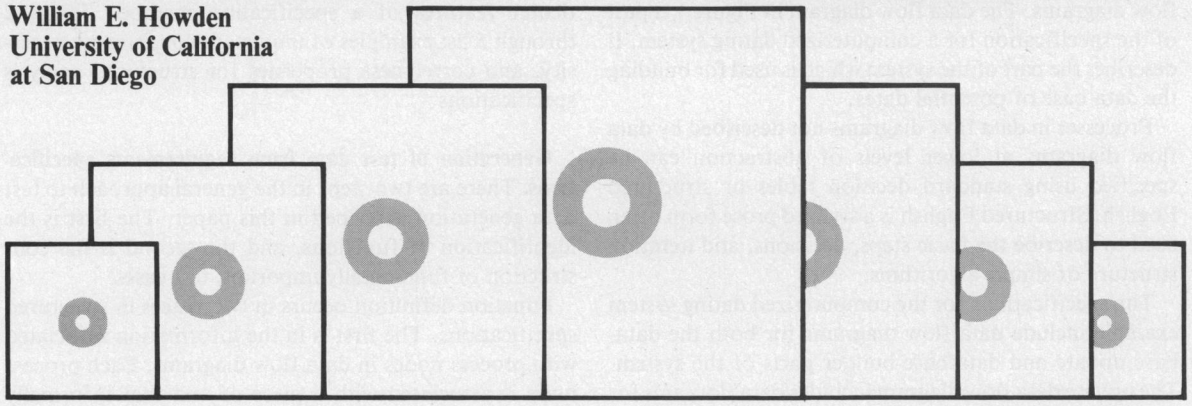
SPECIAL FEATURE:

LIFE-CYCLE SOFTWARE VALIDATION

William E. Howden
University of California
at San Diego



Two validation activities should take place during each phase of the software life cycle: analysis and test data generation. There are many ways to approach these tasks.

The software life cycle can be divided into requirements, design, programming, and maintenance. Validation has also been considered a phase of the life cycle and is sometimes inserted between programming and maintenance. Recent experience, however, indicates that validation should be integrated into all phases rather than isolated in a separate stage that takes place long after requirements and design have been completed. Studies show that the later validation is carried out, the more expensive it becomes to find errors made early in the development process.[1]

In the integrated approach described in this article, validation is a part of each phase of the life cycle. Two validation activities—analysis and test data generation—take place during each phase. The programming and maintenance phases also include actual execution of program tests. Analysis involves the direct examination of specifications and code for errors or erroneous properties. Test data generation involves the construction of test cases that are based on the important functional properties of specifications and code.

Requirements

Validation and requirements specifications. It is difficult to discuss the details of specification-based validation activities outside the context of a particular specification method. Of the different specification methods that have been developed (SADT, PDL, PSA/PSL, etc.), structured analysis[2] is among the most general, and its use is assumed in the following discussion.

Structured analysis specifications include data flow diagrams, data dictionaries, structured English and decision-table process descriptions, and file and data-base schemata. Various aspects of the method can be emphasized to suit various areas of application. The data flow diagram is often the most important part of the specification for a data processing application. The specification for a scientific program might consist entirely of a complex process description and data dictionary entries for describing input and output data.

Figure 1 contains a typical data flow diagram of the type used in structured analysis. The circles, or "bub-

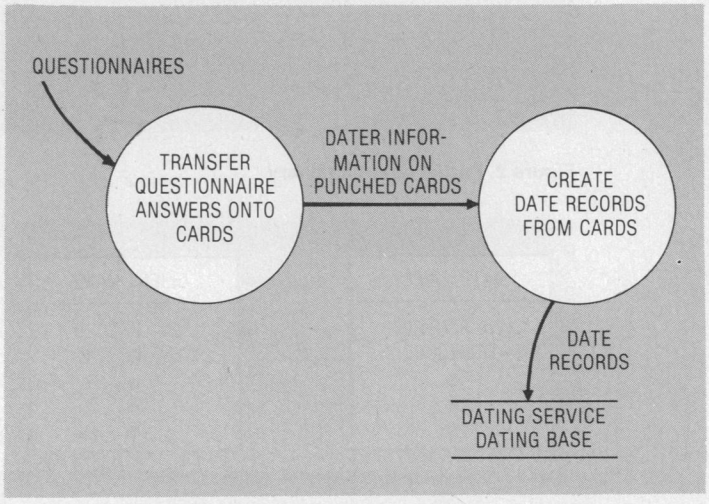


Figure 1. Data flow diagram for dating service data-base builder.

