RELIABILITY AND INTEGRITY OF LARGE COMPUTER PROGRAMS

by

C. V. Ramamoorthy, R. C. Cheung and K. H. Kim

Memorandum No. ERL-M430

12 March 1974

RELIABILITY AND INTEGRITY OF LARGE COMPUTER PROGRAMS

by

C.V. Ramamoorthy, R.C. Cheung and K.H. Kim

Memorandum No. ERL-M430

12 March 1974

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

RELIABILITY AND INTEGRITY OF LARGE COMPUTER PROGRAMS

C.V. Ramamoorthy, R.C. Cheung and K.H. Kim

University of California, Berkeley
Department of Electrical Engineering and Computer Sciences
Computer Science Division
Electronics Research Laboratory

1. Introduction

1.1 Cost of software

It was not too long ago that programming was generally considered as an art. In these past few years, the emergence of the term 'software engineering' indicated a major change of public opinion. Programming is not only considered as a science but also as a branch of engineering where disciplines can be enforced. This is due partly to more understanding of the 'art' of programming and partly to the strong pressure of economics. There was a time when hardware was king and every effort possible was spent in improving the utilization of the hardware of the computer. However, the rapidly decreasing cost of the computer itself and the continuously rising salary of the human programmers have compelled us to focus our attention on improving the efficiency of software development. For example, software occupied only about 25% of the United States Air Force budget for electronic data processing in 1960 (75% for hardware) while in 1973 software occupies about 80% of the USAF budget for EDP. The cost of software is still rising continuously in a linear fashion. This trend is expected to continue and the lopsidedness of the software-hardware cost ratio is probably characteristic of other organizations too.

Software has become big business in the United States. For the United States Air Force, an annual expenditure of between $1 billion and $1.5 billion has been spent on software for the fiscal year of 1972. This amounted to about 4 or 5% of the total Air Force budget. [Boe 73] At present, overall software costs in the United States are probably over $10 billion every year, over 1% of the gross national product. [Boe 73].

1.2 Problems of software

Our past experiences with software development have been depressing. Most of the software development projects are unsuccessful in terms of specification, time and cost. The final software product delivered is often unresponsive to the actual needs of the organization it was developed for. The users are promised one thing but end up with another. In many cases, a significant portion (up to 67%) has to be rewritten, after the system is delivered, in order to meet the operational needs

This research was sponsored by the Office of Naval Research Contract N00014-69-A-0200-1064.

of the users. Delay in delivery is commonplace while gross underestimation of the cost by a factor of four is not unusual. For example, the IBM OS/360 was delivered one year behind schedule and was estimated to cost more than 200 million dollars. [Ale 69].

Big as these direct costs of software may be, the indirect costs due to delays and errors are even greater. Software is usually on the critical path in the overall system development so that any delay in software delivery will directly upset the schedule of the whole system, which is extremely expensive. Moreover, the management can do very little to speed up the software development. Adding more programmers to a late project simply makes it later. To scrimp the testing, integration, or documentation procedures cost much more in the long run. Generally, the simple solution adopted is to eliminate all expandable capabilities, making the system unappealing to the user. This is especially true for many real-time systems.

Not only is the software always late and expensive, the final delivered product is also very unreliable. Much software are released with thousands of bugs still in it. Each new release of the OS/360 contains roughly 1000 new software errors. [Boe 73]. Even after the program is considered to be thoroughly tested, there were 18 discrepancies found in the software during the 10-day flight of Apollo 14. [Boe 73]. This becomes more scary when we consider the complexity of the programs for national defense and air traffic control.

1.3 Reduction of software cost

1.3.1 Software-oriented system design

From the discussion above we can see that cost and reliability are the two major causes of concern about software. In many projects, especially real-time systems, the software effort has to wait until the hardware is procured, or at least until the selection is made. Then the programs are written under the hardware constraints. This procedure has several disadvantages. The time spent on hardware procurement pushes software farther out onto the critical path. Any delay in software delivery will incur an unaccountable amount of indirect cost to the whole system. Besides, the selection of hardware is made without much consideration to the software development. A typical study of the extent to which hardware constraints affect software productivity is shown in Figure 1. [Wil 70]. We can see that as we approach 85% utilization of hardware speed and memory capacity, the software cost rises abruptly. The hardware constraints may drastically increase the cost and time for software development. With the decreasing cost of hardware and rising cost of software we have to avoid this unnecessary saturation point. We should make the hardware selection after we understand sufficiently well the requirements of software. We would rather acquire a computer with 50% to 100% extra capacity than to risk having a computer too "small" for our purpose. Whenever hardware constraints affect software development, the cheaper hardware should be traded off to save on the more expensive software. In order to get software off the

critical path, we have to initiate software development earlier in the system development cycle. The software should be specified first and a simulator or micro-programmed computer can be used to support the software development. After we have established a solid basis in software development and have enough knowledge about its requirements, we will then give the detailed design specification of the hardware required to support the software. The hardware can then be built, or selected from existing systems, in parallel with the software development and testing. In this way, the hardware is more responsive to the need of the software. It will use a more up-to-date technology and will probably be cheaper. Besides, hardware development requires less time than software and significant delays are rare.

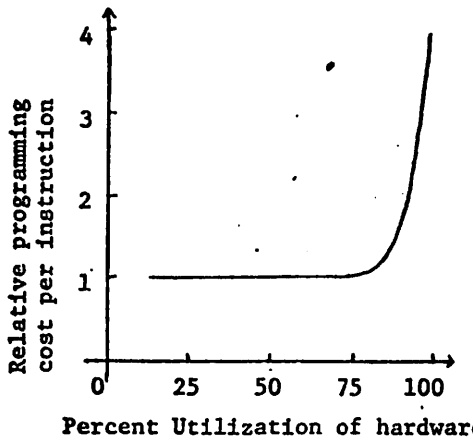


Figure 1. Effect of hardware utilization on software productivity

1.3.2 Increasing software productivity

Let us now look into methods to increase the software productivity of each individual programmer. It is difficult to specify what is meant by software productivity. A common measure is the number of source level instructions that a programmer produces per unit time, e.g., the number of Fortran statements per week. A study by Sackman [Sac 70] shows that the productivity of individuals may vary by factors up to 26:1. The productivity of a programmer can be improved by many methods. On-line programming may cause an improvement of 20% over batch programming. [Sac 70]. The selection of the right programming language, especially the use of special purpose language, may cause a productivity improvement of several fold. There are also tradeoffs between the productivity of the programmer and the efficiency of the program produced. Other important factors that affect the productivity may include stability of program design, amount of mathematical instructions, number of subprograms, concurrent hardware development, etc.

1.4 Improvement of program reliability

A careful reader may notice that all the factors discussed so far are involved in coding of the program only. Software development can be roughly divided into 3 phases: design, coding and testing. A study [Boe 71] has shown that for large-scale programs about 36% of software effort is spent in analysis and design, 19%

in coding and auditing, and 45% in checkout and testing! About half of the effort is spent in removing errors made in design and coding of the program. Any improvement in the reliability of the program and cost of debugging will therefore significantly decrease the total software cost. The goals of reducing cost and increasing reliability can be achieved simultaneously by minimizing the software bugs introduced during the design and coding stages of the program. In order to investigate techniques for reliable programming we must first understand the meaning of software reliability, the characteristics of large programs, and the nature and behavior of software bugs.

1.4.1 Meaning of software reliability

Software reliability is a term that every programmer understands while nobody can give a formal definition. Although very meaningful work has been done in hardware reliability, the theory cannot be immediately applied to software because of the basic differences in behavior and characteristics. In hardware, the reliability of a system is usually defined as the probability that a specified function will be adequately performed for a specified time by the system. In general, it is assumed that the hardware system is perfect (100% reliable) to start with and the components deteriorate with time, creating a probability of failure. In contrast, the elementary components of software are instructions, whose behavior does not change with time. Besides, these components cannot fail. Errors are not caused by the failure of the elementary components but rather by incorrect combinations of them. The interactions between these components are much more complicated than the interconnections of hardware components. The piece of software is put into operation with many bugs still in it. There are no feasible methods of measuring the number of bugs in a program. More complicated still, even when we detect a "software bug" and correct it, we are still not sure that the total number of bugs left in the system is decreased by one, since we cannot predict if our correction procedure has any side effects on the other parts of the program. The correction procedure is not as simple as replacing a faulty hardware component with a good one.

Serious effort has been attempted by many people in deriving a quantitative measure of the 'reliability' of a program. Many reliability models have been proposed. Shooman [Sho 73] proposed a model using a "software reliability function" R(t) as the probability that the system will not fail up to time t. This model is apparently borrowed from hardware reliability theory. Other people, such as Jelinski and Moranda [Jel 73] have formulated similar models. All these attempts have been less than satisfactory because they completely ignore the differences in behavior between software and hardware. They failed to establish connections between the parameters of the models and actual software properties. The applicability of such models is doubtful.

Here, we will not attempt to give any formal definition of the reliability

of a program. Instead, we will treat software reliability as a qualitative measure and discuss different factors which will affect the quality of a program from a reliability point of view. We will say that an error is committed if, given the input value and the specifications of the computation to be performed by the program, the output value is either incorrect or indefinitely delayed.

The reliability of a piece of software may be evaluated from two points of view. We can rate the reliability of a program by the "number" of software bugs inherent in the program, i.e., the number of mistakes made during the design and implementation of the program. Reliability is therefore an inherent property of the piece of software product and is subject to assessment by an analysis of the program. However, software bugs, like software reliability, is not easily subject to quantitative evaluation. It is not clear what is meant by the "number of software bugs" in a program or how to measure it. It has been suggested that the rate at which software errors are detected can be used as a projection of the number of software bugs still resident in the program. The accuracy and validity of such a projection is still questionable. Moreover, even if we were able to measure the number of bugs in a program, there is still no convenient way for us to normalize such a measure so that it can be used as a comparative parameter of the reliability of different programs.

We may also treat the reliability of a program from the viewpoint of the quality of service it gives to a user. To a user, the reliability of a program is evaluated by the correctness of the output that he receives. The reliability of a program can therefore be defined as the probability that a run of the program will give the desired output with a valid set of input data. Since it is the process controlled by the program that performs the required computations, this definition really measures the reliability of the process rather than the program. Since the sequence of codes executed (the process created) is heavily dependent on the values of the input parameters, the probability of obtaining the correct result will depend on the input data selected. Therefore, the reliability of the program should be a weighted function according to the distribution of the input data (the process created) of the given user environment. It depends on the user environment. This seems to be a more reasonable evaluation of the reliability of the program because there may be a part of the program that is full of bugs but rarely used. These software bugs will not affect the operation of most users and are therefore harmless. This definition of reliability is related to the probability that a software bug is activated by a set of inputs. As an extension, we should also take into account the criticality and penalty-cost of the software error. A software bug in the missile firing procedure of a defcnse missile system may make the whole program unacceptably unreliable, even if the rest of the program is error-free.

The user-viewpoint definition of software reliability has other drawbacks. The reliability is not an inherent property of the program. The incorrect

functioning of a program may be due to some program independent errors. Even if the program were perfectly coded, there may be mistakes due to the key-punch operator, the compiler and assembler, and the operating system. Errors may be caused by the incompatibility of the program and the computer hardware. More resources may be requested than the computer can supply. Bugs can occur due to truncation or imprecision in the calculation by the hardware. These errors are extremely difficult to detect since they only arise with the "right combination" of the variables. Other hardware malfunctions, including transient errors and data-sensitive faults, will also give us the wrong result. Input-output oriented errors are not uncommon since many devices have different idiosyncrocies. Errors can also be caused by parallel and asynchronous operations. In multiprocessing systems, typical errors may include resource deadlocks, storage encroachments, timing and scheduling anomalies. These program independent errors are particularly serious to real-time programs and should be checked before the program is put into operation. If we desire a reliable system, we have to take into account the operating environment of the program besides the reliability of the program itself. However, from now on, we will only restrict ourselves to program dependent errors and discuss different techniques to minimize these errors, leading to a more reliable program.

1.4.1 Characteristics of large programs

Large programs (or programming systems), as referred to here, are characterized by complex structure and many instructions. Due to the size of the program, it is usually developed by a large number of programmers, sometimes in different locations. There will be a large number of program components with complicated interactions. It is very difficult for a person to have a good understanding of the whole system. A large program will contain a significant number of possible flow paths so that exhaustive testing is unfeasible. For this reason, such programs are not expected to be completely error free. The number and critical nature of errors or possible errors in the software product will determine its reliability. There are a number of differences between small and large programming systems, such as the methods of implementation, phase structure, expansion of the system, and tolerance to user abuse. These will affect the extent and effectiveness of different validation techniques.

Since the small system has limited authorship, the implementation techniques are homogeneous, i.e., similar methods are used to solve a given type of problem. Consequently, if a given method is validated for one occurrence, it is also validated for other occurrences. By contrast, large systems are developed by large numbers of programmers, each having his own way of thinking. Problems in communication may prevent them from arriving at a common optimal solution for a problem. Instead, various implementation techniques are used for the same purpose, requiring additional validation procedures.

The simple data structure of small systems is contrasted with the complex structure of data bases employed by large systems. The former allows an intuitive understanding of the use of data while the latter obscures the meaning and use of variable names and provides more opportunities for misuse of data. It is also very difficult to provide an effective data structure for a data base used by different programmers in a variety of ways.

The phase structure of a system is another important aspect in validation. Small systems tend to have independent phases with limited interaction. Consequently it may be possible to exercise all possible paths and check interactions by examining the data dependencies of each phase. Large systems, however, will contain complex interactions among functional tasks which are coordinated by a supervisory system. Dependencies are expressed in the supervisory calling sequence. For these systems a thorough investigation of all paths is not feasible and more sophisticated techniques are required to validate interfaces.

The expansion and modification of a system will require re-evaluation and validation. For small systems this task is relatively simple since the effects of changes are limited and easily traced. Modifications to large systems may have more far reaching effects and their acceptance by all other parts of the system must be certified.

The detection and correction of operational errors depends on the system's tolerance to user abuse. Small systems are employed in a limited user community. This implies adequate communication between developer and users to specify program requirements and locate faults. On the other hand, large systems often have a wide community of users and communication is hindered. The detection of faults is more difficult and subtle faults may be propagated through the system. The complexity of the system inhibits understanding by the user.

The problems associated with large programming systems are largely the result of faulty integration of system components, due to communication problems among programmers and lack of understanding of the whole program. The segmentation and interaction between components of large systems presents several types of problems, including data integrity, interface problems, and sequencing problems. In addition to these considerations, there are also errors common to smaller programs such as semantic errors, unreachable code, logic errors, etc. The nature of errors will be investigated in the next section.

1.4.2 Nature of software bugs in large programs

In order to gain some insight into the nature of software bugs, let us briefly review the typical steps of the development of a large software system:

1) Specifications of the requirements of the system.
2) Design of the overall structure and decomposition of the program in flowchart form and the descriptions of the different software modules.
3) Coding of each software modules in some suitable programming language,

usually a "high level" language.

4) Debugging of each software modules with testing sample data.

5) Integration of the tested software modules and debugging of the whole system.

6) Check out of the whole system for delivery.

In all the above steps there are sources of error. The program specifications may be incomplete, leading to ambiguities or software bugs. The designer may have failed to understand fully the problem or have conceived a faulty algorithm. He may have overlooked special cases of the input data. During the coding of the program, there are even more errors. Errors may arise from incorrect semantics and language constructs, such as the misspelling of variables and labels, incorrect use of mixed-mode operations, etc. Logic errors such as "off by 1" in indexing or shifting are not unusual. Array over-write and wrong initialization are other common errors. The use of certain statement constructs such as computed GO TO statements in Fortran are error-prone. This type of statement depends on the value of a variable for determination of transfer locations. A bug would cause the transfer to nowhere or to an unexpected part of the program. Structural errors of the program are also common, such as incorrect flow of control, unreachable program segments, no exit path from a segment, etc. An additional area of concern is that of loop termination. A program loop may be executed an incorrect number of times or even indefinitely, depending on combinations of variable values in conditional branches and limits of explicitly defined loops.

Difficult as it may be, these errors can be pretty well controlled with a little bit of care and patience from the programmer. The real problems usually arise from the faulty integration of the software modules. This is due to a development process in which a large number of programmers are involved. An individual working on a single system component may overlook certain obscure but possible conditions. The lack of complete and rigorous interface specifications, coupled with the misunderstanding of the scope or intent of component operation, may lead to improper use or unforeseen side effects. The flow of information from one component to another is often the source of interface errors. For example, consider the passing of parameters in calling a subroutine. If the number, format, and type of parameters are not consistent, the subroutine may make unexpected modifications to the parameters or improper operation due to the passing of incorrect parameters. In a large program errors can also arise from the improper sequencing of operations, which is obscured by the complexity and number of flow paths.

