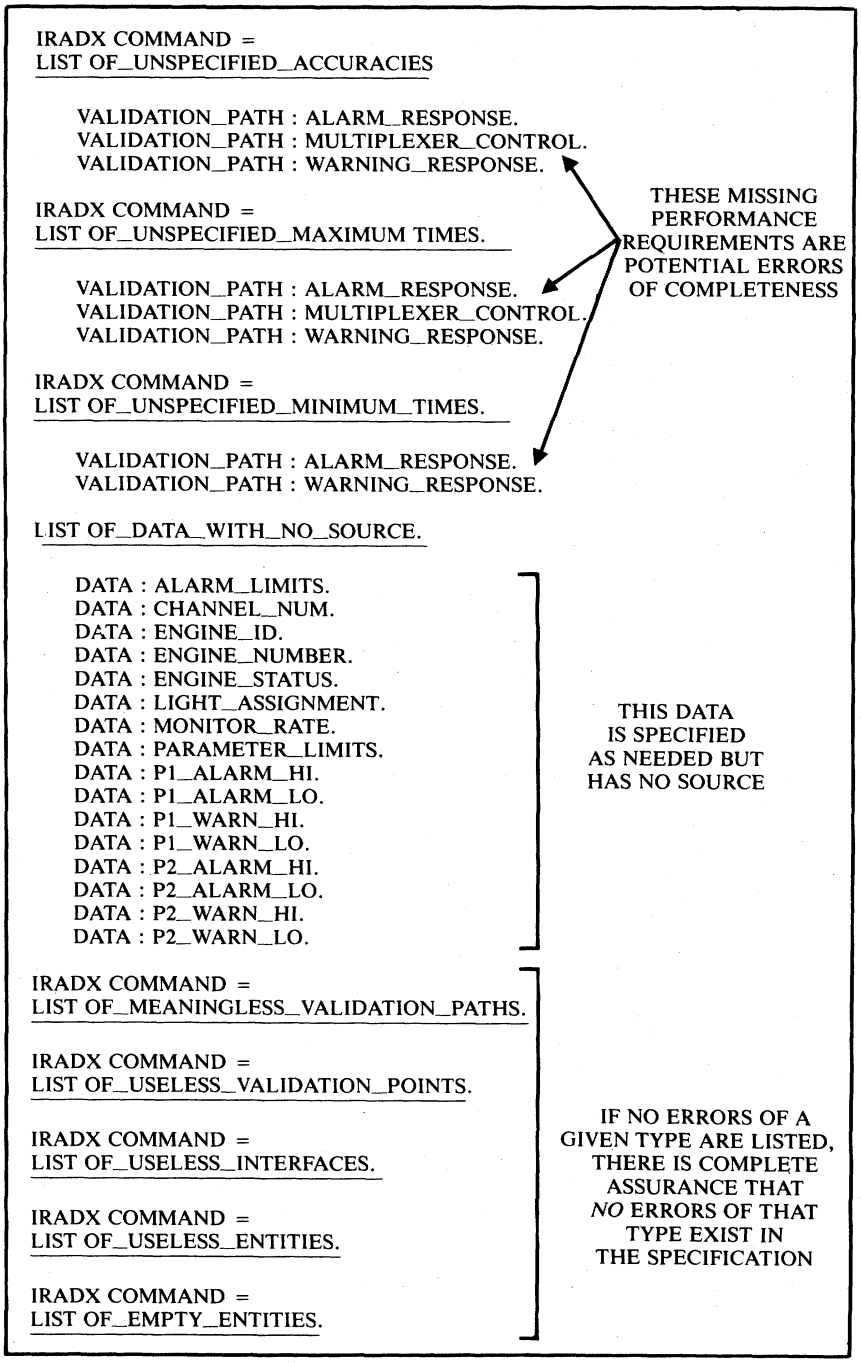

Figure 10. Error checks in the Requirements Analysis and Data Extraction system provide immediate, automatic analysis of requirements completeness and consistency.

never used, since it is not input to a processing step and is not contained in an output message. Thus, the engine history that was required to be maintained is never accessed. This probably represents an incompleteness—that is, an overload set of processing requirements.