An earlier version of this paper appeared in the Infotech publication *Life-Cycle Management,* Infotech, Maidenhead, England, 1980, pp. 101-116.

0018-9162/82/0200-0071$00.75 © 1982 IEEE

bles," describe processes that must be carried out by the system. The arrows, or graph edges, describe data that flows between processes. The processes in a data flow diagram can be described by other, less abstract data flow diagrams. The data flow diagram in Figure 1 is part of the specification for a computerized dating system. It describes the part of the system which is used for building the data base of potential dates.

Processes in data flow diagrams not described by data flow diagrams at lower levels of abstraction can be specified using standard decision tables or structured English. Structured English is a stylized prose form often used to describe the basic steps, decisions, and iteration structure of simple algorithms.

The specifications for the computerized dating system example include data flow diagrams for both the data-base update and data-base builder parts of the system. The update data flow diagram includes data flow arcs for date-transactions. Figure 2 contains the data dictionary entries that define the structure and content of a date-transaction. It specifies that a date-transaction can be one of three types: a date-history, a new-date, or a delete-date-transaction. A date-history consists of a blisscount and from one to 10 compatible characteristic codes. New dates and delete-dates are also defined in the dictionary entries.

Figure 3 contains a simple data-base schema for defining a dating-system data base that consists of two files, one keyed by a date-name field and the other by a client-name field. The schema indicates that it must be possible to access the client records for clients listed in the date records of the date file.

Analysis of requirements specifications. The principle idea in the analysis of requirements specifications is to make sure that they have certain necessary properties. Table 1 lists some of these properties.

Specific guidelines for requirements analysis can be constructed by relating the properties in Table 1 to particular features of a specification method. Tables 2 through 5 list examples of important consistency, necessity, and correctness properties for structured analysis specifications.

Generation of test data from requirements specifications. There are two steps in the general approach to test data generation described in this paper. The first is the identification of functions, and the second is the construction of functionally important test cases.

Function definition occurs in two places in structured specifications. The first is in the information associated with process nodes in data flow diagrams. Each process node is associated with a piece of text that informally describes the functions performed by the process denoted by that node. Process nodes at the lowest level of abstraction in a data flow diagram are described in more detail by more formal definitions of the functions they perform.

DATE-TRANSACTION → | DATE-HISTORY |
| NEW-DATE |
| DELETE-DATE |

NEW DATE → NAME + PERSONAL CHARACTERISTICS + PREFERENCE CHARACTERISTICS + {INTEREST CODES}$_1^{20}$

DATE-HISTORY → BLISSCOUNT + {COMPATIBLE CHARACTERISTICS CODE}$_1^{10}$

DELETE-DATE → NAME

Figure 2. Partial data dictionary.

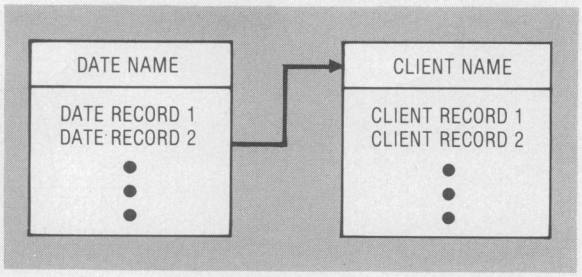


Figure 3. Dating system data-base schema.

Table 1.
Important properties of requirements and design specifications.

PROPERTY	COMMENTS
Consistency	Specifications information must be internally consistent. If the information is duplicated in different documents, consistency between copies must be maintained.
Completeness	Specifications must be examined for missing or incomplete requirements and design information. All specification functions must be described, including important properties of data.
Necessity	Each part of the specified system should be necessary and not redundant.
Feasibility	The specified system should be feasible with existing hardware and technology.
Correctness	In some cases, it is possible to compare part of the specification with an external source for correctness.