The order of operations for a certain process may be changed when integrated with other processes. For example, a routine which accesses and transforms data in a certain sequence may be disrupted if a second routine alters the same data. Improper interface and sequencing may lead to errors in data integrity. Data integrity refers to the maintenance of proper data in correct locations at the

prescribed time. Errors include overruning array bounds, so that adjacent data are destroyed, and non-alignment of common data blocks. Equivalences between variables of different names may cause the unexpected change in the value of one variable when the value of its equivalent is altered.

Therefore, we can see that the most common errors in implementation can be described in roughly 5 major categories according to the place where they are found: 1) Interfaces, (2) Sequencing, (3) Data integrity, (4) Semantics and language constructs, (5) Structure and well-formation.
These are not intended to be all-encompassing and there will be some interaction and overlapping, but most errors are traceable to one (or more) of these problem areas.

1.4.3 Behavior of software bugs in large programs

In general, the complexity of a system will depend on the number and interaction of system components, while at the component level, complexity depends on the number of branches and external references. For a large program, exhaustive testing is unfeasible. Therefore such programs always contain residual errors which survive the design,development and debugging stages. The occurrence of errors in the development of the program may be expected to follow a general pattern as in Figure 2.

Initial use will uncover increasing numbers of errors as the system is used more frequently and to fuller capacity. The correction of major errors will then result in a gradual decrease in error detection until only infrequent errors occur. The piece of software becomes "operational" when the rate of errors found is less than a certain number epsilon, which represents the level of tolerance of the user to software bugs. It may seem strange to note that the number of errors that are detected and fixed after the system is operational seems to be almost constant. One may expect a monotonically decreasing number of errors with our debugging effort, since bugs are constantly detected and removed from the program. However, in the process of correcting a detected error, the programmer may unintentionally introduce some subtle errors in other parts of the program, especially if good documentation is not available. A study by McGonagle [McG 71] shows that 19% of the errors of a set of programs resulted from unexpected side effects to changes. Another reason for the constant error rate is that a large portion of the program is not tested or exercised. Errors in this large portion of the code remain dormant until much later.

The behavior of systems with several releases may present a pattern similar to that of Figure 3, since every release represents a major revision of the program specifications and modification on the program code. If the residual type of errors can be detected and corrected before a program is released for use, the peaks of these curves will be effectively reduced, thus improving the confidence level and reliability of the program. This is the objective of the evaluation and

validation process.

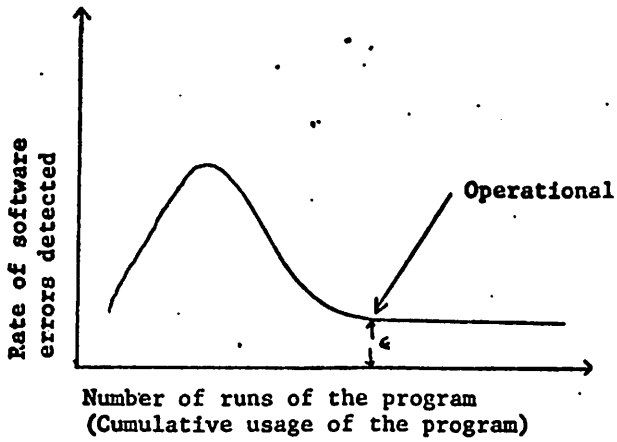


Figure 2. Behavior of software errors

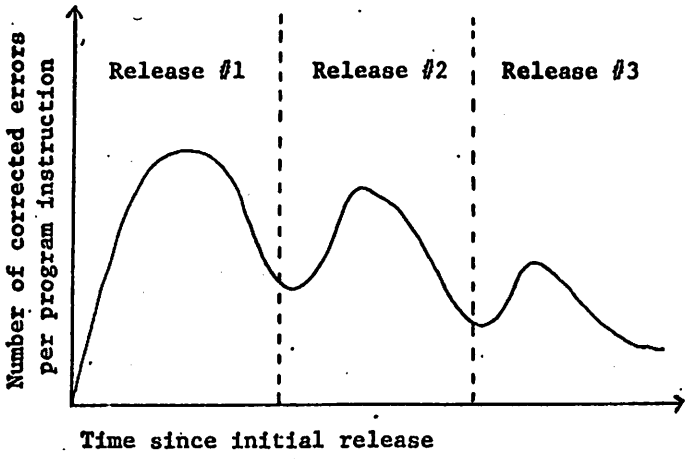


Figure 3. Errors in multiple release program

1.5 Conclusion

From our discussion above, we can see that the main fundamental reasons for the large number of errors are the complex system application, the loose specifications, together with the large number of programmers. The inefficiency of 'bug removal' is due to the lack of software validation and evaluation tools and methodologies. The reliability of a program can be improved by 2 approaches: the 'analytic' approach and the 'constructive approach'. The latter includes methodologies for developing more reliable software, such as structured programming and software defense. The former approach is primarily concerned with testing and validating the program after it is written, using techniques such as proving program correctness and automated tools. In large real-time programs like those in ballistic missle defense and air-traffic control, errors are very disastrous. After the removal of critical software errors, one still has to worry about the integrity of the program at the moment it is being executed in order to ensure the reliable operation of the

system. Security measures have to be implemented to protect the program against unauthorized tampering of the program. All of these will be discussed in the following sections.

2. The analytic approach to improve software reliability

The analytic approach is primarily concerned with the validation of the reliability of the program after it is written. It is done through an analysis of the program after it is coded. Roughly speaking, two approaches can be taken. The first approach involves the proof of correctness of the program by some formal means. A proof of correctness can, of course, establish our confidence in the reliability of the program. However, this approach becomes infeasible when the size of the program is large. The largest program proved by this method has only 433 Algol statements. [Goo 68]. The second approach has the more humble goal of detecting and removing errors from the program. This is the more conventional method of debugging. Although this method can never show that a program is completely reliable, it is practical for a large program because a lot of the techniques can be automated. The computerized assistance greatly facilitates the debugging effort of the programmer. After the major errors are removed, the program may be quite reliable.

2.1 Proving program correctness

The process of proving program correctness is an analytic method to show that the program, with inputs satisfying some constraints, will terminate and will produce outputs which are specified functions of the inputs, provided that the program is correctly compiled and executed in a 'perfect computer'. By a 'perfect computer', we mean the Utopia of every programmer, with such features as a memory large enough for any program, an arithmetic processor with no errors due to round-off, underflow, overflow, etc. Although proving program correctness does not consider the compatibility of the program and the machine, it does prove the correctness of the coding of an algorithm in a suitable programming language, provided that syntactic errors are absent. Hence, it establishes our confidence in the reliability of the program and reduces the testing cost of the program. In fact, most of the software errors are caused by the incorrect coding of the program by the programmer.

Rigorously speaking, a proof of correctness should include a proof of program termination. In practice, we may separate the verification procedure into two steps. The first step is a proof of partial correctness, i.e., that the program yields the correct answer if it terminates; the second step will be a proof of termination of the program. (Some procedures can perform both steps simultaneously.) Two approaches can be taken in establishing the correctness of a program, namely, by an informal proof, or by a formal proof utilizing a mechanical theorem prover.

2.1.1 Informal approaches to proving program correctness

The approach towards an informal proof of program correctness dates back to the days of Goldstine and Von Neumann [Gol 63], who noted that the program can be verified, at least in principle, if the programmer can describe the state of all the program variables after each step, or possibly after some selected steps, of the program. An inconsistency at any point will indicate a programming error. McCarthy [McC 62, 63, 67], used a function-theoretic approach similar to this. He assumes that at the start of a computation each cell of the computer memory contains a number. An ordered sequence of these numbers is the "state vector" of the computation. Each computer operation is considered as a transformation of the existing state vector into another state vector. Therefore the program can be considered as a function in the state vector space. McCarthy introduced a formalism (conditional forms) for defining programs as recursive functions. Afterwards, the process of verifying the correctness of the program reduces to a problem in recursive function theory and a method (recursion induction) can be used. Essentially, recursion induction is a set of axioms that can transform a recursively defined function into an equivalent function. McCarthy and Painter [McC 67] have used this approach to verify a very simple compiler.

Naur [Nau 66], generalizes such an approach by considering the state vector as a vector of symbolic values rather than numeric values, e.g., X, Y, and Z instead of -1, 2.3, etc. Computer operations are carried out with these symbolic values to obtain symbolic expressions such as X + Y and X - Y as new elements in the state vector. Logical connectives can be introduced to accommodate branches by indicating the conditions leading to different symbolic values. The symbolic outputs therefore express the transformations the program performs on the input variables. This approach, known as the proof of algorithms by general snapshots (state-vectors), is impractical since there will be too many symbolic expressions, each of which can become very complicated even for a small program.

A natural simplification of the above procedure to make it practical is to trace only the transformations of important variables and to develop symbolic expressions for these variables only at strategic locations within the program. Hence, we are using only a subset of the elements of the state vector and the "state" of these variables are updated only after some computations, not after each computer operation. Therefore, the sequence of specified state vectors becomes a set of "assertions" about the relationships of important variables scattered through the program. The process of verifying the correctness of the program becomes a proof that each assertion is true every time it is reached by the program. In order to prove an assertion, we can assume that all previously reached assertions are true. There is no well-defined procedure to formulate and locate these assertions. In general, there is a tradeoff between the complexity of the assertions and the number of assertions that we have to use. Floyd [Flo 67] develops the logical

foundations for the informal-assertion method of proving program correctness, and subsequently suggests how the process of verification can be mechanised. Let us illustrate with an example the process of assigning assertions to a flowchart program and the proof of consistency of an assertion as a function of the previously reached assertions.

Example:

This simple example illustrates the informal inductive assertion method of proving program correctness. This program divides a positive integer X by another positive integer Y by repeated subtraction. Let Q be the quotient and R the remainder. The flowchart of the program is shown in Figure 4.

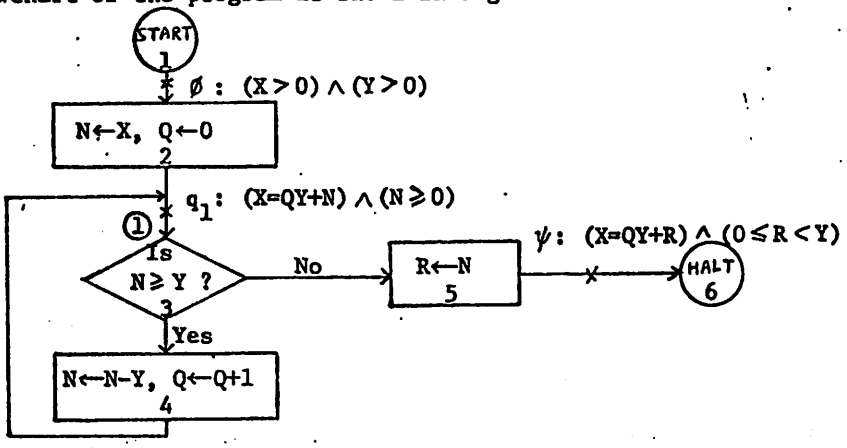


Figure 4: Flowchart of the example

The input assertion, denoted by ∅, specifies the domains of the input variables and the relationship between their values. In this example, the input assertion is $(X > 0) \wedge (Y > 0)$. The output assertion, denoted by ψ, specifies the desired relationship between the output variables and the input variables, i.e., the desire result from the program. In this example, the output assertion is $(X = QY + R) \wedge (0 \leq R < Y)$. By examining the program we conjecture that the assertion $(X = QY + N) \wedge (N \geq 0)$ must be satisfied any time the program control is at point ①.

In order to prove the correctness of the program by the inductive assertion method, one must show that the truth of the assertion at the beginning of each path of the program, followed by the execution of the path, implies the truth of the assertion at the end of the path. First of all, we must show that ∅, together with the execution of the path (1, 2), implies that q_1 is true, i.e., $(X > 0) \wedge (Y > 0)$, together with the execution of $(N \leftarrow X) \wedge (Q \leftarrow 0)$ implies $(X = QY + N) \wedge (N \geq 0)$. It does not take too much effort for the reader to see that this is true. Therefore q_1 is satisfied when control first passes to point ①. Next, we must show that if q_1 is satisfied any time control is at point ①, then q_1 must also be satisfied whenever (if at all) control returns to point ①. Therefore we have to show that q_1, followed by the execution of the path (3, 4), implies q_1 itself, i.e.,

$(X = QY + N) \wedge (N \geq 0)$ is still satisfied after the operation $(N \leftarrow N-Y) \wedge (Q \leftarrow Q+1)$, if $N \geq Y$. This is obviously true since $X = QY + N$ can be rewritten as $X = (Q + 1)Y + (N - Y)$ and $(N - Y) \geq 0$ since $N \geq Y$. Therefore, q_1 is satisfied no matter how many times the loop is executed. Now we must show that the correct result is indeed produced. We have to show that q_1, followed by the execution of the path (3, 5, 6) implies ψ, i.e., $(X = QY + N) \wedge (N \geq 0) \wedge (N < Y)$, together with the operation $R \leftarrow N$, implies that $(X = QY + R) \wedge (0 \leq R < Y)$. This can be verified very easily. The partial correctness of the program is therefore validated. The question of termination of the program can be answered easily by observing that the loop can only be executed a finite number of times since N is decreased by Y every time the loop is executed and $Y > 0$. The correctness of the program is hence established.

After the example, let us describe the inductive assertion method in a more systematic way. As presented here, the formulation is as described by Good. [Goo 70]. A program is a finite ordered set of statements, with the first statement as start and the last one as halt, and the remaining statements as null, assignment, or two-way branch statements. An assertion is a predicate attached to a point in a program. The first assertion is the input assertion, denoted by $\emptyset$. It is attached to start, and specifies the domains of the input variables and the relationship between their values. The last assertion is the output assertion, denoted by ψ. It is attached to halt, and specifies the desired result from the program. A path is a sequence of statements $(S_1, S_2, \ldots, S_n)$ such that it is a valid execution sequence of the program.

Let $(S_1, S_2, \ldots, S_n)$ be a path with assertion q_1 attached to S_1 and assertion q_2 attached to S_n. The path is said to be verified if it can be shown that q_2 is satisfied if q_1 is satisfied at the beginning of the path and the statements $S_1, S_2, \ldots, S_{n-1}$ are executed. A verification condition for a path is the condition that must be satisfied in order to verify the path. The proof of correctness for the program consists of choosing and attaching the inductive assertions at different locations and of verifying all the paths in the program by constructing and proving the verification conditions.

The first step of the process is to choose a subset C of statements from the program to which assertions have to be attached. The set C should contain the first (start) and last (halt) statements, and at least one statement from every loop in the program. The reason for choosing at least one statement from each loop is to allow breaking the program into loop-free paths. Then the programmers have to supply the assertions for every statement in C. The assertions will include $\emptyset$ and ψ. The choice of assertion is very closely related to the choice of statements for C. The assertions can be quite simple if they are appropriately located. Therefore, it requires much insight of the programmer about the behavior and structure of the program. Still the method of choosing assertions is more like an

art than a science and no general guideline seems feasible. It is similar to the choice of the induction hypothesis of mathematical induction. Therefore, this is the most difficult part of the process and it does not seem hopeful that this part can be automated, although it can be computer-aided.

After the assertions have been supplied and attached to the program, we have to construct a verification condition for every path that proceeds from one assertion q_1 (a statement in C) to another q_2, with no other assertions in between. Therefore, the verification condition depends on the initial and final assertions, together with the operations performed by the statements in between. The forward accumulatiom method of construction of a verification condition is presented here. The verification condition of a path P is formed as

$$q_1 \wedge (\text{assignment terms}) \wedge (\text{traversal conditions}) \Rightarrow q_2,$$

where the assignment terms are the operations of the assignment statements along the path and the traversal conditions are the logical conditions for the branch statements along P under with the path P will be taken. The assignment terms and traversal conditions do not include the last statement in the path since the assertion q_2 has to be satisfied before the statement to which it is attached is executed. The construction of the verification conditions can be fully automated by using predicate calculus. Afterwards, the conditions can be mechanically proved in order to validate the correctness of the program, if the program is indeed correctly coded.

Systems have been implemented to automate part of the process of proving program correctness. The philosophy is to let the computer take over as much of the burden as possible. Two of the most well-known ones are that implemented by King [Kin 69], and by Good. [Goo 70]. King's Program Verifier only accepts a special Algol-like language, with only integer variables and one-dimensional arrays. Relational operators ("greater than", etc.), "GO TO"s, and logical connectives ("and", "or", etc.) are included. The assertions are Boolean expressions supplied by the programmer. The verification conditions are generated automatically using a backward traversal of the path. An automatic theorem prover is then used to prove these conditions. King's system is the most automated system of this kind yet implemented. Good's system, on the other hand, is an interactive program. It also uses a programming language similar to King's. Assertions are manually supplied. The verification conditions are automatically generated by the system. The programmer then supplies proofs of the verification conditions. The proofs are accepted by the system without question and stored in the computer. When all such proofs have been supplied, the computer outputs the completed proof.