Finally, there is a set of error categories with no indicated errors of these types in the specifications. When there are no errors of any type remaining, the database will support a good specification for detailed software design and coding.

In sum, this example shows how an automated verification and validation aid can help us avoid and eliminate problems in our requirements and design specifications. This type of aid is particularly good in resolving completeness and consistency problems, but it does not provide much help in resolving user-interface or maintainability issues. This means that we need to employ a mix of techniques to cover our full range of criteria.

Evaluation and recommendations

Evaluation of V&V techniques. Table 1 provides a way to determine the best mix of V&V techniques for a particular project. It evaluates the techniques discussed above with respect to their ability to help verify and validate specifications. The evaluation is done in terms of the basic V&V criteria discussed earlier.

A retrospective analysis indicated that SREM could have caught 63 percent of the requirements problems on one project.

Some quantitative information is also becoming available on the relative effectiveness of those techniques. For example, one retrospective analysis indicated that SREM could have caught 63 percent of the requirements problems on one project. Several studies of design inspections have indicated that they can catch 60 percent of the errors present. Further quantitative information can be found in Chapter 24 of Reference 1.

Small-system V&V. To verify and validate a small specification, the customer and/or project manager should

- outline the specification before opening,
- read the specification for critical issues,
- interview the specification developer(s),
- determine the best allocation of V&V resources, and
- coordinate efforts and resolve conflicts.

The V&V agent should

- read the specification,
- use checklists and manual cross-referencing,
- use simple models if accuracy or real-time performance are critical,

Table 1.
Verification and validation techniques rated according to eight criteria.

	Completeness		Consistency		Traceability		Human	Re-source	Maint.,	Accur-	Relative Economics	
	Small	Large	Small	Large	Small	Large	Engr.	Engr.	Relia.	acy	Small	Large
Simple Manual Techniques												
Reading	**	—	**	—	**	—	**	—	**	—	***	*
Manual Cross-Referencing	***	*	***	*	***	*	*	—	*	—	**	*
Interviews	*	*a	*	*a	**	**a	**a	—	**a	—	***	***
Checklists	*	*	—	—	—	—	***	*	***	*	**	*
Manual Models	—	—	—	—	—	—	—	*b	—	*b	**	**
Simple Scenarios	*	—	*	—	**	*	***c	—	—	—	**	**
Simple Automated Techniques												
Automated Cross-Referencing	***	***	***	***	***	***	—	—	—	—	*d	*d
Simple Automated Models	—	—	—	—	—	—	—	**b	—	**b	*	**
Detailed Manual Techniques												
Detailed Scenarios	*	*	*	*	**	**	***	—	—	—	—	*
Mathematical Proofs	***e	—	***e	—	***e	—	—	—	—	—	—	—
Detailed Automated Techniques												
Detailed Automated Models	**	**	**	**	**	**	—	***	—	**	—	*
Prototypes	**	**	**	**	**	**	***	***	*	***	—	*

Ratings:
*** Very strong
** Strong
* Moderate
— Weak

Notes:
a. Interviews are good for identifying potential large-scale problems but not so good for detail.
b. Simple models are good for full analysis of small systems and for top-level analysis of large systems.
c. Simple scenarios are very strong for small-system and strong for large-system user aspects.
d. Economy rating for automated cross-referencing is strong for medium systems. If the cross-reference tool needs to be developed, reduce the rating to weak.
e. Proofs provide near-certain correctness for the finite-mathematics aspects of software.

- use simple scenarios if the user interface is critical, and
- use mathematical proofs on portions where reliability is extremely critical.

Users should

- read the specification from the user's point of view and
- use a human-engineering checklist.

Maintainers should

- read the specification from the maintainer's point of view,
- use maintainability and reliability/availability checklists, and
- use a portability checklist if portability is a consideration.

Interfacers should

- read the specification, and
- perform manual cross-referencing with respect to the interfaces.