Table 2.
Consistency properties for structured specifications.

PROPERTY	COMMENTS
Level Balancing	The data flow into and out of a process node in a data flow diagram must match the data flow into and out of data from subdiagrams used to refine that process at lower levels of abstraction.
Data Conservation	The information required to generate the data flow from a process node in a data flow diagram must be consistent with the information available from the data flow into the node.
Flow/Dictionary Consistency	There should be exactly one data dictionary entry for every data flow item in the data flow diagram.

Function definitions also occur implicitly in data-base schemata. Data-base functions are used to reference, update, and create data bases and files. They must be able to update and retrieve data from files on the basis of key values and follow each branch in an access path through a data base. The implicitly defined access functions of the schema in Figure 3 must be capable of referencing and updating records in the date and client files on the basis of date and client names. They must also be able to follow the single interfile access path and reference the set of all client records associated with any given date.

Several general principles can be identified for the generation of functionally important test cases. The input and output data for a function often fall into various abstract classes, each of which corresponds to a conceptually different functional capability of a program. A test must be selected for each class. The data within a class can have important numeric properties (length, numeric value, etc.), whose allowable range of values is bounded (e.g., length $\geqslant 1$). *Extremal test values* correspond to data whose property values lie at the extremes of allowable ranges. *Special values* are associated with different types of functions. For example, functions that carry out sequences of algebraic computations are error-prone over zeros, numbers small in absolute value, and numbers large in absolute value. Functions that move data from one location to another should be tested over distinct data sets, and those that compute relations between data should be tested over identical sets of data.

The input and output data for a function can often be written as a compound structure of constituent substructures.[3] Test data sets in which different functionally important values for constituent substructures are combined should be constructed in order to form functionally important values for compound structures.[4]

The input and output data for the process functions in a structured specification are described by data dictionary entries. Systematic procedures can be developed for deriving functionally important test cases from data dictionary definitions. The definitions in Figure 2 indicate that there are three classes of date-transactions. Test cases should be constucted for each. A new-date type of data value contains from one to 20 interest codes. This is a property of the data, and extremal test cases in which the property is both one and 20 should be selected. A date-history is a compound data item consisting of a blisscount and from one to 10 compatible characteristics codes. Test cases that contain different combinations of important values for each of the constituent parts of a date-history must be selected.

Table 3.
Completeness properties for structured specifications.

PROPERTY	COMMENTS
Data Sufficiency	All information required by a process should be contained in the data flow into the process.
Dictionary Completeness	Complete dictionary entries must be provided for each input and output data flow. At the lower levels of abstraction, the entries should describe type, dimensions, units, and allowable ranges of values.
Process Sufficiency	All required system processes must be represented by a process node in the flow diagram.

Table 4.
Necessity properties of structured specifications.

PROPERTY	COMMENTS
Extraneous Data Flow	The information in each data flow should be needed by the receiving processes.
File Sinks	It should be possible to associate files declared at lower levels of abstraction in a data flow diagram with data flows at higher levels. Otherwise, the file might not be necessary in the more abstract data flows and might be redundant.

Table 5.
Correctness properties of structured specifications.

PROPERTY	COMMENTS
Formula Selection	Mathematical formula appearing in process descriptions should be correct.
Decision Table Construction	The relationship between the conditions and the actions in decision tables should be correct.

Design

Validation and design specifications. Analysis and test data generation methods similar to those applied to requirements specifications can be used to validate design specifications. An additional set of methods can be used to compare requirements and design specifications and assure their mutual consistency.

The following discussion of design validation is based on the assumed use of structured design.[5] Structured designs consist of hierarchies of abstract modules or functions.[6] The function(s) at the top of the hierarchy denote the overall functional capability of the program. Functions at lower levels denote functional capabilities a program needs in order to implement a higher-level capability.

Figure 4 is a structure diagram of the type used in structured design. The diagram was developed from the data flow requirements specification in Figure 1. It describes the hierarchy of functions needed to implement the data flow requirements. Each box represents a functional capability. Lines between boxes denote intermodule data or control references. The small annotated arrows, or data couples, indicate flow of specific data items between functions. Different kinds of notation can be used to distinguish computational from switching data couples.