The proof of program correctness by such an approach has certain degree of success. However, it can only be applied to programs of relatively small size. The largest programs proved by hand using such an approach consist of several hundred instructions. The most automated systems have only proved programs of less

than a hundred instructions. Besides, many of the assertions and the proofs have to be supplied by the programmer himself. This process is as fallible as writing the program. In order to make advances in this area, it seems that a completely mechanical verifier is the only foolproof approach. Such a fully automated approach will require more formalism in the proofs. In the next section we will briefly describe some of the formal approaches to the proving of program correctness.

2.1.2 Formal approaches to proving program correctness

As concluded in the last section, it seems that a formal mechanical approach to program verification appears to be the most reliable, although the efficiency is expected to be low. The verification of a program can be reduced to the proving of a theorem in the first-order predicate calculus. Speaking very informally, the first-order predicate calculus is a formal system which consists of constants, variables, functional constants, predicates, the logical constants T (true) and F (false), the logical symbols $\wedge$, $\vee$, $\sim$, $\supset$, $\equiv$, $\forall$, and $\exists$, which can be combined to form well-formed formulas of first-order logic according to some rules with the aid of commas and parentheses for punctuation marks. (A formal and complete description of formal logic is beyond the scope of this paper. Interested readers are referred to the discussion by Manna. [Man 69a].) With this in mind, the section attempts to give the reader some intuitive feelings about the formal approach. The discussion will be very informal and interested readers are encouraged to read the referenced papers for a more rigorous and complete treatment.

Most mechanical theorem-provers for first-order logic employs the resolution principle. This is an indirect proof of a theorem: we assume the negation of the theorem and try to derive a contradiction. There are systematic methods to construct such a proof and the process can be made more efficient by introducing some heuristic procedures. This resolution process works very satisfactorily if the conjectured theorem is indeed true, but very inefficient otherwise.

Manna [Man 69b] has proposed a formal approach to proving program correctness. He shows that one can set up well-formed formulas in the first-order predicate calculus corresponding to an arbitrary flowchart program. In order to facilitate the conversion, the program is expressed in a standardized form such that for every program statement i, we can define a well-formed formula W_i. (W_i is very similar to an "assertion" in the Floyd-Naur sense and q_i is the predicate associated with W_i.) A well-formed formula [P, ψ] is then formed as

$$q_1 \wedge W_1 \wedge W_2 \wedge \cdots \wedge W_n .$$

where q_1 is an unspecified predicate associated with W_1. Then Manna's Satisfiability Theorem states that the program P is partially correct with respect to $\emptyset$ (input assertion) and ψ (output assertion) if and only if $W_p[\emptyset, \psi]$ is satisfiable (i.e., is true under some interpretation of the predicate symbols q_i, for example,as the Floyd assertions), where

$$W_p[\emptyset, \psi]: \quad (\forall x)\ \{\emptyset(x) \supset [P, \psi]\ (x)\}\ .$$

We can see that this theorem is essentially equivalent to Floyd's results.

The more important work is the Unsatisfiability Theorem which states that the program P is correct with respect to $\emptyset$ and ψ if and only if $\hat{W}_p[\emptyset, \psi]$ is unsatisfiable (i.e., is false under every interpretation of the predicate symbols q_i), where

$$\hat{W}_p[\emptyset, \psi]: \quad (\exists x)\ \{\emptyset(x) \wedge [P, \sim \psi]\ (x)\}\ .$$

Therefore, a program can be demonstrated to be totally correct in a single proof process. However, the disadvantages of this approach is that the theorem prover is very complex and the entire program is treated as a single entity and thus decomposition of the verification process is impossible. It also requires the program to be written in a special language so that a well-formed formula W_i can be easily generated from statement i.

The interest in the formal proof of correctness of programs has produced a new area of research, that of automatic program synthesis. Much interesting work has been done by Manna and Waldinger [Man 71]. The programmer would only supply the input-output relationship of the desired program, together with some assertions about the program algorithm. Theorem-proving techniques can be used to prove the correctness of the assertions and the proof itself can be used to build the desired program. This program, generated by the computer, would hopefully be free of errors.

2.1.3 Conclusion

An assessment of the techniques for proving program correctness has been discussed by Elspas et al. [Els 72]. The reader is also referred to the complete bibliography of London [Lon 70] for more information. All these techniques are infeasible for any sizable programs. [Lin 72]. In order to make meaningful advances in this field, more research works are still needed in the field of formal specifications of programs, formal semantics of programming languages, and mathematical theory of computation. A concise and precise specification of the program will allow us to know effectively what the correct program should do. Formalization of language semantics will allow us to convert a source-language program into a canonical model easily. Mathematical theory of computation enables us to develop the verification conditions and the proof of the program. Before any significant breakthroughs have been achieved in these areas, mechanical verifiers will continue to be very inefficient. Automatic program synthesis will be a very distant goal.

In the meantime, we just have to settle for the use of the informal techniques for proving program correctness. Though inefficient, these techniques are very effective when we integrate the proof with the program design, especially when applied selectively to the critical sections of the program. They are also useful for the testing of small modules of the program before integration. However, these

techniques are subjected to human errors and are very 'unreliable' for large programs. Automated software evaluation and validation systems seem to be the more feasible analysis tools for large programs.

2.2 Automated Evaluation and Partial Validation

2.2.1 Introduction

The current trend in software shows an increasing demand of large real-time software. At present, the techniques in proving program correctness are infeasible to solve the problem of reliability in large software systems. Naturally it has become necessary to use a more cost-effective and practical approach. In order to analyze any sizable programs efficiently, computer assistance becomes essential. Since a complete validation of the correctness of a program is impractical, we will only aim at a partial validation of the program, using techniques that are subject to a high degree of automation. The objective of automated evaluation and partial validation is to achieve an acceptable degree of assurance of the reliability and performance of the produced software to be put into operation.

The characteristics of any computing system can be classified into two categories: the structural characteristics and the behavioral characteristics [Ram 67]. A program is usually specified by its behavioral properties, such as the relationship between its inputs and outputs. The structure of the program, however, is usually left to the discretion of the designer. The program can then be looked upon as the superposition of behavioral characteristics of the components on its structural form. The complete validation of the program means to verify the correct operation of the system for all possible inputs, by obtaining and evaluating the complete behavioral characteristics. However, the collection and examination of all behavioral characteristics is practically an infeasible task, especially in the case of a large program.

A more feasible approach would be to decompose those characteristics into a certain number of classes and then to validate each class of characteristics to a limited extent. This is the basic idea underlying the partial validation.

Decomposition of behavioral characteristics is a non-trivial task. Fortunately, the careful examination of the structural characteristics reveals various useful informations which could help devising the decomposition scheme and the validation strategy. Naturally, the analysis of structural characteristics forms the important initial basis for most automated evaluation and validation systems (AEVS).

2.2.1.1 Error detection techniques

Different types of software errors in large programs have already been discussed in section 1.4.2. Most of them have to be detected and corrected during the debugging phase of the program. Considerable attention has been given

to debugging systems, resulting in sophisticated techniques for trapping and tracing variable values, interactive step-wise execution of programs, and many other features [Rus 71]. These conventional systems have been successful in providing very useful aids to programmers in correcting errors. However, there are certain limitations which present problems in debugging large programs. For example, the amount of information necessary to determine long and complex paths through a large program may be prohibitive. Additionally, debugging systems are basically designed to trace the source of known errors which occur during execution with various test cases. Therefore, they do not necessarily predict errors or possible errors.

Efforts to detect residual errors must go beyond traditional debugging systems to provide a more complete program analysis. Various techniques employed in validation systems are now discussed.

(1) The checking of component interfaces requires a description of the system structure and detailed information on parameters and information passed between components. Graphical analysis is particularly useful in this area since it provides a means of displaying the program structure at various hierarchial levels. The interrelation and interdependency of components can be determined from this graphical representation and, together with lists of the data passed, can be examined to uncover interface errors.

(2) Sequencing errors can be detected through the automatic extraction from program code of certain specified events. The flow paths defined by these sequences of events can be compared with the proper sequences.

(3) The problem of data integrity necessitates a detailed examination of variables and their use throughout a program. This includes a mapping of data common to various components, locations where this data is used and modified, etc. This information can be used to detect such errors as unauthorized use or availability of data. The use of execution-time monitors is also important in checking the values of critical variables such as array indices, conditional branch expressions, etc.

(4) Semantic errors and error-prone constructs can largely be detected by a detailed examination of the program source code. This is most easily accomplished by an automatic language processor which examines each program statement, recording pertinent information and recognizing the error-prone conditions peculiar to that language.

(5) Program structure is easily determined by graphical techniques as previously mentioned. The graphical representation can be examined for its connectivity characteristics, thus exposing errors in the well-formation of a program.

From the above description, it is evident that many of the residual errors present in large systems can be detected using the techniques outlined. It is

assumed here that, once detected, these errors may also be easily corrected. These techniques are used throughout the structure of the large validation system. However, error detection is only one of the functions of any AEVS. The other validation functions are discussed in the next section.

2.2.1.2 Validation functions

The correct operation of a large real-time software system has two aspects. The system not only has to produce the correct output for the given input but also has to satisfy the performance requirement such as execution time bound, space constraints, etc. Therefore, validation functions can be classified roughly into two groups: error diagnosis and performance verification. The former is supported by a diagnosing aid system, while the latter is supported by an evaluation aid system. The AEVS is an integration of both aid systems.

The implementation of these functions has appeared in various forms: debugging, simulation, formal proof of program correctness, software testing, etc. Here testing is a systematic process which mainly determines that an error exists, while debugging is regarded as a follow-up process which localizes the cause of errors and corrects them. This doesn't mean that software testing is restricted to error detection. As will be seen later, it can support other validation functions such as error location and performance verification, too. According to the type of the function it provides it is further characterized as either functional testing or performance testing [Elm 71].

The essential requirement for an effective validation process is the amenability to automation. Software testing is extensively employed in most AEVS's due to its high adaptiveness to automation and its effectiveness in validating a large software. This and other processes are discussed later in detail.

2.2.1.3 Structure of the AEVS

The common philosophy in most AEVS's is to provide automated tools which relieve the programmer of collecting data about both structural and behavioral characteristics and assist him in evaluating the collected data. Strategies in those systems are generally of two types. One is a hierarchical bottom-up validation in which each module is (partially or completely) validated first and then the validation of module interactions and interfaces follows. The other is a hierarchical top-down validation in which validation starts with the global program and proceeds to the smaller segment (or module) with the increasing assurance.

Each of these two has its own advantages and suitable application environments. Where the program contains abundant bugs, the bottom-up approach will be more effective since validation can proceed in more straightforward fashion without much interference from multiple errors. On the other hand, the top-down approach will be more cost-effective where the program is expected to have a small

number of bugs. In this paper we assume that the program is at least syntactically correct. Although not essential, the top-down approach is implicitly adopted in several places.

From the implementation point of view, an AEVS consists of two parts: static analysis and dynamic analysis. The static analysis is the validation process performed only by examining the external form of the software, i.e. code itself, without executing it. It reveals most of structural characteristics. A considerable amount of behavioral characteristics are also verified by it. On the other hand, the dynamic analysis is the process of software testing performed by running the software with the devised test inputs and evaluating the output results. Its function is to validate various behavioral characteristics which cannot be efficiently identified by the static analysis. The performance of the dynamic analysis is enhanced on the basis of informations provided by the static analysis.

Various techniques employed in both parts are discussed in subsequent sections. Most of them can be found in two representative systems, ACES [Mee 73, Ram 73a, Ram 73b] and PACE [Bro 72a, Bro 72b]. Before going into those validation techniques, techniques of modelling programs are examined since the efficient program model is a cornerstone for automated validation.

2.2.2 Overview of Program Models

The purpose of modelling programs is to obtain an easy-to-use representation of only those informations relevant to the intended analysis, while unnecessary details are masked. The model must be simple. They must be easily represented and manipulated in a computer. The representation of the process must be homogeneous such that the same analytical tools can be used at any level. This implies that by choosing the proper level of representation, details not useful for the problem at hand must be masked out. Another important requirement is that the modification on the program being modeled should not be cumbersome in simulation. Since the structural characteristics serve as useful guidelines for the cost-effective validation, the model should be suitable for an efficient structural analysis.

Among various models three representative ones are briefly reviewed in this section: finite-state-machine (FSM) model, decision-table model and directed-graph model [Ram 71b].

2.2.2.1 FSM model

In this model, the computer (the sequential machine) is taken from state to state by a transition table (procedure) and a set of inputs (data). Therefore all behavioral characteristics are embedded in this model without being abstracted. It is evident that the size of the model becomes unmanageable in the case of a large program due to the rapid increase in the number of states. Although

certain formal proof techniques benefit from this model [Man 69b], the inability of the model to contend with a sizable program is a serious drawback.

2.2.2.2 Decision-table model

In this model, a program is represented by a decision table [Kin 67]. A rudimentary decision table is illustrated in Fig. 5.

	Action 1	Action 2
Condition 1	Y	N
Condition 2	Y	-
Condition 3	N	-

Fig. 5 Example of a decision table

The vertical coordinate lists a set of conditions that may or may not occur in all possible combinations. The horizontal coordinate lists a set of actions to be taken by the program. These could be different procedures or merely GO TO statements. Each column of the table indicates the subset of conditions that must be satisfied if the action listed under that column is to be carried out. Y stands for yes and N stands for no, and a dash (don't care symbol) signifies that the particular condition involved is irrelevant to the action in the corresponding column.

A decision table contains less information than a corresponding flow-chart for the same logical process. A flow-chart of a conditional phrase contains the logical rules of the problem and also specifies the order in which various tests will be carried out. The decision table does not indicate how the logic should be structured in terms of program steps. Consequently, it doesn't support the exploration and analysis of the structural characteristics. This is the major drawback making the model inefficient for the purpose of validating a large program, though it provides a partial verification of the logical correctness of the program and an assistance in generating test inputs.

2.2.2.3 Directed-graph model

This model has its root in the flow-chart. The representation is conceptually simple and natural. In this model, a program is abstracted into a directed graph where each node corresponds to a set of statements and each directed arc represents a possible transfer of control from one node to another node. When each node represents one statement or a set of sequentially executed statements, a graph model of a program is called a program graph. A simple example of a program graph is shown in Fig. 6.

```
1(    V1 = P1
      I = 0
2   50 I = I+1
3(    V2(I) = P2(I)
      IF (I.EQ.10) GO TO 100
4(    ——
      GO TO 50
5   100 V3 = SUM(V2)
6     IF (V3.GT.100) GO TO 150
7(    ——
      GO TO 200
8(  150 ——
      GO TO 200
9(  200 V1 = V1+V3
      RETURN
```

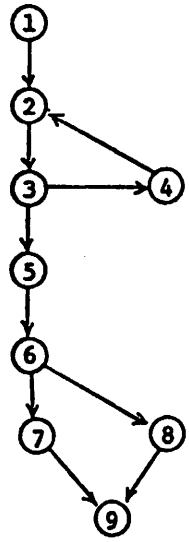


Fig. 6 Example of a program graph

This model clearly reveals structural characteristics of a program, while unnecessary details about functional characteristics are masked out. Necessary functional characteristics are selectively associated with each node depending upon the intended analysis. All possible paths, loops, entries and exits can be easily detected.

The size of the program graph is generally in direct relation with the size of the program. The complexity of the analysis increases more rapidly with the size of the program graph. Therefore, it becomes desirable to devise a procedure by which a large object can be attacked piece by piece where the size of each piece as well as the complexity of its analysis becomes more manageable. Techniques of iterative abstraction have been developed. The iterative abstraction contains two aspects: loop abstraction and link-subgraph abstraction. A loop in a program has often been an obstacle in efficient validations. It contributes to increases not only in the number of logical paths and the structural complexity but also in the undecidable properties. The separation of a loop and the manipulation of a loop independent of the rest of the program graph makes the total validation process simple and uniform. This is the motivation behind the loop abstraction.

A maximal strongly connected (MCS) subgraph is a strongly connected subgraph that includes all possible nodes which are strongly connected with each other. The replacement of every MSC subgraph in the program graph by a single node transforms the program graph into the reduced program graph (RPG) [Ram 66, 67]. Fig. 7 shows the RPG corresponding to the program graph of Fig. 6.

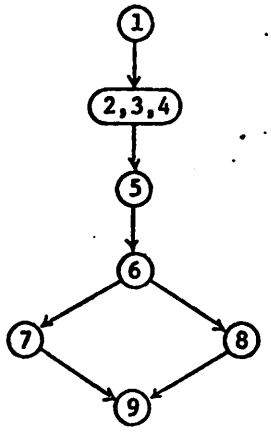


Fig. 7 The RPG of the program graph of Fig. 2

This loop abstraction contributes to the significant reduction in the size of the model and the structural complexity. If further abstraction is desirable due to the large size of the RPG, then the link subgraph abstraction can be applied. A link subgraph is a subgraph that contains no strongly connected subgraphs or unconnected subgraphs in it [Ram 67]. Fig. 8 is the result of the application of the link subgraph abstraction to the RPG of Fig. 7. This is called the basis graph.