Medium-system V&V. To verify and validate a medium specification, use the same general approach as

Reliability and availability checklist

Rather than being long and exhaustive, this checklist concentrates on high-leverage issues in obtaining a highly reliable and available system.

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), and 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), and 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-normal input values properly handled?

Comment: Options are similar to those in (c). "Off-normal" 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, and 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?

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

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

(q) Are database diagnostics specified for checking the consistency, reasonableness, standards compliance, completeness, and access control of the database?

Backup and recovery

(r) Are there adequate provisions for backup and recovery capabilities?

Comment: These include capabilities for archiving programs and data, diagnosing failure conditions, interim fallback operation in degraded mode, timely reconstitution of programs, data, and/or hardware, and operational cutover to the 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 (database, pointers, catalogs, input data), and failure of programs (deadlocks, unending loops, nonconvergence of algorithms).

with small systems, with the following differences:

- use automated cross-referencing if a suitable aid is available,
- use simple-to-medium manual and automated models for critical performance analyses,
- use simple-to-medium scenarios for critical user interfaces, and
- prototype high-risk items that cannot be adequately verified and validated by other techniques.

Large-system V&V. Use the same general approach for large specifications as for medium systems, except use simple-to-detailed instead of simple-to-medium models and scenarios.

For special situations, of course, you will have to tailor your own best mix of verification and validation techniques. But in general, these recommendations will provide you with a good starting point for identifying and resolving your software problems while they're still relatively easy to handle. ■

References

1. *IEEE Standard Glossary of Software Engineering Terminology,* IEEE Std. 729-1983, IEEE-CS order no. 729, Los Alamitos, Calif., 1983.
2. B. W. Boehm, *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, N.J., 1981.
3. B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems,* Winthrop Press, Cambridge, Mass., 1980.
4. S. L. Smith and A. F. Aucella, "Design Guidelines for the User Interface to Computer-Based Information Systems," ESD-TR-83-122, USAF Electronic Systems Division, Bedford, Mass., Mar. 1983.
5. D. Ferrari, *Computer Systems Performance Evaluation,* Prentice-Hall, Englewood Cliffs, N.J., 2nd ed., 1983.
6. M. Lipow, B. B. White, and B. W. Boehm, "Software Quality Assurance: An Acquisition Guidebook," TRW-SS-77-07, Nov. 1977.
7. M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal,* Vol. 15, No. 3, 1976, pp. 182-211.
8. M. W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Trans. Software Engr.,* Vol. SE-3, No. 1, Jan. 1977, pp. 60-68.
9. T. E. Bell, D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Trans. Software Engr.,* Vol. SE-3, No. 1, Jan. 1977, pp. 49-59.
10. D. Teichroew and E. A Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Software Engr.,* Vol. SE-3, No. 1, Jan. 1977, pp. 41-48.
11. S. H. Caine and E. K. Gordon, "PDL: A Tool for Software Design," *AFIPS Conf. Proc.,* Vol. 44, 1975 NCC, pp. 272-276.
12. P. Freeman and A. I. Wassermann, "Ada Methodologies: Concepts and Requirements," DoD Ada Joint Program Office, Nov. 1982.
13. M. H. Cheheyl et al., "Verifying Security," *ACM Computing Surveys,* Sept. 1981, pp. 279-340.
14. S. L. Squires, B. Branstad, and M. Zelkowitz, eds., "Special Issue on Rapid Prototyping," *ACM Software Engr. Notes,* Dec. 1982.
15. B. W. Boehm, T. E. Gray, and T. Seewaldt, "Prototyping vs. Specifying: A Multi-Project Experiment," *IEEE Trans. Software Engr.,* 1984 (in publication).
16. M. W. Alford, R. P. Loshbough, and L. R. Marker, "The Software Requirements Engineering Methodology: An Overview," TRW Software Series, TRW-SS-80-03, May 1980.

Barry W. Boehm is a member of the *IEEE Software* Editorial Board. His photo and biography appear on p. 6.