The structure diagram in Figure 4 contains function boxes for both manual and automated parts of the dating system. "Get completed questionnaires" is likely to be a manual activity; "create data records" is carried out by the software. Both kinds of functions and their structural

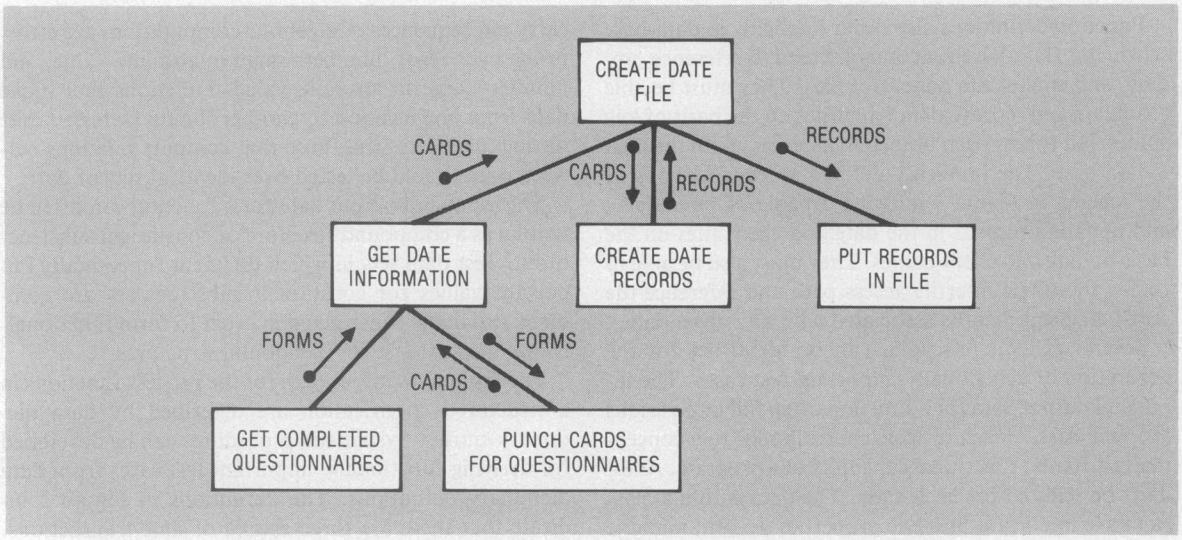


Figure 4. Structure design diagram for dating service data-base builder.

interrelation can be described by structured design diagrams.

The functions in a structured design can be divided into two broad classes: general and detailed. General functions correspond to major functional computations in a program. They often correspond to separate modules or are identified by comments that denote the sections of code that implement the functions. Detailed functions are invented during the later stages of design or in the early stages of programming. Frequently, they are not documented by program comments. They can correspond to single lines of code.

Analysis of design specifications. The general properties in Table 1 can be used for analysis of both requirements and design specifications. Table 6 contains specific design validation guidelines which were constructed by applying the general principles in Table 1 to particular features of structured design.

One of the key ideas in analyzing the consistency between design and requirements specifications is traceability. It must be possible to trace each design module or function to the process, data base, or data flow in the requirements specification that it will support. It must also be possible to trace requirements elements forward to design elements.

Table 6.
Sample completeness and consistency properties for structured designs.

PROPERTY	COMMENTS
Connection Completeness	All intermodule (or interfunction) data and control references must be modeled by connection lines in the structured design diagram.
Couple Completeness	All data sharing between modules (or functions) must be represented by data couples of the appropriate kind (i.e., by switching or computation data couples).
Data Conservation	When a data couple indicates the passing of data from one module to another, the source module must either generate the data, transform data supplied by an input couple to the source module, or pass data directly from an input couple to the source module.