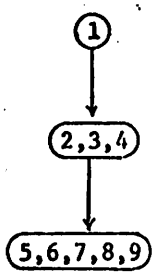


Fig. 8 The basis graph of the program graph in Fig. 3

On the basis of this modelling technique, various analysis can be performed. In general, those can be categorized into two types according to the order of abstractions analyzed: top-down and bottom-up. The basic idea of top-down strategy is as follows. First, the basis graph or RPG is used by the first-step analysis. As a result, the more detailed analysis of a certain node becomes necessary. Then the subgraph corresponding to this node is taken for the next-step analysis. If the subgraph is a link subgraph, there is no difference between the first-step analysis and the second-step analysis. If it is a MSC subgraph, then the technique can be applied for opening the loop with the removal of feed-back arcs [Ram 67]. This is illustrated in Fig. 9.

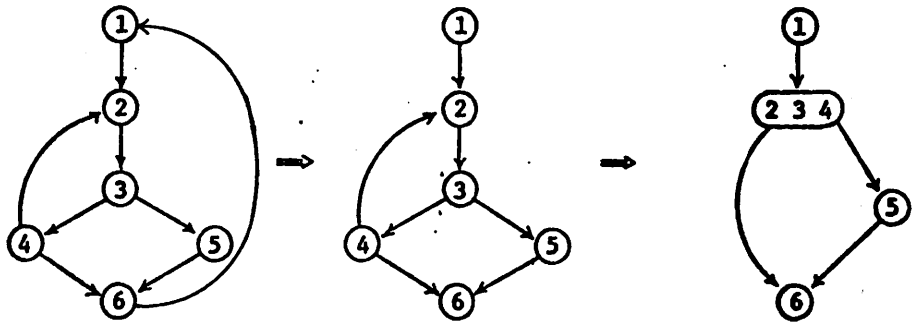


Fig. 9 Loop opening and reduction

Thereafter the modified subgraph can be used by the second-step analysis or abstracted into the RPG and then analyzed. Therefore, the analysis is essentially of iterative nature.

In the bottom-up strategy the analysis proceeds in the reverse order of the top-down analysis. Subgraphs at the lowest level are taken for the first-step analysis and abstracted into a node in the graph model for the next-step analysis. The graph model used by the last-step analysis is the abstraction of a total program.

A number of techniques for manipulating the graph model by the computer are available. Some basic ones appear in Appendix A. In the rest of the discussion, this graph model is used as a basis in describing various validation strategies.

2.2.3 Static Analysis

As mentioned earlier, an organized validation effort can be developed using the two step approach: static analysis and dynamic analysis. The former is based on the examination of the program code while the latter is based on the test runs of the program. The static analysis part of the AEVS is discussed here and the dynamic analysis part is discussed in section 2.2.4.

The main objectives of the static analysis are (1) to analyze the program for the detection of various semantic and structural anomalies, and (2) to provide backgrounds for the efficient dynamic analysis. That is, various structural characteristics are identified and unreliable constructs are pointed out as the target of the dynamic analysis. In pursuit of these objectives, a large amount of repetitive scanning processes are involved. In order to increase the efficiency of the analysis, the generation and use of the data base are common to most AEVS's. Therefore, the static analysis contains three major aspects: data base generation, structural analysis, and detection of vulnerable constructs.

2.2.3.1 Data Base Generation

The construction of a data base is intended to provide a convenient means of retrieving various program characteristics. The philosophy here is to

relieve both the program investigator and the analyzer of the tedious process of examining program listings for information required to validate the program. Furthermore, it forms the basis for other validation techniques, thus eliminating much duplication of effort.

For the sake of clarity, the discussion on data base proceeds with typical examples rather than general arguments. Most of the examples are extracted from two systems, ACES and PACE. Typical examples of data bases consist of the following components: symbol table, symbol use table, statement type table, global storage map and the program graph.

The symbol table and symbol use table are illustrated in Fig. 10.

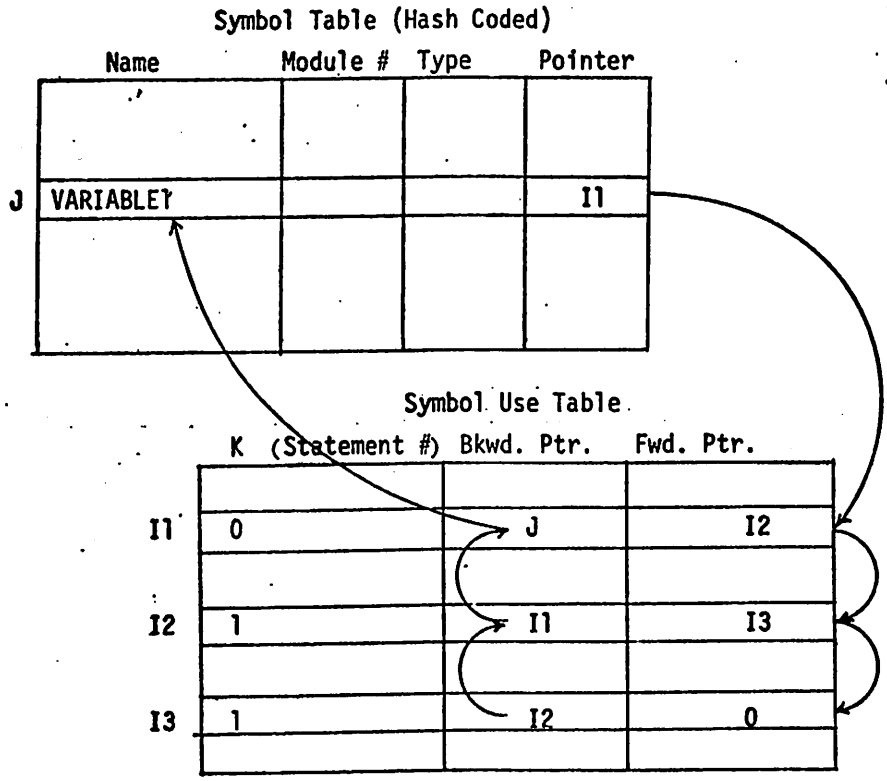


Fig. 10 Examples of symbol and symbol use tables

The symbol table normally contains information regarding all variables, items, functions, macros and labels used in a program. An entry in this table consists of the symbol name, module number, type and linkage to the symbol use table. On the other hand, the symbol use table contains a record of each use of a symbol name in a program. An entry consists of an indicator for the type of use (either input or output to the statement), the statement number in which the symbol was used, and linkage to other references to the symbol contained in the table.

The use of a hashing technique seems to be suitable for providing an

efficient access to the symbol table. That is, the address of the storage where a symbol name is stored is determined by the hash-coding with the character code for the characters making up a symbol name. This technique provides a good distribution of table entries and a rapid access to any particular entry. The linked list structure of the symbol use table provides immediate access to the chain of occurrences for each symbol name, while information (list of symbol references) pertaining to a given statement are grouped in sequential locations of the table.

These two tables provide complete static information on all program symbol names and statements in a neat way and allows the retrieval of answers to questions such as the following.

(1) Does variable V_i appear as an input (output) to any of the following statements: $s_1, s_2, \ldots, s_k$?

(2) In what statements does V_i occur?

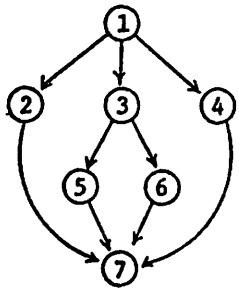
(3) What are the inputs (outputs) to statements s_k?

(4) Does any variable appear as an output and not as an input?

(5) What are the inputs for conditional branch s_i? Where do they appear as outputs? What are the inputs to these statements? (In this manner the user can determine which variables and statements affect the outcome of a conditional branch statement.)

This information is important in the analysis of semantic properties and anomalies. Moreover, it is an useful aid to implementing changes in syntax, program modifications, and changes in programming practices. For example, the effects of changes in a program variable, macro or label can be easily determined by accessing the list of references to that symbol in the program module and other related modules.

The statement type table is simply a list of codes indicating the statement type of each statement in the program. The logical structure of a program is stored either in a connectivity matrix [Appendix A] or in a successor table, a modified version of a connectivity matrix. This is shown in Fig. 11.



program graph

	pointer
1	3
2	4
3	6
4	7
5	8
6	9
7	0

pointer-array

	successor	link
1	2	0
2	3	1
3	4	2
4	7	0
5	5	0
6	6	5
7	7	0
8	7	0
9	7	0

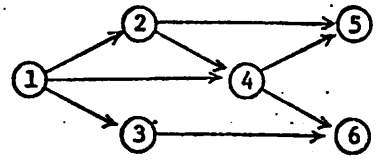
table

Fig. 11 Storage of program structure

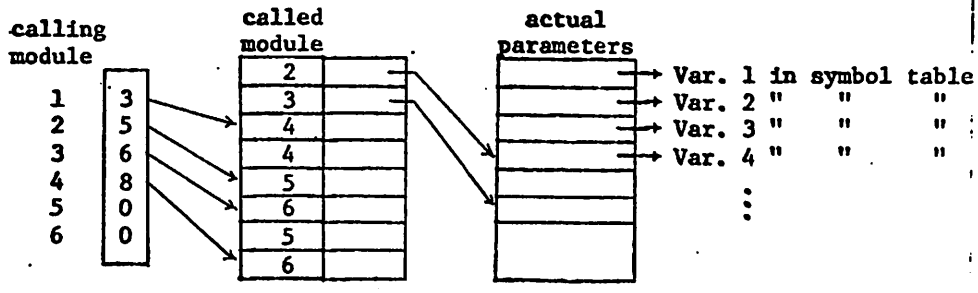
The successor table consists of a pointer-array and a table. The pointer-array contains an entry for each node. Each entry consists of a pointer to a chain of entries in the table. Such a chain represents all possible successors of the node. That is, a row in the table contains a successor node and a pointer for chain linkage. For example, an entry for node 3 in the pointer-array of Fig. 2 points to the chain of successors (6,5) in the table. This table together with symbol and symbol use tables are used as the basis for the extraction of structural characteristics of a program.

As mentioned earlier, integration of independently developed modules is the source of a large number of errors. A substantial portion of these errors can be detected by the static analysis. For this purpose, information on global storages and interfaces needs to be stored in the structural data base. Those information are typically stored in two tables, a common table and a module interface table.

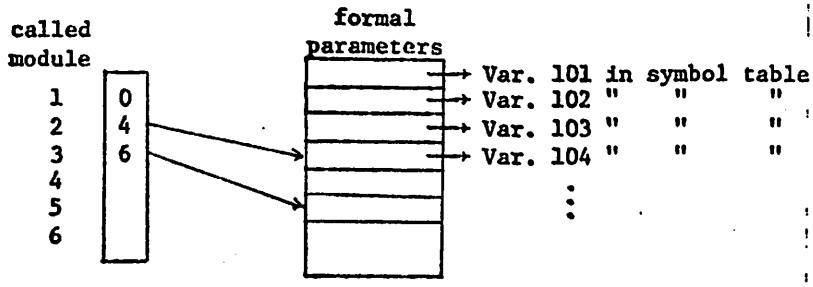
A module interface table consists of an actual parameter table and a formal parameter table. Fig. 12 illustrates a graph representing module interfaces and the corresponding module interface table.



module interfaces



An actual parameter table



A formal parameter table

Fig. 12 An example of a module interface table

This table together with others provide a convenient means for the detection of various structural flaws and semantic anomalies. It is a simple matter to check the consistency of types and numbers between actual parameters and formal parameters and to detect the recursive calling, etc. Moreover, they can be used for the determination of the affected areas when a few modules are modified.

A common table consists of a common storage table and a common variable table. An example appears in Fig. 13.

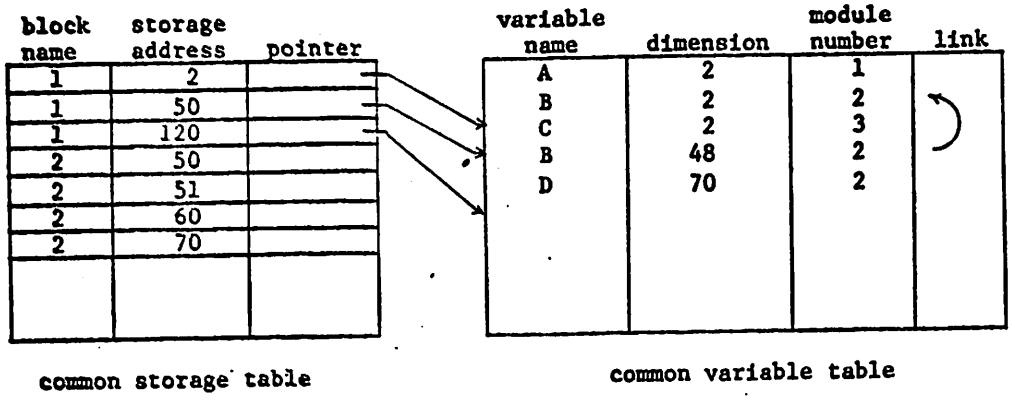


Fig. 13 An example of a common table

These tables are constructed by examining all declaration (especially COMMON and EQUIVALENCE) statements in the source program. The common storage table contains one row for each continuous and homogeneous segment of global storage. In Fig. 13, memories 1 and 2 in a block 1 are always referenced by the same variable, i.e., by A in module 1, B in module 2, or C in module 3. The third column in the common storage table contains a pointer to the set of common variables referencing to the storage segment. The common variable table contains information regarding variable name, dimension and module number for each common variable. An array type of common variable is sometimes divided into several rows in the table when the referenced common storage is divided into several rows in the common storage table. The common variable B of module 2 in Fig. 13 is an example. On the basis of these and other tables, procedures can be designed for the detection of misequivalencing, unnecessary declaration and inconsistency of variable type or dimension. It is also feasible to design a system for the optimal allocation of global storages [Bro 72b].

In summary, this data base provides not only an efficient and convenient means of retrieving information required by various validation procedures, but also a strong assistance in maintenance and modification of a program. Thus it is desirable to take the generation of the data base as an integral part of program documentation.

2.2.3.2 Structural analysis

The analysis of program structure is essential to the validation process

since it allows the detection of structural flaws and the identification of critical or interesting flow paths in the program. The data base contains necessary information for this analysis in a well-structured form. Included in the structural analysis are well-formation check, loop enumeration, path identification, and reaching and reachable vector generation, etc.

Well-formation check is a process of examining the program structure to see if there is any structural flaw. It includes the detection of unreferenced labels, unreachable statements and statements with no successors. These characteristics do not necessarily lead to the run-time error, but these are unplanned, undesirable and unreliable (error-prone) ones. Fig. 14 shows examples.

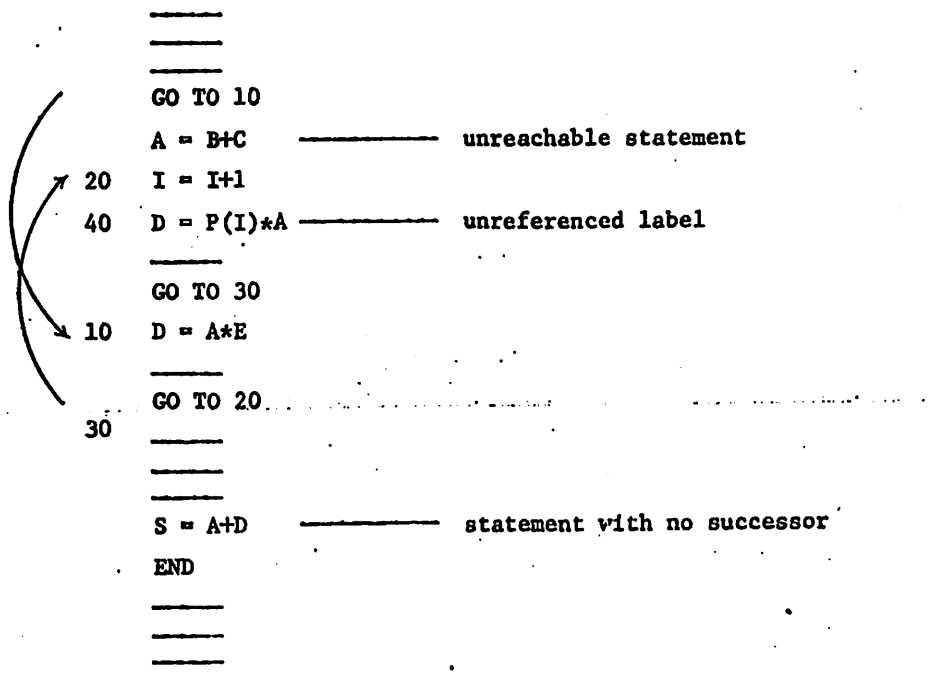
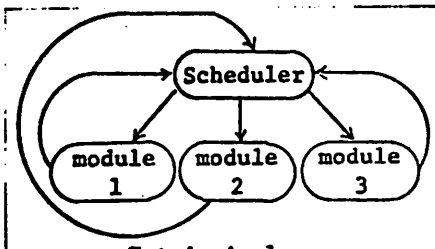


Fig. 14 Examples of structural flaws

Loop detection is performed by applying the procedure described in Appendix A to the program graph. An analysis of each loop characterizes the loop as intrinsic, deterministic or non-deterministic one. An intrinsic loop is the one which can be determined not to terminate by the static analysis. A loop is said to be either deterministic or non-deterministic according to whether the number of iterations can be determined by the static analysis or not. Thus the number of iterations in the case of a deterministic loop is apparently data-independent. Fig. 15 illustrates each type of loop.



