Automated Software Quality Assurance

HARRY M. SNEED, MEMBER, IEEE, AND ANDRÁS MÉREY

Abstract—This paper describes a family of tools which not only supports software development, but also assures the quality of each software product from the requirements definition to the integrated system. It is based upon an explicit definition of the design objectives and includes specification verification, design evaluation, static program analysis, dynamic program analysis, integration test auditing, and configuration management.

Index Terms—Dynamic analysis, review techniques, software metrics, software qualtiy assurance, static analysis.

I. Introduction

QUALITY assurance plays a vital role in the software life cycle process. After each development phase, it is necessary to confirm the result of that phase before proceeding to the next. Manually this can be an expensive process, one that is difficult to justify economically, not to mention the fact that it is difficult to find qualified persons willing to carry it out. The only real way to make quality assurance both systematic and economically feasible is to automate it [1].

As yet there have, however, been few attempts to integrate and automate the whole validation and verification process. This is due to the fact that in order to be automatically processed, the various software products must be in a machine processable form, and this presupposes some type of formal language for each semantic level of software development. Until now, the only machine processable software has been the programs themselves. But programs are only one of at least seven different software products [2]. These are

- requirements documentation,
- specification documentation,
- design documentation,
- programs,
- program documentation,
- test documentation, and
- user documentations (see Fig. 1).

A complete quality assurance plan is one that encompasses all of the above named products. Thus, an automated quality control system should be able to verify and validate each of these products [3].

The SOFTING Software Engineering Environment is intended to be such a system. It is a family of horizontally and vertically integrated tools with a common development database in which all of the software products are represented. Like ISDOS and REVS, it not only supports the creation and maintenance of the software products, it also ensures their quality through a series of verification, validation, and evaluation procedures.

Manuscript received November 8, 1982.

The authors are with the Software Engineering Service, 8014 Neubiberg, West Germany.

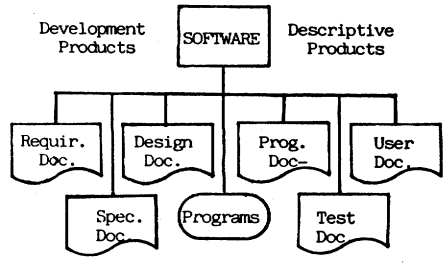


Fig. 1.

These procedures for each phase of the software development cycle are the subject of the following discourse.

II. Quality Requirement Definition

Quality is relative. In order for quality to be measured it must first be defined. This is a task of the project manager in the requirements phase.

The SOFTMAN project management sytem supports this effort in that it requires the project manager not only to require quality but to specify the design objectives as well. The four main objectives are

- quantity,
- quality,
- time, and
- costs.

Quantity is understood as the scope of a system in terms of the proportion of possible entities included and relationships implemented, i.e., what the system actually does as compared to what it should have done.

Quality is, on the other hand, the sum of the characteristics of a system, i.e., how the system is actually constructed and how it performs as opposed to how it should have been built and how it should perform.

Time and *costs* are directly measurable in terms of calendar months passed and money expended.

The above named design objectives are contradictory to one another. Quality can only be increased by reducing quantity or by increasing time and costs. Quantity can only be increased by sacrificing quality or by increasing time and costs. Time and costs are also tradeoffs as demonstrated by Brooks [4]. Adding personnel to reduce time increases costs and vice versa.

The relationship of these conflicting goals is represented by the so called devil's quadrat which is, in effect, a quadratic equation for computing tradeoffs between four opposing variables when assuming a fixed capacity (see Fig. 2).

The fixed capacity of the productivity of the developing organization is determined by a combination of productivity factors such as

0098-5589/85/0900-0909$01.00

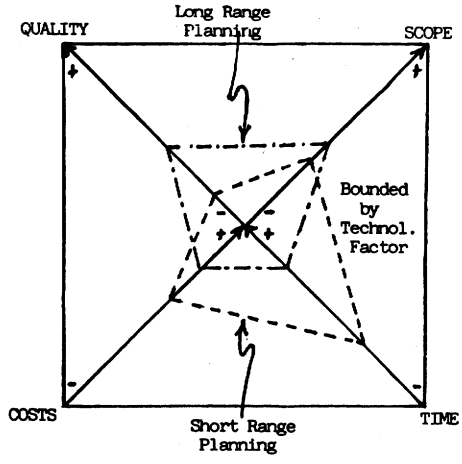


Fig. 2.