Specific techniques can be constructed for analyzing the consistency of structured analysis requirements specifications with structured design specifications. *Transform analysis* can be used to examine the consistency of data flow diagrams for transformation systems with structured design diagrams. The major computational activity in a transformation system is the transformation of input data into output data. The data flow diagram for such a system consists of a linear sequence of data flow edges and process nodes. The initial part of the sequence is concerned with acquisition or input of data, the middle part with transformation, and the terminal part with output. The corresponding design diagram should consist of a root node module (or function), which is connected to an input module, a transform module, and an output module. The input module "gets" the data, the transform module transforms it, and the output module "puts" the transformed data. Data couples should indicate the passing of data between the root node modules and the subordinate modules. If the input part of the data flow diagram in the requirements specifications consists of several process nodes and data flow arcs, then the input module in the design will be data coupled to one or more modules at lower levels of design abstraction that correspond to the compound input data flow structure. The transform and output modules can also be refined at lower levels of abstraction with other design modules.

Figures 1 and 4 contain simple corresponding data flow and design structure diagrams derived from examples in DeMarco.[2] They describe the transform-oriented data-base builder part of the computerized dating system example. Transform analysis can be used to analyze the consistency of the two representations.

Transaction analysis involves a similar type of requirements/design consistency analysis for transaction-oriented systems. The data-base updating part of the

dating system that uses the date transactions described in Figure 2 is a transaction system.

Generation of test data from design specifications. The generation of test data from design specifications involves the same basic steps as test data generation from requirements specifications: identification of functions and construction of functionally important test cases.

Each of the module or function nodes in a structured design diagram identifies one or more design functions. The data couples in the diagram identify the input and output for the functions. If data dictionary entries like those in Figure 2 are used to describe the input and output data, systematic procedures can be devised for constructing functionally important input and output cases.

Program development

Validation and programs. More extensive research has been carried out on techniques for analyzing and generating test data from programs than on analyzing and generating test data from specifications. This is due to the newness of life-cycle validation and systematic specification methods.

Analysis of programs. In the literature, analysis of programs is often called "static analysis" and program testing "dynamic analysis." Computer programs can be internally analyzed and also analyzed for consistency with regard to specifications. Several methods for analyzing programs and for comparing programs with requirements and design specifications are

- subroutine interface analysis
- path flow analysis
- statement analysis
- checklist code inspection
- simulated execution, symbolic evaluation
- code/specifications consistency
  - code comments/requirements consistency
  - code comments/design consistency

Subroutine interface analysis requires the availability of the source code for the program being validated, along with the source code for all subroutines called by the program. Formal and actual parameters are compared for consistency of number, precision, and type. The consistency of such declared properties as precision and type can be checked automatically by a software validation tool.

The programmer should also check the consistency of undeclared problem-dependent parameter attributes. In scientific programs, for example, implied units such as kilograms/metre^2 or kilometers/hour might be associated with parameters. The analysis of the consistency of units of actual and formal parameters can be automated if the programmer documents this information in machine-readable comments.

The most common forms of path flow analysis depend only on the availability of source code and can be automated. Other types require the code under analysis to be documented with special kinds of information. If the information is maintained in machine-readable form, this kind of analysis can also be automated.

There are two types of flow analysis: data flow and control flow. Data flow methods examine the flow of data through a program. The most widely used data flow method involves examination of the source code for variables that are referenced before they are assigned values and those assigned values that are not referenced later.[7] Each path must be followed through the code of a program (up to repetitions of loops) in order to detect data flow anomalies.

Studies of programming errors indicate the importance of classifying all subroutine parameters as input, output, or computational. Data flow analysis can be used to ensure that (1) input variables are only referenced, never assigned values and (2) output variables are always assigned a value along some path through the program. If the input/output classification of parameters is documented in the form of machine-readable comments, tools can be built to carry out the analysis automatically.

Control flow analysis deals with sequences of control transfers. It requires searching for incorrect and unstructured constructs, such as branches into the middle of DO-loops. The use of backward-branching conditional statements for loop building is prohibited by some programming standards. Such statements can be detected with control flow analysis.

Statement analysis, like path flow analysis, can involve only source code or both source code and special types of program comments. It can be used to search for localized errors and to check the completeness of certain classes of required program comments. It does not require the analysis of data flow along program paths or across subroutine boundaries. It employs rules triggered by specific types of source code statements or program comments. Statement analysis rules are often language-dependent and can also depend on programming standards that add new syntactic restrictions to a language.