Intrinsic loop

```
    DO 10 I=1,100
10  A(I) = B(I)*C(I)+D(I)
```

Deterministic loop

```
    READ 1,N
    DO 10 I=1,N
10  A(I) = B(I)*C(I)+D(I)
```

Non-deterministic loop

Fig. 15 Types of loops

After the loop detection, the reduced program graph (RPG) is generated and kept for the subsequent analysis.

A logical path in the program is represented by a path in the program graph. In general, a path may contain loops in it and two paths containing the same loop are considered as two different ones when the number of iterations of the loop is not the same for both paths. The number of paths in a large program is normally prohibitive, especially where the program contains a few loops. Therefore, a definition of an interesting path is adopted for the purpose of practical validation such that the number of paths becomes much reduced while no useful information for validation is lost by the use of interesting paths. There exist several approaches to the definition of an interesting path [Mil 74, Ito 73]. A typical definition [Ram 74b] which is also adopted in the rest of the discussion is either a path in an RPG or an interesting path in each MSC subgraph. A path in an RPG is a series of arcs from entry to exit. An interesting path in an MSC graph is defined as follows. A node in a graph is said to be essential if it is reachable from an entry node and an exit node is reachable from it. Removal of feedback arcs from an MSC graph produces the following two subgraphs. One consists of all essential nodes and all arcs between them, while the other consists of the remaining nodes, arcs and removed arcs. The former is called a forward subgraph and the latter is called a backward subgraph. Both subgraphs are then transformed into RPG's, respectively. Now an interesting path in an MSC graph is defined as a path in the RPG of either the forward subgraph or the removed subgraph. Fig. 16 illustrates interesting paths in an MSC graph.

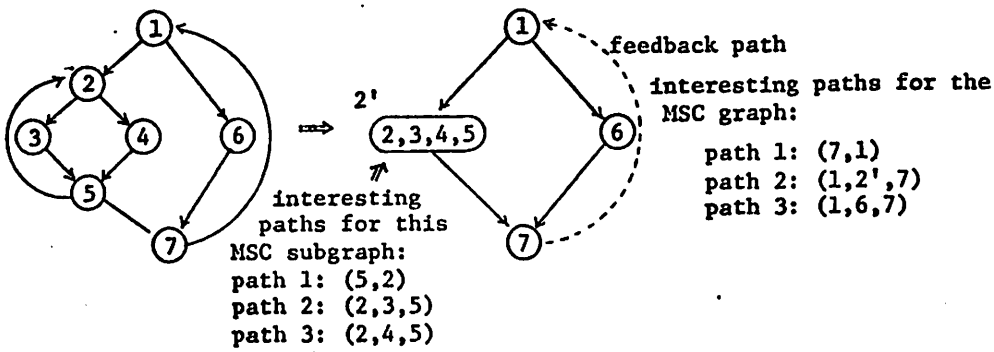
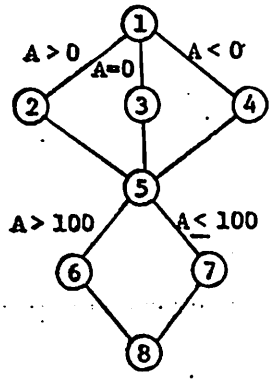


Fig. 16 Interesting paths in an MSC graph

Hereafter, an interesting path and a path are intermixed in use. This definition of a path is of iterative nature. Paths are defined in accordance with the level of abstraction. For instance, paths inside the strongly connected subgraph (2,3,4,5) in Fig. 16 are irrelevant to the definition of paths for the global MSC graph.

All paths in an RPG can be easily identified [Har 65]. In the case of an MSC subgraph, a procedure is applied to remove a few backward arcs and then both forward and backward subgraphs are identified. Thereafter, both subgraphs are reduced and interesting paths are identified by the procedure used for an RPG.

An additional feature, the detection of non-physical paths, is included in the path identification. A logical path is said to be non-physical if no inputs to the program can lead to the execution of the path. An example of a non-physical path is shown in Fig. 17.



non-physical paths:
(1,3,5,6,8)
(1,4,5,6,8)

Fig. 17 Examples of non-physical paths

Although the complete detection of non-physical paths is infeasible and it may involve an exhaustive process of logical inference, a substantial amount of non-physical paths can be detected by the static analysis and the detection is an important support to the dynamic analysis.

The structural analysis also includes the generation of reaching and reachable vectors for a specified set of statements. The reaching vector of a particular statement provides a list of those statements whose execution may lead to the execution of the statement in question. On the other hand, the reachable vector is a list of those statements which may be reached after the execution of the statement in question. This information can be easily extracted by manipulation of the program graph (i.e., connectivity matrix or successor table) as shown in Appendix A and by the consideration of non-physical paths.

2.2.3.3 Detection of vulnerable constructs

Since we assume that the program is at least free from syntactical errors, our concern in static analysis is in the detection of semantic errors or anomalies. This analysis will provide not only the running configuration of a program

to be used in dynamic analysis but also the guidelines for more cost-effective testing processes. The data base forms the basis for the efficient performance of this analysis. Included in this analysis are the detection of redundant statements, uninitialized statements, interfacing anomalies, undependable language constructs, etc.

A typical example of a redundant statement considered here is an assignment statement whose left-hand side variable never appears in predicates or in the right-hand sides of later statements. Analogously, uninitialized statements are the ones whose right-hand side variable never appear in input statements, subroutine calls or in the left-hand sides of earlier statements. These code-segments in a program are highly error-prone areas, though those constructs do not necessarily lead to run-time errors. A misspelling or mistake in keypunching often leads to these types of constructs. The detection of these constructs can be performed on the basis of the data base. Fig. 18 shows examples of these constructs.

```
N = 100
N = M+1          ———— uninitialized statement because of
                      a mistake in keypunching
_____
_____
_____
ERRORR = ERROR**2 ———— redundant statement
SUM = SQRT(ERROR)
_____
_____
```

Fig. 18 Examples of redundant and uninitialized statements

Interfacing anomalies refer to various semantic anomalies occurring from the integration of independently developed modules. The module interface table and the common table are effective supports to the analysis of these. Using this information in module interface, symbol and symbol use tables, it is a simple matter to detect mismatches in types and numbers between actual parameters and formal parameters as well as recursive calling. On the other hand, the common table provides a convenient basis for the detection of anomalies in global storage allocation such as misequivalencing, inconsistency of variable type or dimension declarations, and allocation of unnecessary storage.

Use of certain features available in the language often result in the degraded reliability of the produced program, though it may increase the execution efficiency. A computed GO TO statement in FORTRAN is an example. This type of statement depends on the value of a variable for determination of transfer locations. It often happens that the value of this variable exceeds the limit and possibly catastrophic transfers occur. It has been pointed out that even GO TO statements are generally harmful to the program reliability [Dij 68a]. Consequently, it is desirable to include in the AEVS a feature detecting these

vulnerable constructs and pinpointing those areas for the thorough dynamic analysis.

2.2.4 Dynamic Analysis

The dynamic analysis in automated evaluation and partial validation is a complementary process to static analysis. It is intended to verify various behavioral characteristics which remain unchecked by static analysis. It is basically a process of software testing consisting of driving the program with the devised test inputs and evaluating the outputs. As mentioned earlier, this process is greatly assisted by the static analysis. The static analysis provides information which can be used as guidelines for cost-effective testing. Dynamic analysis performs both validation functions, that is, error diagnosis and performance verification. Although both static and dynamic analyses participate in error diagnosis, performance verification is mainly achieved by dynamic analysis. A typical implementation of these validation functions takes the forms of program profile generation and diagnostic and performance testing. These are discussed in sequel in the following sections.

2.2.4.1 Program profile generation

The term program profile is used to mean a table of frequency counts which record how often each statement is performed in a typical run [Knu 70b]. In a more general sense, it refers to a collection of statistics on program behavior shown in typical runs. Information contained in the profile is typically execution frequency of each statement, execution time of each statement, frequency of successes on the logical test for each conditional branch statement, maximum and minimum values of instances of certain variables, frequencies of references to certain variables, etc. Fig. 19 shows an example of a program profile.

	Statements	Executions	Time	Successes
	DO 25 I=23,24	200	400	
	IF (CHAR(I).EQ.SPACE) GO TO 18	1354	4170	108
	DO 15 J=1,11	1246	2492	
	IF (CHAR(I),NE.SPCHAR(J)) GO TO 15	12344	61488	12112
	GO TO (100, 90, 70),J	232	464	
15	CONTINUE	12112	12112	

25	CONTINUE	1154	1154	

Fig. 19 Example of program profile

This program profile serves as an useful basis in various phases of software design. We discuss techniques of obtaining program profiles first and then justify the usefulness of the profile on validation processes.

The typical approach to the generation of a table of frequency counts is

based on the use of software counters automatically inserted by the system at appropriate locations inside a program. This frequency counter is an element of a more general class of software termed monitor or self-metric software. That is, the monitor or self-metric software refers to the program-segment inserted inside the target program and used as tools for obtaining execution characteristics of the program. Other monitors will be introduced as it becomes necessary in later sections. The physical implementation of a frequency counter takes the form of either a counter-incrementing statement or a call to the subroutine which in turn increments the appropriate counter. This is illustrated in Fig. 20.

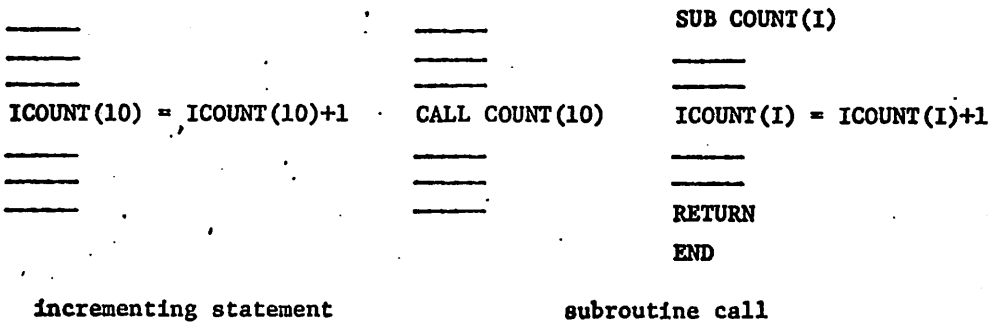


Fig. 20 Example of a frequency counter

A consideration must be given to the artifacts accompanied with frequency counters. That is, the effects on the program execution due to insertion of counters must be considered. If memory constraints or timing constraints are critical, the addition of a counter may cause unacceptable perturbations because of the measurement overhead or the increased storage requirement. In the case where a program is running on a computer system with a paged memory, the insertion of counters may lead to the different paging traffic.

With regard to this, it is desirable to insert a minimum number of counters sufficient for profile generation. A technique is available to determine a minimum number of counters and suitable locations for the insertion of them [Che 74]. It is based on the manipulation of a program graph.

The measurement of total execution time of each statement is based on the use of both execution frequency and estimated time for one execution of the statement. A reasonable estimation can be made by the syntactic analysis of each statement with respect to the number and types of operators, etc. Then, the total execution time is the product of this estimated time and execution frequency. It is also a simple matter to extract frequencies of successes on branches from the information provided by frequency counters.

Now that the generation of a profile is discussed, we consider the usefulness of program profile in regard to software validation.

First, program profile aids in diagnostic testing. It provides guidelines for an effective testing. In general, the most active or frequently executed

portions of a program are thoroughly tested while the less active portions receive inadequate testing. Program-segments with zero or low frequency counts could be given more attention in testing and singled out for early and intensive testing.

Second, profile often provides useful information for error detection. The statistics on branches and calls leave a record of what happened and often it is sufficient to indicate errors. The number of iterations of each loop is often useful for checking convergence of an employed algorithm.

Third, profile plays a significant role in performance verification. The examination of a profile, especially statistics on execution time of each code-segment, is often sufficient to check if the performance meets the requirement. Furthermore, it simplifies the improvement in the performance of a program. Since it is generally true that most of execution time of a program is spent in a relatively small portion of program code, portions with the high frequency counts can be designated as candidates for program optimization. Ingalls reports in [ING 71] that in a typical program only 3% of the statements make up 50% of the program's execution time. When either testing or examination of a profile reveals the unsatisfactory performance of the produced program, profiles can guide cost-effective strategies of program optimization.

Besides these, usefulness of program profile can be recognized in other phases of a program's life. Indication of good algorithms and sensitivity analysis of program performance to the change in the system environment could be supported by the use of program profile.

2.2.4.2 Diagnostic testing

As mentioned earlier, testing is regarded as a systematic process of error detection by means of exercising the program with test inputs and evaluating the outputs, while debugging is regarded as a process of error location and correction. However, testing can often assist error location as will be shown later. The complete testing refers to a testing with all possible inputs. It is a process too exhaustive to be practical in the case of a large program. Naturally, a more practical testing which establishes a sufficient degree of confidence in the reliability of a program becomes desirable. The common philosophy is to view the behavioral characteristics into a number of classes and then to verify each class to a practically sufficient extent. Structural characteristics recognized by the static analysis provides useful information for the decomposition of the behavioral characteristics which in turn supports testing strategies. This approach is to decompose the behavioral characteristics into a number of classes such that each class corresponds to one or a set of logical paths. Paths here are the interesting paths identified by the structural analysis in section 2.2.3.2. In any case, testing of each path in a program is a fundamental and

primitive operation. In order to manipulate paths, each path must be identified and then isolated whenever desired.

Isolation of each path can be performed in several ways. The most convenient one among them is to install and operate a new type of monitor. The concept of the blocking gate (BG) approach to the hardware diagnosis [RAM 71d] can be easily transported to the software diagnosis. A blocking gate (BG) is a device installed on the connection between two system elements, which blocks or unblocks the transfer of information under the control of a test driver. In the case of software, it is another kind of software monitor which blocks or unblocks the transfer of control between program segments. Blocking could mean an execution of STOP statement or a transfer to the test driving system.

By the same reasoning applied to frequency counters, it becomes clear that the minimum number of BG's capable of isolating every path is desirable. A technique is available to find such a set of BG's and suitable locations for installation [Ram 74b]. It turned out that the same locations can be used for installation of various other types of monitors useful to validation processes. These are introduced in later sections. The physical function of a BG depends upon the testing strategy and thus is discussed together with each strategy.

2.2.4.2.1 Test input generation

When testing is performed with randomly generated inputs, it is called the random-input testing. On the other hand, when test inputs to exercise a certain path or set of paths are devised either manually or by the system using the information provided in the course of design, the testing is called the synthesized-input testing. In the former case, input variables together with associated types are available from the data base and each input variable is assigned a random value of the specified type. It is often necessary to use some information provided by the designer even in this case. It is unpredictable which path will be exercised and thus the physical implementation of a BG takes the form of a code which transfers the control to the test driver when the path is blocked. This is illustrated in Fig. 21.

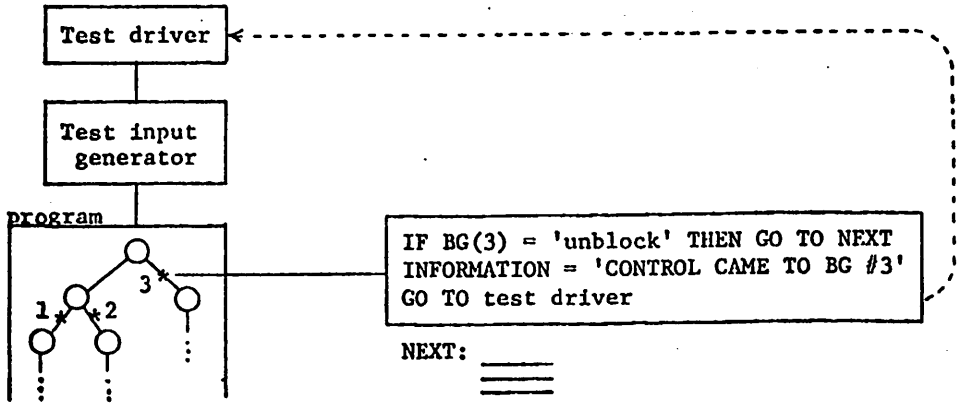


Fig. 21 A BG in the random input testing

In the latter case, it is known a priori which path or one of a set of paths will be exercised. The physical implementation of a BG in this case becomes a safety device. That is, it detects unexpected situations where the control gets out of the range of paths for which the inputs have been synthesized. This may occur due to either the incorrect synthesis of test inputs or the errors in a program. Therefore, a BG plays a role of the detector of both program and test design errors. Now we proceed to discuss several testing strategies based on the operation of BG's or other types of monitors.