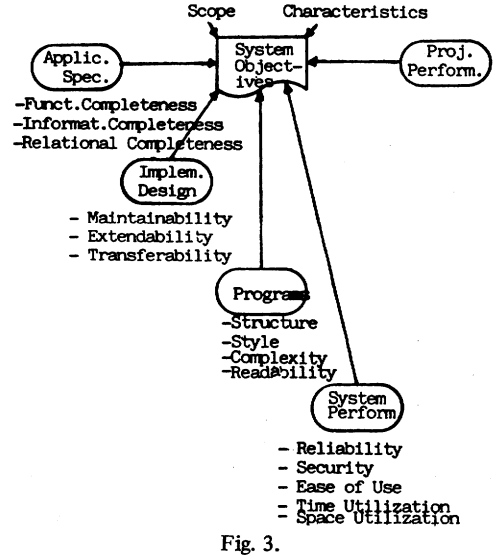


Fig. 3.

- hardware facilities,
- software aides,
- developer experience, and
- application complexity.

Boehm has defined these factors in his COCOMO Model [5].

SOFTMAN requires the manager of a project to first calculate his productivity factor and to weigh the basic design objectives. For this purpose, he has a fixed number of points to distribute as practiced by the MECCA Method [6]. Once this is done, he must further distribute the points already assigned to quantity and quality since each of these main goals contains subgoals.

Quantity encompasses the subgoals

- functional completeness,
- informational completeness, and
- relational completeness.

Functional completeness is a measure of the proportion of processes within an application area which are automated by the new system.

Informational completeness is a measure of the proportion of data communications and data storages, i.e., the objects within an application area which are actually reproduced or stored by the new system.

Relational completeness is measure of the proportion of relevant relationships between objects and processes in the application which are actually implemented in the new system.

Quality contains the subgoals

–extendability,
–transferability,
–maintainability,
–reliability,
–security,
–efficiency, and
–usability.

Extendability is the measure of the amount of effort required in order to change or enhance a system relative to the amount of effort needed to originally develop it.

Transferability is a measure of the amount of effort required in order to transfer the system from one environment to another relative to the original development effort.

Maintainability is a measure of the effort to keep a system operational relative to the effort required to develop it.

Reliability is a measure of the proportion of cases, i.e., time, in which the system operates correctly relative to the total number of cases. Correctness is, in turn, a question of consistency between actual behavior and specified behavior.

Security is a measure of the degree to which a system secures its data and recovers from failures without human intervention.

Efficiency is a measure of the execution time and storage space actually necessary to operate the system as opposed to that planned.

Usability is, in turn, a measure of user satisfaction with the new system as compared to the old system, whereby the old system may have been a manual one.

These design objectives are indispensable for the succeeding quality assurance procedures. The objectives of scope such as functional, informational, and relational completeness are required to measure the completeness of the application specification. The technical objectives such as maintainability, transferability, extendability, and usability are needed to measure the adequacy of the implementation design. The operational objectives like reliability, security, and efficiency are essential to judge the quality of the implementation. Finally, the managerial objectives–time and costs–are necessary to control the development process.

SOFTMAN is designed to force the project manager to commit himself to certain measurable objectives in the requirements definition phase. From this point on, all the properties of the system under development are assessed in accordance with the management prerogatives (see Fig. 3).

III. Specification Verification

The first software product which can be subjected to an automated quality control is the requirements specification, i.e., application design. This preliminary product can be analyzed automatically in regard to its

- completeness,
- consistency, and
- feasibility [7].

Completeness is defined in the context of the specification as the proportion of objects, processes, and relationships plan-

ned which have actually been fully specified. Consistency is the lack of contradiction between various parts of the specification. Feasibility is the probability that the system specified can be implemented within the time and cost restraints stated in the requirements phase.