Some statement analysis rules useful for Fortran programs are listed below. They use only the program source code during the analysis.

Expression analysis:
- use of mixed-mode expressions
- equality relations involving REAL variables

Data statement analysis:
- more array elements than data values
- precision of values different from precision of variables
- too many digits in a value
- variable occurs twice in DATA statements

Array dimensions:
- value $\neq 1$ used for dimension of formal parameter array

Two types of statement analysis methods involve program comments. In the first, the comments are checked against program code for consistency. Programmers might be required, for example, to list the subroutines they think are called by their programs. If this list is included as a comment in the program, it can be automatically checked against the subroutines actually called. In the second type, the static analyzer does not

check or use the comments for correctness or consistency analysis. It checks only to see that the comments are present and complete. Suppose that programmers are required to describe the types and ranges of all input and output variables for procedures. If the description is included in the procedure as a comment, its completeness can be automatically checked.

Checklist code inspection involves a line-by-line inspection of the source code. For each statement, the relevant error classes in an error checklist are determined, and the question, "Does his statement contain an error of this type?" is asked. The method is not foolproof, and there is no guarantee that the analyst will always answer correctly. It has been found, however, to be an effective validation procedure.

The items in code inspection checklists are more specific than those in design and requirements analysis. Checklists are similar to statement analysis rules. The difference between them is that statement analysis rules involve the description of constructs whose incorrectness can be determined from the construct alone or from comparison of the construct with machine-readable comments. The errors listed in code inspection checklists describe problems that depend on information not present in the code and known only (and possibly informally) by the analyst.

Code inspection checklists are based on studies and classification of programming errors. The error classes should be causitive rather than symptomatic. Table 7 contains an error classification constructed from the results of several studies. It has proven to be generally applicable. The classes in Table 7 are not necessarily disjoint; there is a certain amount of overlap. The general error classes can be broken down into more refined error classes.

To carry out a simulated execution of a program, the code and some sample input must be available. If the sample input consists of symbols that stand for input values, as in symbolic evaluation,[8-11] it is necessary to know which path(s) are to be followed through the program. Symbolic expressions representing the expected symbolic output should also be provided.

If a program path is symbolically evaluated over symbols that stand for classes of data, the resulting output will consist of symbolic expressions describing the computations carried out by the path for classes of data rather than for single data points. This technique is particularly useful for scientific programs in which different paths or parts of paths are supposed to compute known mathematical formulae. The symbolic output from such a path or partial path can be compared directly with the formulae to verify the correctness of the computations along the path.[8]

The code for a program should be preceded by a section of comments called the program header. If the header comments that describe variable characteristics and subroutines called within the program are written in a formal machine-readable language, a software tool that checks them for completeness can be built.

Program comments also occur at intermediate points in the program text. Intermediate comments should describe the design functions used in building the program. They should identify the functions computed by various pieces of code, along with their input and output variables.

Functions that are separate in the design can be merged and overlap in the code. For example, when it is necessary to carry out several functionally separate operations on the same array or vector, program efficiency might require the operations to be carried out simultaneously, during a single pass over the data rather than sequentially in several passes.

The information contained in program comments should be carefully compared with the corresponding information in the requirements and design specifications. Table 8 lists examples of information contained in program comments and the corresponding specifications information with which it should be compared.

Generation of test data from program code. The most widely used program-based test data generation method is *branch testing*. It requires that the programmer construct a set of tests which causes all (or a given percentage of all) branches to be executed at least once.[12] The technique can be refined by requiring that all hidden branches be executed. A branch predicate consisting of a disjunct of k terms can be considered to have k hidden branches. Each hidden branch corresponds to the selection of a true value of one of the disjuncts.

There are three important situations in which branch testing is effective. In the first, an error causes a program to generate incorrect output whenever a certain branch is executed. The second occurs when an error causes a section of code to become inaccessible. In the third, errors cause unexpected branches to be traversed. The first kind of error is detected by examining program output values;

Table 7.
General error classes.