2.2.4.2.2 Path-by-Path testing

The strategy of this testing is to test every interesting path at least once. That is, a set of test paths is a set of interesting paths defined in section 2.2.3.2. Besides this, several approaches to the selection of test paths exist. This is mainly due to loops, especially non-deterministic loops. One example is to define all test paths as a set of all paths in the program graph under the constraint that no path may contain more than a certain number k of iterations of a loop [Ito 73]. The suitable definition of all test paths should be determined with the consideration of the size of a program, the requirement in the degree of assurance and the amount of testing costs. When each path is tested, it is desirable to sensitize it since the overall testing becomes more systematic and cost-effective. That is, it is desirable to make it the only active path while all other paths are blocked. The simplest way of sensitizing a path is to block all BG's except the ones installed on the path to be sensitized. In addition, the BG's on the sensitized path may be transformed into other useful types of monitors such as out-of-bounds detector for interesting variables, etc. Either random inputs or synthesized inputs may be used. In the case of the random-input testing, the evaluation of test outputs does not easily lend to automation. Although, run-time checks facilitated in the system or software monitors installed on the path can detect various erroneous conditions, the manual inspection is inevitable in general.

On the other hand, the generation of synthesized test inputs normally includes corresponding outputs or criteria for determining the correctness of outputs. In this case, the validation of outputs becomes more amenable to automation and the speed of the whole testing process can be increased. The current trend in software design is to take the synthesis of test inputs and outputs as an integral step of the design process. However, the completeness of synthesized test inputs are hardly expected. It is quite probable that test inputs exercising a certain test path are missing. Moreover, the synthesis of complete test cases becomes generally infeasible in the case of a large program, though it simplifies the testing to a large extent. Therefore, the combination of both random-input testing and synthesized-input testing would be the most practical strategy.

There is one more obstacle commonly encountered in most testing strategies. It is a non-physical path. Although the detection of non-physical paths can be performed to a certain extent by the static analysis, the complete detection is not feasible in general. In fact, this is one of the factors obscuring the synthesis of complete test cases. The practical approach to the solution is to iterate random-input testing to exercise the interesting path and regard it as the non-physical one when the path is not exercised within a certain limit of time or iterations. The accurate decision whether it is a non-physical path is again subject to the manual inspection.

2.2.4.2.3 Test point insertion

In testing a subgraph, two approaches are possible. One way is to sensitize the path in the global program graph leading to the subgraph and then to sensitize each path inside the subgraph. Test inputs always enter through the entry to the global program graph. The other way is to install test points right before the entry to the subgraph and after the exit from the subgraph and then to enter test inputs through the first test point and evaluate outputs at the second test point. Although this method requires an analysis for obtaining input variables and output variables to be used in each test point, it could speed up and simplify the testing process.

In addition, the test point together with the segmentation can be applied to further simplify the testing process in a large program [Ram 71a]. This is illustrated in Fig. 22.

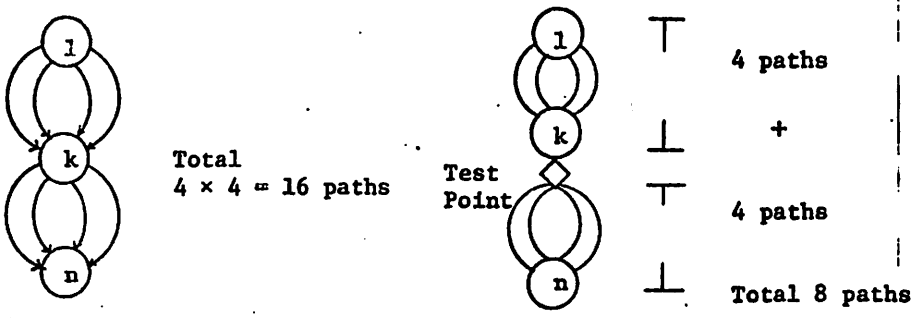


Fig. 22 Example of test point

That is, the number of test paths can be reduced by installing test points on the locations determined by the segmentation algorithm. The validation of the first segment will provide the legitimate values of the state vector which will be in turn used as test inputs for the validation of the second segment.

2.2.4.2.4 Other simple testing strategies

There could be almost an infinite number of testing strategies in addition to the strategies discussed in the preceding section. In this section, some strategies simpler than the ones already discussed are briefly introduced. The

common philosophy in those strategies is to take a suitable subset of interesting paths as test paths. The consequence is that the testing becomes less expensive though the assurance provided is also reduced. A typical one is to test only a minimum set of paths covering all arcs in the program graph. Such a set of paths is called a covering set of paths. Another is to test only a minimum set of paths covering all nodes in the program graph. When the software is supposed to have few bugs, these strategies become more cost-effective.

2.2.4.2.5 Error location

Upon the detection of errors on a certain test path, a process of error location and correction must be followed. This is the area where debugging resides as a host. In this section we discuss the usefulness of testing in regard to error location. The idea is based on the principle that the cross-section of two malfunctioning paths has a high probability of containing bugs. Therefore, the diagnosing aid system can be built in such a way that as soon as a malfunctioning path is detected, all paths crossing the detected one are identified and scheduled for testing. This mode of testing is called the cross-testing. The extent of debugging will be significantly reduced in this way. The software monitors embedded on the detected path can provide additional information useful to error location.

2.2.4.2.6 Path frequency counting

There is still another mode of testing. It is called stochastic testing. In this approach, a program is continuously tested with a sufficient number of randomly generated inputs. Test outputs are evaluated collectively at the end of the whole test run. During the test run of a program, frequency of traversal of each path is counted. Paths with high frequencies of traversals may be regarded as sufficiently tested, while paths with low frequencies may be taken for additional tests. In order to test paths with low frequencies, ones with high frequencies are blocked by BG's so that the testing efficiency may be increased.

In addition, the detection of non-physical paths can be achieved to a certain extent. Path-frequency counts can be contained in a program profile. The tool for path-frequency counting is another type of software monitor called a path-frequency counter installed on the same location as a BG. In other words, counters installed on same locations where BG's are resident are sufficient to count all path-frequencies. The detail is referred to in [Ram 74b].

2.2.4.2.7 Performance verification

Program profile discussed in section 2.2.4.1 provides a certain degree of assistance in performance verification. Based on it, the total execution time of a program can be measured and compared to the performance requirement. However, from the general nature of a profile, it shows only general tendency but

doesn't provide sufficient confidence in performance for various inputs. More thorough performance verification becomes desirable and it can be achieved to a certain extent by the testing with the sufficient number of test inputs. Whenever the testing is performed for error diagnosis, an additional check can be made if the execution time on the path has been within a certain bound. Once the path whose execution time exceeds the limit is detected, the execution-time profile can pinpoint the major candidate for optimization for that path. In addition, this combination of diagnostic and performance testing identifies a set of critical paths. Therefore, both error diagnosis and performance verification are performed interchangeably and implemented in a AEVS, an integration of diagnosing aid and evaluation aid systems.

2.2.5 Operational practices of AEVS's

In this section, the current status of AEVS's and the practical experiences in using those systems are briefly reviewed. There have been a number of reports on successes resulting from the utilization of AEVS's in the development and maintenance of various software systems.

The ACES [Ram 73a, Mee 73] contains features such as data base generation, thorough structural analysis, unreliable constructs detection, profile generation and critical variables monitoring. This system has been successfully used by the SAFEGUARD Systems Evaluation Agency as a gross survey of substantial amounts of program code. For example one partial process -- a small portion of the complete software system -- which was analyzed by the ACES, consists of 90 routines and subroutines containing approximately 23,000executable statements. Results showed unreliable practices such as computed GOTO statements with untested jump parameters, DO-loops with untested initial or final values of the loop parameters, and transfers of control into the middle of DO-loops. These conditions were further investigated by the user and either resolved or reported to the developer for modification.

In testing and maintenance of the Houston Operations Predictor/Estimator (HOPE) program, cost savings achieved by the use of the PACE [Bro 72a, 72b] was $8000 per year. The PACE disclosed that the existing test file consisting of 33 test cases covered 85% of the subprograms and that one-half of this number were exercised by almost every case. It required 4.5 hours of computer time and 35~50 man-hours of test results evaluation. Consideration of these statistics initiated the subsequent analysis to produce a more effective test file. A file of six cases was generated. These tested 93% of the subprograms, but they required less than 24 man-hours of test results examination. Similarly, the cost required in verification and retesting of the Automated Verification System (AVS) was reduced by $1000 per year by the use of automated tools. These are representative examples and similar reports are becoming more frequent.

The operating cost of the AEVS is worth receiving attention. The precise cost is dependent upon the organization and capability of the AEVS as well as the size and nature of the source program. Available statistics are very limited at present. The observation made during experiences of the ACES showed the general tendency that the construction of data base took approximately one and an half times as much as the compile time and the size of the data base was two and an half times as large as the size of the object code. Instrumentation of the program generally resulted in 20% expansion in the program size and the execution time.

A fully automated validation is beyond the capability of current AEVS's.Although it is premature to make a rigorous quantitative judgement on the basis of these examples, the increasing availability of,similar reports substantiates the prediction of more successes of future AEVS's on software validation.

2.2.6 Summary

In this section, we have examined features of currently available AEVS's. At present, the partial validation and automated evaluation appears to be the most effective approach to the validation of a large program. Although the absolute correctness cannot be proved by this approach, the degree of assurance obtained by the assistance of the sophisticated AEVS will be acceptable in most practical situations. The success and efficiency of this approach depends largely upon the approach in software design. The well conceived design process can simplify the validation processes to a great extent and increase the effectivenss in validation. On the other hand, the design process could become more efficient on the basis of powerful AEVS. The largest obstacle on the way to the fully automated validation has been the synthesis of test inputs. Problems encountered in the program correctness approach to the validation of a large program reappears in the automated synthesis of test inputs. Future works on structuring software design processes such that validation, especially the test input generation becomes highly amenable to automation will be of great significance.

2.3 Conclusion

The analytic approach to improve software reliability has been reviewed. The proof of program correctness approach enables us to validate many simple but frequently used algorithms. Hopefully, we can build up a library of validated algorithms which can be used to construct more complicated algorithms. The use of automated tools for evaluation and partial validation is an increasingly popular approach to improving both the productivity of the programmer and the reliability of the program. Although at present, a considerable amount of human judgement and manual labour are involved, the validation procedure can be much simplified if the program is prepared with the goal of reliability in mind.

3. The constructive approach to improve software reliability

We can see that the analytic approach has several disadvantages if the program is written without any consideration for its reliability. All the techniques developed become infeasible, ineffective and inefficient when the program is too large. The analytic approach is designed for detecting and correcting errors. There is no guarantee that the end product after extensive debugging is free of error, as Dijkstra pointed out, "Program testing can be used to show the presence of bugs, but never to show their absence." [Dij 69a]. There are no criteria to determine when our debugging effort should end. Besides it seems that debugging is a waste of effort on something (bugs) which should not be there in the first place. Why should we spend 45% of our effort (in debugging and testing) to get rid of the mistakes that we made in the first 55% of our work (in the design and implementation stages)? More care and time in design and implementation are clearly needed since it will not only reduce our debugging effort but also give us a more reliable program.

The constructive approach to improve software reliability has the objective of never finding the first error in the program. The design and implementation of the program are carefully and patiently performed, always keeping the reliability and correctness of the program in mind. Basically two approaches can be taken. A collection of programming techniques, called "structured programming", can be used to develop more reliable software by better design, management and coding methods. Programming redundancies, called software defenses, can also be introduced to the system to detect and contain error propagation in real-time systems.

3.1 Structured programming

Structured programming, a term mentioned so often these days, has been considered as a "major intellectual invention", one that can be compared to the subroutine concept and even the stored program concept. [McC 73]. However, no one really "invented" structured programming. A few people, especially Professor E.W. Dijkstra, have contributed a great deal to the formulation and consolidation of the philosophies of "reliable programming", which then become known collectively as "techniques for structured programming".

The term "structured programming" has been associated with many meanings in the literature due to the broad spectrum of techniques it encompasses. In some places, it has been associated with the syntax rules of a program, especially as a case against the GO TO statement [Dij 68a] and restrictions placed on the type of control structures that can be used to code a program. [Lis 71, Mil 71]. In other contexts, it has been used to denote a design method for reliable systems, the so-called "top-down approach". [Mil 71]. It has even been related to the management method called "chief programmer teams". [Bak 72a].

Dijkstra [Dij 68b] defines structured programming as "to construct his (the designer's) mechanism in such a way, i.e., so effectively structured, that at every stage of the testing procedure the number of relevant test cases will be so small

that he can try them all." It is therefore, a method of structuring the program so that it can be "exhaustively" tested and confidently verified. Baker [Bak 72a] defines it as "a method of programming according to a set of rules that enhance a program's readability and maintainability". Hence, it can be considered as a programming style for clarity. Mills [Mil 71] defines it as "a complex of ideas of organization and discipline in the programming process". Structured programming is then both a design methodology and a technique for coding programs such that the resulting software product is more reliable than an equivalent program developed using conventional methods.

3.1.1 Structured programming as a coding technique

3.1.1.1 "GO TO" - free programming

The enthusiasm in structured programming is often traced back to the famous letter from Dijkstra [Dij 68a], "Go To Statement Considered Harmful", in which he suggested that "the GO TO statement should be abolished from all high level programming languages" because "it is too much an invitation to make a mess of one's program". Dijkstra pointed out that it is the process controlled by the program that accomplishes the desired effects for a programmer. In order to minimize logical errors, one must "shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible". With the unrestraint use of the GO TO statement, it will become extremely difficult to trace the progress of the dynamic process evolving in time by examining the static program. Consider trying to understand a small algorithm (process) in the middle of a large program. If the algorithm has a conditional GO TO statement which transfers control outside the algorithm, then it is necessary to understand the effect of the code at the destination of the GO TO before the algorithm can be understood. This requires examining the effect of the external environment. If this GO TO leads to another and then another, the tracing of the external environment may eventually obscure all our understanding of the algorithm since the control or decision statements are separated in space on the page from the computations evoked from them. These jumps, sometimes in both forward and backward directions in the program, make it difficult to follow the logic of the program and difficult to visualize at any given point of the program what the present conditions are (such as the sequence of operations executed, the state of the variables, etc.) The program text does not correspond in space on the page listing to the execution of the program in time. Furthermore, as a program is debugged and changes are made to correct errors or to meet new specifications, the complexity of the program grows rapidly. Any change in an algorithm with GO TO's can have "side effects" in control on the environment in which it is used because this algorithm may jump out of its local environment and affect other parts of the program. New bugs are created due to these unanticipated side effects of the changes. If, on

the other hand, an algorithm has no GO TO statements, then the effect of the dynamic process created by the algorithm can be understood very easily as the cumulative effect of all its statements without worrying about the external environment of the algorithm (except for the state of the input variables). Therefore the dynamic process is "localized" in the static code. Any change in the code will only affect the "local" process. As a result, the user has more confidence in the program since it is readily readable and understandable. It is also very easy to modify, debug and maintain the program. It is therefore not surprising that Dijkstra remarked that the quality of programmers seems to be inversely proportional to the density of GO TO statements in their programs. [Dij 68a].

After all the discussions on the evils of the GO TO statement, one may still wonder if it is possible to write programs without them, and whether the replacement will create the same kind of problems. We would like to replace the GO TO statement with statements that will force the decisional statements to be associated with the computations evoked by them. Then the computational process evoked by the program execution (in time) will correspond more closely to the program text and becomes more easily understood. Bohm and Jacopini [Boh 66] have laid down the theoretical basis for structured programming by showing that it is possible to write any program using only three control structures. A program in this language will be a compound statement formed by simple assignment statements and predicates* according to the following rules:

1. If S1 and S2 are statements, then the concatenation of S1 and S2 is a statement. (SEQUENCE).
2. If S1 and S2 are statements and P is a predicate, then the conditional statement IF P THEN S1 ELSE S2 is a statement. (IF THEN ELSE).
3. If S is a statement and P a predicate, then the iterative statement WHILE P DO S is a statement. (DO WHILE).

A program written in this language will have a flowchart made up only of the single-entry single-exit structures as shown in Figure 23. Each block in the figure may be replaced by one of the three structures. Therefore the control structures can be nested.

Programs written in this block structured programming language has a very simple control structure. There are no GO TO statements and no labels. There is a direct correspondence between the static form of the program and the dynamic flow during its execution. Using only concatenation, alternation and iteration as the control structures, the process is "localized" with the flow of control in the program. The computations evoked by a decisional statement can be closely followed.

---

* A predicate is defined as an logical expression which when evaluated will yield a value which is either TRUE or FALSE.