The purpose of such an analysis is to prevent an unripened prematured concept from being implemented. An automatic analysis at this time is a prerequisite to a manual design review since it is extremely difficult to review the contents of an application design which is incomplete, contradictionary, and infeasible. The chances of being understood by the potential user is also greatly reduced when it is formally incorrect. The design reviewees will only be distracted from assessing the contents of the proposed solution by the many formal errors in its definition. For these and other reasons, the formal verification of an application design should be an important part of any quality assurance program.

In the SOFTING software production environment the specification verification is performed by the SOFSPEC system which also supports the definition, storage, and documentation of the requirements specification.

The application design is represented as an entity/relationship model as proposed by Chen [8] and stored in a linked list data base in accordance with the network data model [9]. The elements of the model are the

- objects,
- processes,
- data,
- functions,
- forms, and
- predicates

of the application design.

The specification is considered *complete* when all elements, attributes, and relationships relevant to a particular application have been identified and defined. A tool such as SOFSPEC can of course not know if all the relevant entities have been defined. It can only compare the objects, processes, and relations defined in the final specification to those given in the initial requirements definition. It can, however, automatically check if all the required attributes of an entity, for instance the length, type, description, and domain of a data item have been submitted. This applies to all the elements of the system. Furthermore, it can detect missing relationships between elements, such as objects with no links to processes, objects with no data, processes with no functions, conditional functions with no predicates, and transformation functions with no related data.

The specification is *consistent* when there are no contradictions between the defined relationships. Thus when one process is defined as a predecessor to another, then the other must be defined as its successor. When a process generates an object, then that object must be generated by it. A communicational entity must be represented by a form and an informational entity must be defined by a data tree. In each data tree, at least one data item must be designated as a search key. Also, each repetitive data item must have an occurence greater than one and each repetitive function must be defined as an iteration. Decision rules may not be ambiguous and the representation of data in forms must correspond to the data types. These are only some of the many consistency checks carried out by the SOFSPEC system. In effect, the consistency verification is similar to an extensive syntax analysis of a formal language.

An application specification is considered to be *feasible* when the size and complexity of the system specified is not greater than that which can be implemented within the time and cost restraints prescribed in the requirements definition. The size of an application is measured here in terms of the number of different elements defined in the specification. The complexity is a factor of the number of distinct relationships between all the entities contained therein. Should these numbers exceed the maximum numbers stated in the project definition, then it is obvious that the system will cost more than what management is willing to pay. If there is a limited number of processes and objects with a high degree of interrelationships, it is also a sign that the inplementation work can not be easily distributed. This means an increase in calendar time.

Besides these three automatic controls, SOFSPEC aslo provides an extensive documentation of the system specification both at the level of objects and processes as well as at the level of data and functions. With the help of this documentation it is possible for the user to review the accuracy of the application design before it is implemented.

IV. Design Evaluation

The second quality assurance point in the computer aided software production process with SOFTING is the design evaluation. The implementation design is a formal definition of the data and program organization. This definition is represented by the system SOFTCON in a design database which, like the specification database, is based on the entity/relationship model.

The data design consists of five elements and five relations. The data elements are:

- the file or logical database descriptions,
- the data communication interface descriptions,
- the data item descriptions,
- the data item codes, and
- the data input/output assertions.

The data relations are:

- the assignment of data capsules to files of logical databases, i.e., the file or database structures;
- the assignment of data items to data capsules;
- the assignment of codes to data items;
- the assignment of input/output assertions to data items, and
- the relationship of data capsules to modules, i.e., files or logical databases to programs.

The program design also consists of five elements and five relations. The program elements are:

- the process or job descriptions,
- the program descriptions,
- the module interface descriptions,
- the module control flow descriptions, and
- the testdriver/stub procedures.

The program relations are:

- the assignment of programs to processes or jobs,

- the assignment of modules to programs and to other modules,
- the assignment of functions and predicates, i.e., test paths, to modules,
- the assignment of assertions to test paths, and
- the relationship of modules to data capsules, i.e., programs to files or logical databases.