ERROR CLASS	COMMENTS
Logic Errors	Errors in the code and control constructs that determine flow of control through the program.
Computation Errors	Errors in the expressions and assignment statements used to compute output values.
Data Reference Errors	Errors in the code used to access data, such as pointer or indexing operations.
Data Declaration Errors	Errors in the descriptions of dimensions, units, etc., of variables and files.
Input/Output Errors	Errors in input source or output destination for data; errors in data conversion and formatting.
Interface Errors	Errors in transfer of control to subroutines.

Table 8.
Comments consistency analysis.

PROGRAM COMMENTS	SPECIFICATIONS INFORMATION
Program Header	Requirements Functions
Intermediate Comments Describing Program Subfunctions	General and Detailed Design Functions

the second and third are detected by looking at branch usage information.

Path testing is similar to branch testing, but requires that each logical path through a program be executed at least once during some test. In general, a program with loops can have an infinite number of paths. Even if the number of loop iterations is limited, an enormous number of different possible paths often remains—and the phrase "all logical paths" must be interpreted in some restricted sense.

Mutation testing requires that all nonequivalent mutants—programs created by small changes, or mutations, in a given program—yield output different from that of the original program for at least one test.[13] The correct operation of a program over a set of tests of this type guarantees that it contains no errors of the type introduced by mutations.

Both branch and path testing are structural methods: their tests are based on the structure of a program. Functional testing techniques for generating test data from specifications can also be used to generate test data from source code. The functions computed by a program as a whole, the general design functions corresponding to subsections of code, and important expressions that compute functionally meaningful values should all be tested over functionally important test cases. Many of the functionally important test cases derived from examination of a program's code will be the same as the functional test cases derived from its requirements and design specifications.

Program testing. There are three phases to program testing: construction of test plans, program execution, and verification of output results. Life-cycle software verification requires the construction of test plans based on requirements and design specifications and on program code. Test plans describe test data over which programs are to be executed, test procedures, and expected output.

In some situations, the code corresponding to a design or a requirements specifications function is buried deep inside a program. This makes it dificult to generate system input data that results in the testing of the code over functionally important input values. In such a case, it might be necessary to use a test harness to artificially insert intermediate values into variables at the beginning of the code section.

The third phase of program testing, verification of test results, can take different forms. The more common methods are

- analysis of output variable values
- analysis of data flow traces
- error interrupts and dynamic assertions
- runtime subroutine interface analysis
- branch and variable usage

The first method is the familiar output values check. This method can be applied both to output values from the program as a whole and to functionally important subsections. It is often reasonable to assume that the program tester is capable of verifying the correctness of a set of functional output values for any given set of functional input values. Sometimes it is also reasonable to assume that the tester can verify the correctness of value traces. The second method, tracing, involves a post mortem examination of intermediate variable values.

The third method involves the trapping of illegal values. Two kinds of illegal values can be identified. The first is associated with problem-independent illegal conditions such as underflow, overflow, and divide by zero. The second class is problem-dependent. One might, for example, expect the value of some variable to always be non-negative at some intermediate point in a program. Illegal, positive values can be detected using *dynamic assertions.*[12] These are boolean conditions on program variables that are inserted at intermediate points in the text of a program. Dynamic assertion tools are used to convert the assertions into executable code. The executable code checks the boolean condition whenever the corresponding part of the program is executed. It keeps track of and reports the occurrence of all illegal values (i.e., sets of intermediate variable values that do not satisfy the boolean condition).

The fourth method uses runtime checks on subroutine interfaces. Array bounds errors that involve formal subroutine parameters sometimes require runtime facilities for checking the consistency of array bounds in the calling and called programs.

The fifth method is associated with errors that are more easily discovered by examining records of branch traversal and variable usage than computed output

values. Branch traversal records can reveal that expected branches were not followed or that unexpected branches were followed. Variable usage records can reveal unexpected (or the expected lack of) usage of program variables.

The test plans built during each life-cycle phase should describe not only the test cases to be used, but also the methods of output verification to be employed.