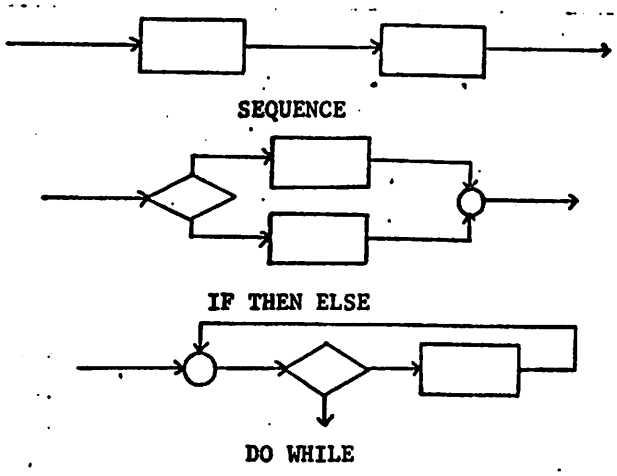


Figure 23 Control structures for structured programming

There is no back-tracking. Without GO TO's, transfer of control always proceeds unidirectionally. A structured program can be broken down into meaningful segments which have only one entry and one exit. Execution always proceeds from the single entry point to the single exit point of a subprogram (block). This simplification of control makes it no longer necessary to flowchart a subprogram. In fact, as a general rule of thumb, a structured program which cannot be understood without flowcharting is too complicated and should be broken down into modules. The straightforward control transfer in structured programming is also very helpful for proving program correctness. The proof of the correctness of a program which does not contain GO TO's becomes much simpler since the termination of the program depends only upon iteration statements (not upon a possibly infinite transfer of control). If a block of code contains a GO TO statement, we have to examine, understand, and prove the termination of the block of code to which the GO TO statement transfers control. A chain of GO TO statements will make the understanding and proof of termination very difficult. If, however, we restrict the program blocks to be executed sequentially or at most in an iterative fashion, we can explicitly state the conditions under which a block of code terminates. Without GO TO statements, the proof of a program breaks down naturally into the proof of separate program components. Also the proving process is much simplified because the program is clearer and easier to understand. Each method of combining the simple assignment statements corresponds to a rule of inference. Concatenation is understood by enumeration, conditional statements by case study and iterative statements by mathematical induction. Programmers are familiar with these rules of inference. [Lis 71]. Therefore, it becomes feasible to prove, at least informally, the correctness of a large program. Dijkstra by using structured programming has developed and informally proved an entire operating system. [Dij 68b].

3.1.1.2 Objections to structured programming

There are, however, some programmers who question the merits of such restrictions imposed by structured programming. Objections to structured programming usually come from one of two sources: basic programmer conservatism and concern about efficiency of the programs produced. The conservative reaction comes about because structured programming is a new technique which may be more difficult to learn and to use than conventional programming. It will also require a change of programming habits, which may affect the software productivity of the programmer. However, the actual degree of difficulty may be overestimated. At the University of California at Santa Clara, only structured programming is taught. Their experiences indicate no unusual problems in teaching and learning structured instead of unstructured programming. Experience reported by Baker [Bak 72a] on the development of the New York Times Information Bank showed that software productivity of a programmer is increased slightly rather than decreased by using structured programming. Besides, there is always the advantage of a reduction in debugging time.

There is also the concern about the efficiency of the program in terms of execution time and memory storage required. Knuth and Floyd have discussed techniques to avoid GO TO statements in a program by using recursive procedure method and by duplication of code. [Knu 70a] Both of these methods will cause an increase in memory storage and execution time. Ashcroft and Manna have shown that every flowchart program can be written without GO TO statements by using WHILE statements. [Ash 71]. Given a set of inputs, the WHILE program will produce the same set of results as the original program but need not perform the same computation sequence although the topology of the original program is preserved. However, new variables are introduced to preserve the values of certain variables at particular points in the program or alternatively special boolean variables are introduced to keep information about the course of the computation. In general, it is always necessary to add extra variables in order to translate a flowchart program into an equivalent WHILE program. Therefore, it may mean an increase in memory storage. However, Ashcroft and Manna have reported the same order of execution time efficiency for the structured WHILE program. It is still unclear if there will be any inefficiency when the program is flowcharted with structured programming in mind. Dijkstra feels that structured programs are just as likely to lead to efficient code as any other type of program. He also feels that an increase in efficiency always comes from an exploitation of program structure. [Dij 65]. Furthermore, structured programs will be written in high level languages and a powerful optimizing compiler can be used to produce efficient code.

3.1.1.3 Other considerations for a structured programming language

Structured programming is a technique that reduces a program's complexity, increases its clarity, and results in easy understanding and maintenance. A reliable program should have very simple structure and its structure should be clearly

visible by an examination of the code. To achieve these goals with the mere elimination of the GO TO statement appears to be a simplistic approach. Reducing a program's complexity can be thought of as a process of removing obstacles from the program: complicated control paths, obscure structures, uninformative comments, unnecessary jumps, redundant and obsolete code, ambiguous constructs, etc. Restricting the programmer's use of control structures to SEQUENCE, IF THEN ELSE, and WHILE may also lead to unnecessary inconvenience sometimes. Different forms of restricted use of 'GO TO' have been proposed, such as the EXIT statement in BLISS [Wul 71] and the COME FROM by Clark [Cla 73]. It has been suggested that the CASE statement (as used in ALGOL) be added to the allowable control structures. [Mil 71]. It is also suggested that the single-exit law be relaxed for abnormal termination. [Don 73]. Improving program clarity can be thought of as a process of adding things to the program: meaningful names, informative comments, clear code layout and indentation for readability, more levels of modularization, good documentation, clean interface, etc.

It is clear that structured programming can be achieved with a combination of good programming style and language design. The drawbacks of existing programming languages have been investigated by Elspas et al. [Els 71], and Kernighan and Plauger. [Ker 73]. A good language for structured programming should not contain features that are conducive to error. It should encourage concise expression rather than cryptic. A language like APL is an open invitation for clever tricks which are very difficult to understand; even by the programmer himself after some time has elapsed. The language should have a rich and descriptive syntax, making it very easy to read and understand, even by people who are not familiar with the language. For example, in APL,

```
→ 6 x i(x > y)
```

means "IF x > y, GO TO 6". However, a statement

```
→ 6 x (x > y)
```

means "IF x > y, GO TO 6, ELSE RETURN". The hidden "RETURN" is often overlooked. Irregularities of treatment of the same syntax construct in different environment is another drawback of many languages. For example, in Fortran, there are different constraints on the integer expression, according to whether they appear in a DO statement, an I/O statement, a subscript, a computed GO TO, a declaration, etc. A programmer would much prefer only one integer expression usable anywhere. Some other irregularities are provided in order to save the programmer some keypunch time. However, saving a few characters can sometimes create a lot of confusion. Algol 68 allows the statement

```
label;
```

as a legal branch to be interpreted as "GO TO label". These irregularities should

be removed. The language specification becomes bigger, and thus enlarges the compiler size. The cryptic expression makes error checking very difficult. Also these special cases may be treated differently in different installations, thus affecting the transportability of the program. Therefore a good language should encourage uniformity and generalization. Algol 68 has some good features in this aspect. Statements and expressions are treated as the same thing in Algol. The CASE statement encourages a uniform organization of the programmer. Program layout and commenting can affect the readability of the program drastically. A language should allow free-format input and proper indentation. The compiler should also improve the readability of the program by providing optional informative outputs of the program layout besides the standard listing. This may be a listing of the program with information about its topology, such as the loops and the branches. Comments are crucial to the understanding of a program. However, too often uninformative comments are written by the programmer. It may not be a bad idea at all to design a "structured" language for commenting a program. Comments written in the form of assertions used in proving program correctness may be useful in understanding the program.

A good language for structured programming should also encourage a programmer to write reliable programs, even at the expense of additional constraints on his style. This affects the syntax of the language, its semantics, and even the pragmatics of implementation. The language syntax should be descriptive of the desired action. Language redundancy can also provide error protection. The requirement of the programmer to declare all program variables and the way they are used in Algol 68 helps to reduce errors due to misspelling of identifiers. During program execution, array-bound checking should be provided by using special hardware (as is done in the B5000) or by run-time system software to verify that subscript values do indeed fall in the declared range. Descriptive and meaningful names should be used and clarity of expression should be emphasized. The compiler should also perform some semantic checking on the program to reveal semantic errors, which may be very helpful for debugging.

Therefore we can see that a good language for structured programming should support the three types of well-structured control structures: SEQUENCE, IF THEN ELSE, and DO WHILE. Unrestricted use of the GO TO statement should not be allowed. It should encourage concise and clear expressions, clear code layout and indentation for readability, and informative comments. It should have a rich and descriptive syntax, uniformity in language constructs, and clear precise semantics. It should contain error reducing properties, such as language redundancy. The implementation of the language should try to improve the reliability of the program. No existing programming languages have all these properties. PL/1 and Algol can support the three basic control structures. Therefore they can easily be used for writing structured programs by eliminating the features that are conducive to error. Other

modifications of their syntax and semantics for more reliable programming are also very desirable. Kelley [Kel 73] has developed an experimental programming language called APLGOL which adds structured programming facilities to the existing framework of APL. The conventional semantics of APL is unaltered and only minor changes are incorporated in the syntax.

A good program design can also help to produce reliable software. Many program errors can be avoided by writing the code first in some "virtual language" and then expand and translate into the desired high level language. This two-step coding procedure will increase the reliability and intelligibility of the resulting program, besides helping the programmer to write informative comments and useful documentation. The "virtual language" need not be formal but should be precise and descriptive of the action to be performed. For a large programming system it may be advantageous for the programmers to agree on such a "virtual language" so that uniform documentation can be provided, making the programs easier to understand by all programmers. In a way, this can be considered as similar to the top-down approach of design of structured programs to be discussed in the next section.

3.1.2 Structured programming as a design technique

We notice that structured programming aims at simplifying the control paths of the program so that it becomes more readable and understandable. However, even when a program is well structured, it may still be very difficult to understand if it contains DO loops with thousands of instructions and IF-THEN-ELSE statements taking up twenty or thirty pages. The program has to be divided up into smaller subprograms of more manageable size called program modules, a common practice called modularization. Modularity allows modules to be written, compiled, and tested independently. Traditionally the process of modularization is performed in a careless and arbitrary fashion. The division of a program into modules is usually done according to the boxes of the flowchart of the program. This may work in small programs. In large programs, there are complicated interactions among the modules. Modules interact in control (via the entry and exit points), in data (through shared data or arguments passed between them), and in the service which they provide for one another. An arbitrary modularization may obscure many of these interactions (interaction complexity) so that subtle software bugs may be created. It may also introduce unnecessary functional complexity by putting too many functions in a module or by failing to abstract a common function shared by several different modules.

From these considerations, we notice that in a good modularization, we should minimize the assumptions that the modules make about each other (to reduce interaction complexity) and we should also limit the size of the modules (to reduce functional complexity). Parnas [Par 71] has also made other suggestions on the modularization of the program. The modules should be defined around assumptions which are likely to change. In specification of modules, we should specify

identities or relations between the externally visible aspects of the module rather than the internal construction.

In terms of structured programming, good modularization can be achieved by two techniques. The first, levels of abstraction, [Dij 68b] allows us to resolve the complexity of the system by conceiving the system as a hierarchy of levels of abstract machines. The second, top down design, [Mil 71], enables us to develop a large program as a tree structure of functional modules.

3.1.2.1 Levels of abstraction

Levels of abstraction was first proposed by Dijkstra for the design of the T.H.E. Multiprogramming System so that it can be proved logically correct and tested exhaustively. [Dij 68b]. The system is designed as a hierarchy of levels of abstract machines, the lowest levels being those closest to the machine. At each level, the abstract machine allows us to understand the operations at that level without requiring the detailed knowledge of how the operations are carried out. For example, the virtual memory can be considered as a level of abstraction while the physical memory is a lower level of abstraction. Two rules are used for the formulation of levels of abstractions. Each level owns resources exclusively for its own use and these resources are not accessible from other levels. Lower levels are not aware of the existence of higher levels and therefore may not refer to them in any way. For example, the disk and core are resources owned by the physical memory level while pages and segments are resources in the virtual memory level. The physical memory is not aware of the existence of the virtual memory.

Each level of abstraction contains a collection of related functions. Operations in each level are interpreted by the abstract machine on the next lower level. Higher levels are supported by lower levels. Therefore, high levels may obtain service and information from lower levels. Each level may contain externally accessible functions in addition to internal functions used exclusively to support the level. Since each level has its own resources, each level can be considered as a level of abstract resources. The division of resources into levels implies that each level has to be 'complete'. The operations at one level have to be supportable by the abstract resources provided by the underlying levels.

The hierarchy arrangement of levels of abstraction allows us to design good modularization of the system. Subtle errors due to shared resources are controlled by treating the ownership of resources in a rigorous fashion. The interface problem is reduced by defining system primitives which must be used for communications between levels. The cooperation of processes are regulated by a set of formal synchronizing primitives. However, the success of the design of the entire system depends critically upon the design of the top level. The design of this level depends on the experience and judgement of the programmer, as well as his understanding of the system. Dijkstra suggests that, in general, decisions should be delayed as long as possible (hold onto the abstractions as long as possible).

Whenever possible, we should gain more understanding of the system before we make a decision. If a module is too large, the principle of "divide and rule" can be applied to decompose it into smaller modules. Besides the program should be designed for adaptability by considering potential generalizations at each stage in the design. This helps us to gain insights into the structure of the system. Specifications are likely to be changed while the system is being built because of more understanding of the system. Modifications are always necessary after the system is put into operation for system optimization and tuning, as well as user convenience. Therefore, structured programming should also be used for implementing the system for maintenance and modifications.

Dijkstra also gave some design rules for the specifications of the modules. [Dij 65]. His "principle of non-interference" states that modules should be constructed to satisfy specifications so that they are independent of each other and independent of the context in which they will be used. The modules are logically independent so that they can be designed and constructed independently. Independence implies that all interfaces have been defined and that all conflicts over resources have been resolved. When the modules are integrated together, the correct working of the system can be established by considering only the exterior specifications (an abstraction) of the modules without requiring knowledge about the interior construction. Therefore, starting from the lowest level, at every stage of intergration, the correctness of the system can be proved by an exhaustive case analysis. Dijkstra concluded that a designer should structure his program so that the number of relevant test cases is so small that they can be exhaustively tested. [Dij 68b]. Besides the T.H.E. System, there are other systems constructed with levels of abstraction, such as the Venus system designed by Liskov [Lis 72a] and the file system designed by Madnick and Alsop. [Mad 69].

3.1.2.2 Top down design

There are basically two approaches to build a system: from the bottom up or from the top down. In the bottom up approach, implementation begins after an initial design which identifies the tasks. The most elementary (low-level) functions are implemented first and then used as building blocks to compose more complicated tasks, and so on. In this way, debugging of the code is easier and can be performed in parallel with the design of more complicated components. However, there are several dangers with this "building block" approach. The building blocks are implemented before the system is well-defined. They may not be the most useful components in building the system. Modifications of these building blocks are frequently necessary when difficulties are encountered in the higher levels. System integration is difficult because the interfaces between programs are not rigidly defined and modifications of building blocks often create subtle side-effects. Worse than all these, the existence of these building blocks may influence the

system design. Therefore, the design of the system is constrainted by decisions made before we have an overall understanding of the system.

The top down approach uses the opposite philosophy. [Mil 71]. The highest level, which represents a rather formal description of the overall system, is specified first. It describes the flow of control among the major subsystems, each having a functional specification. Each of these subsystems is then expanded into an intermediate system of code and functional subspecifications. This process is carried on until all functional specifications are coded. Therefore the system is organized as a hierarchy of levels of function specifications. (Note that the levels used here are different from levels of abstraction because they are not associated with the ownership of resources.)

There are several disadvantages associated with the top down approach. The specification of the components are rigidly defined, including the data structures it employs, without much consideration to how the components will be implemented. This may lead to problems of inefficient implementation. The design may also be complicated since the system is very complex and it may be difficult to write down all the specifications at each level. To understand the operation of the system, one may have to simulate the system as the design proceeds in order to debug his specifications, as suggested by Randell. [Ran 69].

H. Mills, an advocate of the top down design, showed that most of these problems can be overcome by the introduction of structured programming. [Mil 71]. Mills views the functional specifications as similar to mathematical functions which map initial data into final data for some codes yet to be specified. The whole organization is based on functional programming, defining composite functions in terms of other functions. The design structure is carried out directly in code, which can be at least syntax checked, with "program stubs" representing functional subspecifications. This process of functional expansion is carried on, with new functional subspecifications represented by names of dummy members of a programming library, until the whole system is defined. Each functional subspecification, called a segment, may consist of a mixture of control statements, and macro calls (to lower level segments) with possibly a number of initializing, file or assignment statements as well.

Mills also put other restrictions on the construction of a segment. Only structured programming techniques can be used, implying that the control structure will only consist of sequencing, IF-THEN-ELSE, and DO-WHILE. The size of the segment is limited to about 50 lines of code or one page of listing, so that each segment is small enough to be readable and understandable. A segment is also restricted to have only a single entry and a single exit. Therefore, a segment behaves as a simple data transformation function independent of the environment in which it is used so that there are no side effects in the program control. Segments are stored under symbolic names in a library and are substituted at any point in the

program by a macro-like call. The segments form a tree structure with the system specification at the root. The system is written from top down and at each level we can verify that the intermediate system is logically equivalent to its predecessor system. Therefore, the system can be verified to be correct one level at a time by functional expansion up to the lowest level, i.e., the code of the program. Since each segment is small and well-structured, the proof of correctness is much simplified. Interfaces between segments are rigidly defined, minimizing the interface problem. Documentation is automatically provided by the functional specifications and the verification procedure. The finished system also contains traces of the design process, which is very helpful for the maintenence and modification of the program. It is also possible to execute the system at any intermediate level by "simulating" modules that are not yet implemented. The modules are never checked out independent of the system. Therefore, conflicts over resources are detected and resolved early in the design process.