It is important to note that the data and program design are symmetrical and that they complement one another. This facilitates automated design validation.

The design validation in SOFTCON consists of two parts: a content verification and a technical assessment.

The *content verification* is a consistency check against the application design or specification. Using the specification database as a basis the design database is processed to verify that each

- object,
- process,
- data,
- function,
- form, and
- predicate

in the application specification has been properly transferred to the implementation design. The objects must be assigned to files or databases, the forms to data communication interfaces, the processes to programs and the data to data capsules, while functions and predicates are assigned to modules. In addition, the relationships which exist in the technical design are checked for consistency with one another. For instance, a module interface must agree with the module's use of data. This check insures that the design is formally correct.

The *technical assessment* is a measure of design techniques such as

- modularity,
- generality,
- portability,
- redundancy,
- integrity,
- complexity,
- time utilization, and
- space utilization.

The goal of this assessment is to examine to what extent the technical design objectives—extendability, transferability, maintainability, security, usability, efficiency, and reliability—have been met. If the system is supposed to be maintainable, then the design should be modular and noncomplex. If the system is supposed to be extendable, then the design should be modular and general. If the system is supposed to be transferable, then the design should be modular and portable. If the system is supposed to be secure, then the design should have redundancy and integrity. If the system is supposed to be efficient, then the design should be economic in regard to either time, space, or both. Finally, if the system is to be reliable, then it should be secure and noncomplex.

Modularity is the mean of five related ratios (M) where

$$M_1 = \frac{\text{modules}}{\text{application functions}}$$

$$M_2 = \frac{\text{data capsules}}{\text{application oriented data items}}$$

$$M_3 = \frac{\text{modules}}{\text{module calls}}$$

$$M_4 = \frac{\text{data capsules}}{\text{data flows}}$$

$$M_5 = \frac{\text{modules}}{\text{module interfaces}}.$$

Data capsules are logical access units which may be records, tables, or parameter lists. Data flows are defined here as the transfer of data from one capsule in the system to another. Module interfaces are the number of data flows into and out of the modules. This includes parameter passing, input/output operations, message transmissions, and database accesses.

Generality is the mean of two ratios (G) where

$$G_1 = \frac{\text{applicaton independent modules}}{\text{modules}}$$

$$G_2 = \frac{\text{application independent data capsules}}{\text{data capsules}}.$$

Application independent modules are those which contain no application specific functions. Application independent data capsules contain no application specific data.

Portability is the mean of the ratios (P) where

$$P_1 = \frac{\text{environment independent modules}}{\text{modules}}$$

$$P_2 = \frac{\text{environment independent data capsules}}{\text{data capsules}}.$$

Environment independent modules are those which contain no references to a particular operating, database or data communication system. Environment independent data capsules contain no data connecting them to a particular database or data communication system [10].

Redundancy is the mean of the three ratios (R) where

$$R_1 = \frac{\text{repeatable modules}}{\text{modules}}$$

$$R_2 = \frac{\text{reproducable data capsules}}{\text{capsules}}$$

$$R_3 = \frac{\text{logged transactions}}{\text{transactions}}.$$

Repeatable modules are those which can be restarted with no side effects. Reproducable data capsules are those that can be restored in case of loss or contamination. Logged transactions are those which are protocolled for restarting if necessary.

Integrity is the mean of two ratios (I) where

$$I_1 = \frac{\text{edited system input data items}}{\text{system input data items}}$$

$$I_2 = \frac{\text{edited system output data items}}{\text{system output data items}}.$$

System input data are all data items received via an external interface, i.e., form. Edited input data are those whose plausibility is checked before they are processed. System output data are, on the other hand, all data items transmitted by the system to its environment via an external interface. Edited output data are those whose validity is confirmed before they are transmitted.