Maintenance

Good specifications documents are the key to successful maintenance. The effect of a maintenance activity that results in a change to a system can be determined by identifying its domain of influence. In a structured analysis requirements specification, this is done by isolating the part of the data flow diagram affected by the change. This partial flow diagram can be used to construct a plan for reanalysis or retest. Well structured requirements specifications minimize the reanalysis and retest phases.

The domain of influence in a structured design specification is identified by tracing the effect of a change to the design modules associated with the change. The affected modules form a hierarchy of abstraction. The most abstract is the root module of the structured design diagram. The next most abstract is one of the subordinate modules of the root module, and so on. The highest-level module whose subordinate module(s) are altered by the change identifies the necessary reanalysis and retesting.

Changes to a system can be initially described in reference to requirements, design, or code. Changes to a design must be traced to requirements specifications; changes to code must be traced to both design and requirements specifications.

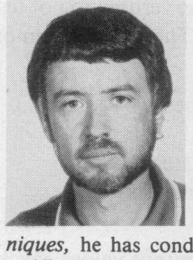
The above descriptions of analysis and test data generations techniques are incomplete; the present purpose is only to sketch the basic features of an integrated approach to life-cycle validation. Additional checklist rules, function identification techniques, and test data selection methods can be defined for each phase of the life cycle.[13,14] ■

Acknowledgments

Much of the research in this article was supported by the US National Bureau of Standards and by the National Sciences and Engineering Research Council of Canada.

References

1. B. Boehm, "Some Experience with Automated Aids to the Design of Large Scale Reliable Software," *IEEE Trans. Software Eng.*, Vol. SE-1, No. 1, Jan. 1975, pp. 125-133.
2. T. DeMarco, *Structured Analysis and System Specification,* Yourdon, Inc., New York, 1978.
3. M. Jackson, *Principles of Program Design,* Academic Press, London, 1975.
4. W. E. Howden, "Functional Testing and Design Abstractions," *J. Systems and Software,* Vol. 1, pp. 307-313.
5. E. Yourdon and L. Constantine, *Structured Design,* Prentice-Hall, Englewood Cliffs, N.J., 1979.
6. L. J. Osterweil and L. D. Fosdick, "DAVE—A Validation Error Detection and Documentation System for FORTRAN Programs," *Software—Practice & Experience,* Vol. 6, No. 4, Oct.-Dec. 1976, pp. 473-486.
7. W. E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," *IEEE Trans. Software Eng.,* Vol. SE-3, No. 4, July 1977, pp. 266-278.
8. J. C. King, "Symbolic Execution and Program Testing," *Comm. ACM,* Vol. 19, No. 7, July 1976, pp. 385-394.
9. L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Trans. Software Eng.,* Vol. SE-2, No. 3, Sept. 1976, pp. 215-222.
10. R. S. Boyer, B. Elspas, and K. Levitt, "SELECT—A Formal System for Testing and Debugging Programs by Symbolic Execution," *Proc. Int'l Conf. Reliable Software,* 1975, pp. 234-245.
11. L. G. Stucki, "New Directions in Automated Tools for Improving Software Quality," in *Current Trends in Programming Methodology, Vol. 2,* R. T. Yeh, ed., Prentice-Hall, Englewood Cliffs, N.J., 1977, pp. 80-111.
12. T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "The Design of a Prototype Mutation System for Program Testing," *AFIPS Conf. Proc.,* Vol. 47, 1978 NCC, pp. 623-627.
13. G. Myers, *The Art of Software Testing,* Wiley-Interscience, New York, 1979.
14. W. Howden, *Validation of Scientific Programs,* Institute for Computer Science and Technology, National Bureau of Standards, Washington, DC, 1980.



William E. Howden is currently an associate professor of computer science at the University of California at San Diego. He has previously worked for Atomic Energy of Canada, McDonnell Douglas Astronautics, and the University of Victoria. He has carried out extensive research projects in both the practice and theory of program testing. Coauthor of the IEEE tutorial *Software Testing and Validation Techniques,* he has conducted professional seminars on software verification in the US and abroad. His published papers deal with a wide variety of topics in program testing.

Howden received a PhD in computer science from the University of California at Irvine in 1973. He is a distinguished visitor of the IEEE Computer Society and a member of the executive board of the society's Technical Committee on Software Engineering.