Complexity is measured in the design as the mean of the predicate, predicate variables, function, and data element ratios (C) where

$$C_1 = \frac{\text{predicates}}{\text{predicate variables}}$$

$$C_2 = \frac{\text{functions}}{\text{predicates}}$$

$$C_3 = \frac{\text{variables} - \text{predicate variables}}{\text{variables}}$$

$$C_4 = \frac{\text{functions}}{\text{variables}}.$$

The predicates are the selection and repetition conditions of the module pseudocode, i.e., the nodes of the flow graph. The predicate variables are the operands in these conditions. The functions are the edges of the module flow graph. The variables are the input and output operands.

The utilization can be measured at design time only in terms of module calls and data accesses per transaction. Time efficiency can thus be stated as the mean of the two ratios (T) where

$$T_1 = \frac{\text{transactions}}{\text{data accesses per transaction} \times \text{transactions}}$$

$$T_2 = \frac{\text{transactions}}{\text{module calls per transaction} \times \text{transactions}}.$$

Space utilization is measured at design time in terms of the required storage compared to the number of data items. The ratio

$$S = \frac{\text{data items}}{\text{average capsule length} \times \text{number of data capsules}}$$

In all cases the result of the attribute measurement is a fractional number from 0 to 1 with 0 being the lowest and 1 the highest grade. Each of these detailed quality characteristics can then be weighted in accordance with the quality goals set forth in the requirements definition to assess the conformity of the implementation design with the design objectives [11].

V. Static Program Analysis

The static analysis of the finalized programs is the third checkpoint in the SOFTING development process. It is carried out by the system *SOFTDOC* with the dual purpose of controlling the program quality and documenting the programs.

SOFTDOC analyzes programs at three different levels, at the module, program, and system level. A module is defined as a compilation unit such as a PL/1 procedure, or a Cobol program. A program is a run unit with a main module and n submodules. A system is, finally, the full set of programs making up an application. The analysis is conducted bottom-up starting with the finished modules and processing up to the system level [12].

The SOFTDOC analysis process is, in effect, a retranslation of the programs into a design. From the analysis of the data structures and data flow in the programs, SOFTDOC generates the

- data item description tables,
- data item code tables,
- data capsule structures, and
- file or data base structures.

From the analysis of the structure and control flow of the programs SOFTDOC generates the

- interface description tables,
- pseudocode tables,
- program structures, and
- data flows.

In addition, it also produces a cross reference table of the relations between data and modules. This design data base is equivalent to that generated by SOFTCON from the application specification. In fact, the elements of the original data base are updated by the program analysis.

There are three ways in which SOFTDOC controls the quality of programs. First, it *audits the code* against a set of coding standards intended to enforce modular and structured programming as well as to encourage a good programming style. Violations of the conventions are reported as code deficiencies.

Secondly, it *measures the complexity* of the programs and the degree of program interdependence. For the measurement of module control flow complexity, the decision outcomes, i.e., segments, the forward paths, the internal subroutines, and the nesting levels are counted. For the measurement of module data flow complexity, the number of data variables used, the number of predicate variables and the total number of data references are computed [13].

Intermodule dependence is measured on the basis of the number of modules and module invocations as well as the number of data items passed between modules. Since all parameter strings are stored in an interface table, SOFTDOC is also able to compare the parameters in the calling module to those in the called module for formal correctness.

Interprogram dependence is measured in terms of the number of modules used in two or more programs, the number of data capsules used in two or more programs, and the average number of files, logical databases, and data communication interfaces per program in the system.

The complexity and interdependence measurements derived by SOFTDOC from the programs are used in the same manner as those obtained from the design. They are compared to the weighted design objectives to determine the degree of correspondence between the programs and the requirements.

The third quality control performed by SOFTDOC is to compare the actual data structures, data flows, program structures, and control flows to those defined in the original design database. Wherever the content of the program deviates from the content of the design, an implementation deficiency is reported, for instance, when a module is missing arguments or results specified in the module design.

Thus, the automated static analysis by SOFTDOC points out three distinct types of program deficiencies:

- coding deficiencies,
- structural deficiencies, and
- implementation deficiencies.

This automated analysis can be enhanced by a manual code inspection. For this purpose, SOFTDOC generate a series of documents at the module, program, and system level.

At the module level it documents the

- input/output data of each internal section or procedure,
- data structures,
- data codes,
- data references,
- module structures,
- module interfaces,
- module control flow, and
- module test paths.

At the program level it documents the

- input/output data capsules for each module,
- intermodular data flow,
- file structures,
- program structure, and
- module calling hierarchy.

Finally, at the system level it documents the

- input/output files, databases, and data communications of each program,
- interprogram data flow,
- data dictionary,
- process structures, and
- module cross references.

With the aid of these documents the manual code inspection can be greatly enhanced [14].

VI. Dynamic Program Analysis

The fourth step in SOFTING's automated product assurance process is the dynamic analysis of the programs. Dynamic analysis is performed by testing the modules in a simulated test environment against the assertions postulated in the specification and by monitoring their behavior. This is the task of the SOFTEST system.

SOFTEST consists of several integrated testing tools, among them

- an instrumentor,
- a test bed generator,
- a test data generator,
- a test monitor,
- an assertion checker,
- a data flow analyzer, and
- a control flow analyzer.

These tools are supported by a test database consisting of

- assertion procedures to simulate each data interface,
- driver and stub procedures,
- data flow coverage tables,
- control flow coverage tables, and
- the appropriate symbol tables.

The first step in the program testing process is to generate test data tables from the input/output assertion procedures supplied by the system SOFTCON. SOFTEST creates such a table for each data interface, module stub and module driver. These tables go into the test database to be used at test time.

The second step is to instrument the programs at the branch level, that is, at each decision outcome, for the purpose of measuring test coverage and to instrument the input/output operations and module calls for the purpose of simulating them [15].

The third step is the generation of a testbed based on the data symbol and module interface tables created by SOFTDOC through the static analysis of the program. The testbed is constructed to assign the storage areas needed by the unit under test, to invoke them and to intercept all stub calls and references to files, databases, or to data communication interfaces [16].

The fourth step is the actual execution of the test. The modules are assigned their common data and parameters by the test driver which calls the appropriate assertion procedure. After having been invoked by the driver, the unit under test is then interrupted at each stub call and input/output operation. The test monitor first interprets the output assertions to validate the output values. Assertion violations are reported in a test log. The test monitor then interprets the input assertions to assign the next series of input values before restarting the test object. This handling of assertion procedures in an interpretive manner denotes a significant departure from the conventional mode of compiling assertions into the modules [17].

The dynamic behavior of the test object is measured by monitoring the data flow as well as the control flow. Each path traversed by a test case is stored in a test path file. Each branch execution is registered in the test coverage file by a trace routine. The data flow is traced at each interrupt point by comparing the state of the current data domain to the state of the last data domain and by storing all values which have been altered since the last interruption. Thereafter, the contents of the present data area become the last domain.

The fifth and final step in the program test process is the documentation of the test results. In all, SOFTEST provides five test reports:

- a log of assertion violations,
- a report of test path traversed,
- a report of data flow incurred,
- a report on the branch coverage, and
- a report on the data input coverage.

These reports assist quality assurance by pointing out where the actual behavior of a program deviates from the expected behavior, both in regard to its result and in regard to its flow. Besides this, the coverage reports can be used to assess the degree to which the programs have been tested. The most important feature of SOFTEST is, however, the fact that the programs are tested against assertions written in an assertion language at specification time with SOFSPEC and enhanced at design time by SOFTCON. This ensures the independence of the test from the implementation [18] (see Fig. 4).

VII. Systemtest Auditing

The final activity of the software developers is to integrate and test all of the system components before turning the system over to the user. This is also the last chance to control the

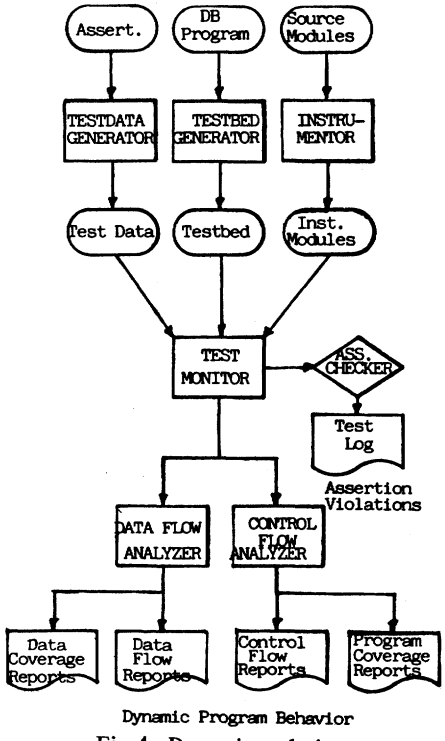


Fig. 4. Dynamic analysis.

quality of the software before the user does. It is, therefore, imperative that the integration test be audited. The objective of the audit is to ensure that the system in its entirety has been adequately tested.

A software system is more than just the sum of all its component parts such as modules and data capsules. It is the sum of all the parts and all effects produced through the collaboration of the parts [19]. In addition, the target environment, both hardware and software, must be considered a part of the test. In order to confirm the behavior of a system, then that system must be tested within its target environment for all relevant cases, including stress testing, security testing, exception testing, etc. for an adequate period of time. What is relevant and adequate can only be determined by independent test auditors supported by automated auditing tools [20].

In the integration test, the programs are executed in conjunction with their operating environment, i.e., database, data communication, and operating system. The measure of their integration is a question of the degree to which all interfaces have been tested. Thus, it is necessary to know how many DB/DC connections, input/output assertions, module invocations and exception conditions there are in the system, and what percentage of these have been tested with each possible option.

The SOFTINT system is designed to instrument each input/output operation, DB/DC function, and module invocation in order to record the degree of interface coverage. To this end, it produces two reports:

- one on the CALL coverage
- one on the I/O coverage.

as well as a system trace report. These reports aid the test auditors in determining the extend to which a system has been tested.

Besides that, SOFTINT supports integration testing by generating files and databases from the same input assertions as used by SOFTEST. Here a record or segment is created for each relevant combination of data values for each file or logical database. The generated files are then used to test the system. After the test is completed, SOFTINT compares the after images of the record with the before images which it had created as well as with the output assertions. Thereby, it detects not only what data have been changed, but also what output data do not coincide with their predicate domain. Both data mutations and assertion violations are reported for the system audit.

Finally, SOFTINT monitors all data communications between the system and its environment and reports them in a symbolic form so they can be easily interpreted by the system auditors.

VIII. Project Monitoring

No software quality control environment can be complete without a feedback loop to the original system requirement. This task is performed by the SOFTMAN system which not only keeps account of the man days, computer resources, and calendar time expended, but also compares the status of the system represented in the specification, design, and test databases with the functional and technical objectives stated in the requirements. For instance, it measures to what extent the system is really maintainable, transferrable, extendable, and reliable by matching the metrics gathered from the static program analysis against the technical objectives. It also measures the extent to which a system has been tested by evaluating the coverage data produced by SOFTEST and SOFTINT. Thus there is a feedback from the actual properties of the software system under development to those features postulated by management. In each case the extent of deviation is reported in percentage points. This gives management the opportunity to take corrective action before it is too late [21].

IX. Summary

This has been a brief summary of the quality assurance aspests of the SOFTING Software Engineering System consisting of seven integrated subsystems, one for each phase of the development cycle. The system is presently being developed by the two Hungarian national Software Institutes–SZKI and SZAMALK. Five of the seven subsystems have been completed and are in operation in West Germany. The two others are scheduled for release by 1986. The system is implemented to run under the operating systems MVS and VM on all IBM and IBM compatible computers. With the completion of the system in 1986, it is hoped that the major portion of the software quality assurance effort will be either partially or fully automated. This could bring about a significant reduction in the cost of quality assurance, while at the same time improving the quality of the quality assurance itself. The experience with the completed systems has been rather discouraging. The programming community, at least that in Germany, is a long way from Software Engineering. Neither the programmers nor their managers really understand the essence of software quality. And one thing has become clear. There is no substitute for

education. Software tools can only be as good as the people who are using them.

References

[1] Ramamoorthy, Dong, Ganesch, Jen, and Tsai, "Techniques in software quality assurance," in *Proc. ACM Congress Software Quality Assurance.* Stuttgart, West Germany: Teubner-Verlag, 1982.

[2] H. D. Mills, "Principles of software engineering," *IBM Syst. J.*, vol. 19, no. 4, 1980.

[3] Bryan, Siegel, and Whiteleather, "Auditing throughout the software life cycle" in *IEEE Comput. Mag.*, Mar. 1982.

[4] F. Brooks, *The Mythical Man-Month, Essays on Software Engineering.* Reading, MA: Addision-Wesley, 1975.

[5] B. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[6] T. Gilb, *Software-Metrics.* Stockholm, Sweden: 1976.

[7] W. Howden, "*Life cycle software validation*," *IEEE Comput. Mag.*, Feb. 1982.

[8] P. Chen, *Entity/Relationship to System Analysis and Design.* Ansterdam, The Netherlands: North-Holland, 1980.

[9] C. Date, *An Introduction to Database Systems.* Reading, MA: Addison-Wesley, 1977.

[10] Boehm and Brown, *Characteristics of Software Quality*, (TRW Series of Software-Technology 1). Amsterdam, The Netherlands: North-Holland, 1978.

[11] S. Mohanty, "Models and measurements for quality assessment of software," *ACM Comput. Surveys*, vol. 11, no. 3, 1979.

[12] Chen, Huang, and Ramamoorthy, "Automated techniques for static structural validation of programs, in *Proc. Compsac 77*, IEEE, Chicago, IL, 1977.

[13] T. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, 1976.

[14] M. Fagan, "Design and code inspection to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, 1976.

[15] J. C. Huang, "An approach to program testing," *ACM Comput. Surveys*, Sept. 1975.

[16] M. Majoros, "SOFTEST–Model of a commercial program testsystem," in *Proc ACM Congress on Software Quality Assurance.* Stuttgart, West Germany: Teubner-Verlag, 1982.

[17] C. Ramamoorthy, "Application of methodology for the validation of process control software," *IEEE Trans. Software Eng.*, vol. SE-7, 1981.

[18] W. Howden "Theoretical and empirical studies of program testing," *IEEE Trans. Software Eng.*, SE-4, 1978.

[19] H. Sneed, *Software-Entwicklungsmethodik.* Köln, West Germany: Rudolf Müller Verlag, 1980.

[20] G. Myers, *The Art of Software Testing.* New York: Wiley, 1979.

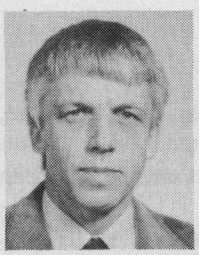
[21] Bersoff, Henderson, and Siegel, *Software Configuration Management.* Prentice-Hall, 1980.

[22] C.-IC. Cho, *An Introduction to Software Quality Control.* New York: Wiley, 1980.

Harry M. Sneed (M'78) received the Master's degree in public administration from the University of Maryland, College Park, in 1969.

He has worked as a Systems Analyst for the U.S. Department of the Navy, Washington, DC, as a Programmer/Analyst for the Volkswagen Foundation, Hanover, Germany, and as a Systems Programmer for Siemens A.G., Munich, West Germany. Since 1978 he has been Director of the Software Engineering Service, Neubiberg, West Germany.

Mr. Sneed is a member of the Association for Computing Machinery and the IEEE Computer Society.



András Mérey received the M.Sc. degree in electronics and the Ph.D. degree in microwave engineering from the Technical University of Budapest, Budapest, Hungary, in 1967 and 1971, respectively.

He is a Senior Research Fellow with the Computer Education and Information Center of Budapest. His research interests include compilers, static analyzers, and software development support environments. Since 1980 he has been a consultant for several German user firms and has participated in the development of a software production enviroment for the Software Engineering Service, Munich, West Germany.

Dr. Mérey is a member of the John von Neumann Society, the Hungarian equivalent of the Association for Computing Machinery.