Mills' approach of system design has been carried out by Baker [Bak 72a] in the design of an information bank system for the New York Times. It was reported that programming productivity was substantially improved and the system had no serious errors for the first twenty months.

3.1.3 Structured programming as a management technique

Reliable programs cannot be produced efficiently without a good management policy. Communication problems among programmers are the chief source of program errors. Conventional management approaches often suffer from the lack of functional separation, communication, discipline and team work. The hierarchical arrangement of a structured program provides a natural organization for the assignment of jobs. The communication problems are minimized by rigid specifications of the components and the interfaces. All these give us an opportunity to use a more systematic approach of management. The Chief Programmer Team approach has been proposed by Mills [Mil 73] and Baker [Bak 72a] as a way to improve the manageability, quality and productivity of programming. The nucleus of a chief programmer team consists of a chief programmer, a backup programmer, and a programming secretary. Other personnel can be added at the discretion of the chief programmer. The main objective is to structure "programming work into specialized jobs, define relationships among specialists, and stress discipline and teamwork". [Bak 73].

The chief programmer is a technical manager whose principal work is to design and code central, critical segments of the system and make specifications of programs to be assigned to other programmers. Besides, he also reviews and then integrates programs coded by other programmers. The backup programmer is another person who is completely familiar with the design and development of the program by working closely with the chief programmer. He reviews decisions and provides test planning for the system. He also formulates programming strategy and tactics, relieving the chief programmer to concentrate on the central problems of

system development. Therefore he is both an assistant and a back-up man for the chief programmer.

The programming secretary is responsible for maintaining the current status and previous history of the project in the Development Support Library (DSL) in both an internal (machine readable) and an external (human readable) form. [Bak 73]. The DSL contains all the project programs and data files in the computer and all the project documentation, listings, and outputs, including test runs, whether successful or not. A detailed history of the development of the program is kept. The programming secretary has to collect from the programmers the project notebooks containing changes to be made in the internal programs and data files. Then he prepares the input and excutes the project programs on the computer, with the help of keypunch operators. The machine executes the program while updating the library data in the internal library file. The secretary obtains the output and enters them with the new source listings in the project notebooks of the external library, with the necessary documentation. The outdated documents, however, are not destroyed but logged in chronological journals. He then returns the notebooks to the programmers. Therefore the programming secretary is also a key personnel in the nucleus. He releives the programmers of most of the clerical and secretarial work for maintaining all project records, current status, and test data so that they can work more effectively and efficiently.

The DSL represents a concept of moving the programming production process from private art to public practice. All computer runs and program data become public assets and the visibility of the DSL simplifies the communication problem among the programmers. The record of the history of project development facilitates the maintenance of the program. The concept of "egoless programming" is also adopted. [Wei 71]. The chief programmer has to read, understand, and verify all program data developed by other programmers on the team. They, in turn, have to do the same on programs written by the chief programmer to define the specifications and interfaces. This ensures that at least two programmers fully understand every line of the developing program.

About 100,000 lines of source code seems to be the maximum size properly assigned to a single team. In really large programs, we have to define a hierarchy of chief programmer teams. A team of skilled programmers may start out the system with the overall system design. After the design is completed, each member in this team may become the chief programmer of the other teams responsible for the next level of design and implementation, and so on. The evolution of assignment in a top-down fashion will retain the spirit as well as the discipline of the Chief Programmer Team.

3.2 Software defenses

Although structured programming can help us to construct reasonably reliable software we are still not certain that no critical error will ever occur. In some

real-time systems, even this low error rate cannot be tolerated. A protection technique, called software defense, [Cha 73] can be used to trap and contain the propagation of software errors in a real-time system. This technique is a precautionary procedure to make sure that there will not be a catastrophic disaster even when a software error occurs. These techniques are highly goal-oriented so that it is very difficult to generalize them. They have been proved to be very useful in the ESS of the Bell Telephone Company [Cha 73].

There are two types of software defense techniques: defensive programming methods and audits. The former includes techniques used in the design of the program and data to detect software errors before they cause system misbehavior. The latter are used to detect, contain, and possibly correct software errors after they have occurred. Since audits are used primarily to protect the integrity of the program, they will be discussed in the next section.

Defensive programming may include a variety of methods. They are special precaution procedures to be implemented in the program to reduce the possibility of a software error. They are dependent on the purpose of the program and the style of the programmer. A commonly used technique is the range check. Range checks can be performed on the values of data, memory locations accessible to a program, and areas where a program control can be transferred to. A state check to verify that the system indicators and the actual states of the resources are in agreement before a resource is allocated can reduce many system errors and mutilation of potentially valuable data. Reasonableness checks on the input data can eliminate many system misbehavior due to abnormal input data. A reverse check is also an effective tool to ensure the correct operation of the system. For example, when a complicated procedure is employed to search a file, we should examine some characteristics of the file searched before any modification to make sure that we are operating on the correct file. Whenever a translation is performed, a reversed translation can also be done as a check. The defensive programming techniques can be viewed as software redundancy to improve the reliability of the program. The degree to which these techniques should be applied must be considered carefully in order not to degrade the efficiency of the program significantly.

In a multiprocessor system, software defense can be provided by the architecture of the operating system. The operating system functions can be distributed among the processors. The design of an operating system with distributed intelligence enables us to achieve a fail-soft behavior in presence of a software error. The operating system of the PRIME System is an example.

PRIME, developed at the University of California, Berkeley, is an experimental time-sharing system designed for continuous availability, data privacy, and cost effectiveness. [Bas 72]. It is a multiprocessor system in which one processor is dynamically designated the job of the control processor and the rest problem processors. The technique of dynamic verification is used in the construction of

the operating system to ensure continuous availability and the data privacy of a user even in the presence of a single hardware or software fault. [Fab 73]. Furthermore, multiple faults will not lead to unreliable operation unless they reinforce each other. Dynamic verification of a decision implies that every time the operating system makes a decision there is a consistency check performed on the decision using independent hardware and software. This technique is applied in the control monitor of the operating system. The control monitor performs the functions of scheduling of processes to be run on problem processors, the allocation of memory pages and disk cylinders to processes, and the management of a virtual communication system. It consists of two parts, the central monitor (CCM) and the extended control monitor (ECM). The CCM is written in a high level language and is executed only by the control processor. The ECM, resident in the problem processors, is microcoded and acts as a local representative of the CCM to enforce its decision. However, dynamic verification is possible because the CCM does not interact directly with the ECM but rather by sending messages to the ECM. Each time a decision is made by the CCM, such as starting a process sending a message to a process, or allocating a resource, the ECM can verify if the action of the CCM is appropriate. Inter-process communications are performed through messages and are similarly checked by the ECM's. The decision making and decision checking processes are performed by different hardware and using different algorithms. Therefore, the integrity of the system is maintained in the presence of a single hardware fault or software error. Such decision verification procedures can also be applied to other software architecture with distributed intelligence.

3.3 Conclusion

Several programming systems of considerable size have been developed using the constructive approach, notably the THE System, the information bank of the New York Times, etc. All these systems have shown to be very reliable after they have been put into use. For example, the information bank of the New York Times had been put into operation for 13 months before the first error was detected that resulted in system failure. The acceptance test took a total of 9 weeks and only 21 errors were detected, all of which were fixed in one day [Boe 73]. The productivity of the programmers is high, 83000 lines of high-level language source code produced in 11 man-years (6 men and 22 months). The reliability of the program is high, with only 25 errors in over a year's operation. This corresponds to approximately one error for each 5 man-months of effort on the project, which is quite remarkable. [Bak 72b]. As a result of using the constructive approach to reliable programming, the project cost was cut by 50% and development time was reduced to 25% of the initial estimate! [Boe 73]. The mission simulation system developed for the Skylab operations by IBM has similar success. 400,000 lines of code were produced in 2 years and the software was delivered on the original schedule in spite of 1,200 formal changes in the requirements. [Bak 73].

The constructive approach therefore appears to be a very useful way of designing and implementing reliable programs. However, there are still people who are skeptical of its success. The systems constructed by this approach so far are relatively small and simple compared to really large programs like the national missile defense programs. The number of programmers involved are small, six full-time programmers in the New York Times Project and six half-time programmers in the THE System. They are experienced, well disciplined (mostly mathematicians with 5 to 8 years of university training in the THE System), and under excellent leadership (Mills, Baker and Dijkstra). It may be doubtful if the same remarkable success can be achieved in a large programming project involving, say, 2000 relatively unexperienced programmers. Besides, most of the theory developed in this area are guidelines and principles rather than procedures which a programmer can follow step by step to construct his program, especially in the design stage. Many of the tasks have to be performed manually and decisions have to be made arbitrarily without any methodology to evaluate them before, or even after, they are made. The design of the system depends as much on the experience and judgement as on the intuition of the programmer. In general, we know what the end product should be without too much idea on how to arrive at it. For example, Dijkstra suggested that the program should be structured in a way so that the number of relevant test cases at each stage of testing will be so small that they can be exhaustively tested. However, there is no general method that will enable us to arrive at this end product. There may even be cases where this is impossible, when the decisions are tightly "interwoven" together. In the design of the system with the "top-down" approach or the "levels of abstraction" approach, it is difficult for us to decide how to form a "complete" level. Besides, the decomposition procedure of large modules is done in an arbitrary fashion. When difficulties are encountered in a level, it may be necessary for us to go back and modify the higher levels. Fortunately, the structural programming approach makes such modification easier.

It is highly desirable for us to be able to mechanise some of the procedures so that automated tools can be used to help us to design, implement and test the program. Computer assistance in validating and evaluating our decisions during the development stage is clearly very valuable. Obviously, such tools are still necessary for the end product. Therefore, it seems that the analytic methods to improve program reliability are still essential for assuring the quality of programs developed by the constructive approach, when the program is too large and complex.

4. Integrity of a program

4.1 Introduction

By now we have already surveyed different methods to construct reliable software. Reliability implies the ability to perform a specific function by the piece of software. However, the correct operation of the process created by the piece of the software cannot be achieved if its integrity cannot be guaranteed. Integrity

is a particularly serious problem in large real-time systems since the program is controlling an on-going physical process such as a nuclear reaction, air traffic control system or a national anti-ballistic missile defense system. In many cases, the security of a program is as important as its reliability. Loss of integrity is usually associated with malicious tampering of the code of a program by an unauthorized intruder in a hostile environment. This is not necessarily the case. Loss of integrity may be caused by a subtle software bug inside the program itself. Modification of code by another user may be unintentional, due to a flaw in the operating system. Transient hardware faults can also cause tampering of codes which are very difficult to detect. Real-time systems are especially vulnerable to intrusions since they have to be on-line and accessible to a large number of users. This makes protection quite difficult. Things are even more complicated in a multi-processor system since processes are created and destroyed in real-time and they may even co-operate in a mutually suspicious manner. The reliability of such software systems has to be safeguarded against intentional or unintentional intrusion.

If an intruder can masquerade ingeniously as a legal user, follow normal procedures and perform normal operations, there is very little that we can do to detect it in real-time. An intrusion is usually detected by abnormal phenomena, such as a user accessing a part of the memory not assigned to him, a user's attempt to read a file with the wrong password, or an execution of a program without authorization. Since a residual software bug can also cause such derivations from the normal behavior, all of these can be viewed as software bugs, either in the user program or in the operating system. There is a close relationship between the reliability and integrity of the program. Security safeguards can therefore be considered as a form of software bug trapping mechanism in real-time. The integrity of a program is protected by security measures, which protect the program from accidental or intentional disclosure to unauthorized users and from unauthorized modification.

In real-time systems particularly those dealing with national defense and banking, it becomes particularly important for the security of the system to be sure that the program contains no critical software bugs and that the system will not compromise the sensitive information when there is a hardware failure. The software bugs in the program can lead to a breach of security and may be planted by an infiltrator. A large computer program must necessarily involve a considerable number of programmers. An intelligent infiltrator will therefore start at the stage when the program is most vulnerable, namely, when it is being written. Subtle software errors can be introduced to make the program inoperative when special conditions arise. Secret entry points and loopholes can also be created by the infiltrator for later usage. Not only are these bugs difficult to find but even when these errors are discovered, the ingenious infiltrator can always appear

as an ingenuous programmer to relieve the blame. (However, in any case, he should be fired!) The risk of the infiltrator is therefore minimal. The only effective countermeasure is to validate the system with automated tool so that it operates correctly as required for all the inputs at all time.

Active infiltration into the system during its operation can be achieved in different ways. [Pet 67]. A person may use legitimate access to ask unauthorized questions. He may find subtle entry points or "trap doors" which may exist by virtue of the combinatorial aspects of the many system control variables. He may also masquerade as a legitimate user by unlawfully obtaining the proper identification such as a password or by intercepting and cancelling the legitimate user's sign-off signals, followed by continued operation under his name. Another common method of infiltration is by examining the contents of core memory left behind by the previous user to look for useful information such as passwords, file names, etc. An intruder may also force his entry into a critical program and execute it. Moreover, a clever person may be able to put his process into supervisor mode and then virtually do anything that he likes. Protection of the integrity and privacy of programs must be provided against all these active threats.

4.2 Security analysis

Security is the process of detecting and preventing unauthorized modification, access and snooping of sensitive information. This implies the necessity of adequate safeguards built into the management and hardware/software aspects of the system. As in any large-scale system, the analysis effort is considerably reduced when it is possible to arrange the system in a hierarchical fashion. Then we can conveniently concentrate our effort at one level at a time, starting from the lowest level. At each level, we can neutralize the threats, thus providing a secured "hardcore" to work on the next higher level. A secure system can be arranged into a hierarchy of 6 levels, according to the vulnerability to threats. (See Figure 24) The lowest or the most critical area which must be secured is the program specification. Inconsistent or poorly defined specifications would provide the ready means to introduce programming bugs, such as trap doors and loopholes. The specification must be very rigid, providing no reason for ambiguity. The next level of security involves the operating system, since it could modify any program under execution. The operating system must be checked to see that it interacts with the job program properly and would not modify it in an unknown way. The third level of security is concerned with the program implementation process of the specification. The program design must be such that the behavior of the execution sequence must be clearly visible from an examination of the static code. In other words, a static analysis of the code should reveal all actions the program takes very clearly. This implies that the program must be structured and modular.

The next level of security involves the machine diagnostic aspects and the computer operator, since bad maintenance procedures could reveal the contents of the

memory or an unscrupulous computer operator may modify the contents of the memory without proper authorization. Hardware faults are also threats to the system security. The next level of security will consider the active threats, which are execution time interferences. Here the safeguards would involve authentication of entries of users into the system, real time sequence checking (relay runner) and real time validation of code before execution. The last level of security would be threats which involve stealing the information physically from storage devices or by monitoring the radiation emanated by the electrical devices.

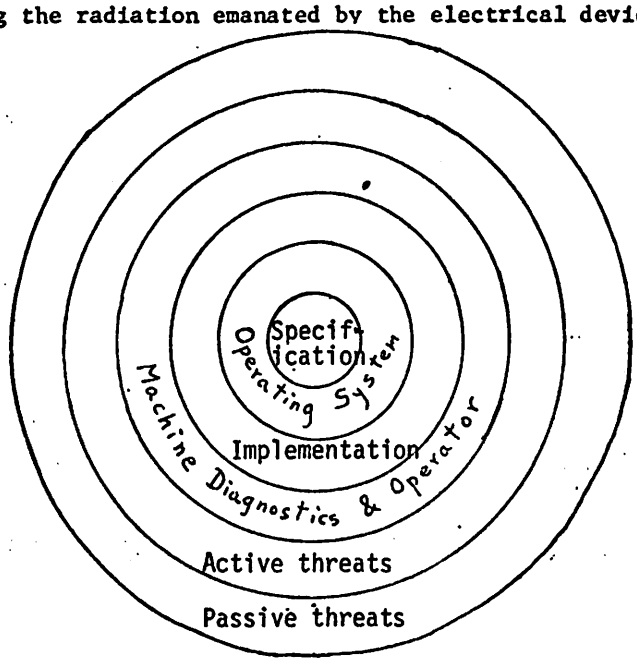


Figure 24 Levels of Security Analysis

4.3 Security safeguards

4.3.1 Safeguards against software and hardware errors.

The first three levels of security can be protected by validating the program against software errors. Techniques for constructing reliable programs can be used. Precise and unambiguous specifications should be formulated by committees. The programs should be written as structured programs so that high level personnels can understand and verify them. Well-structured programs make loopholes and planted bugs easily visible. The interface between the program and the operating system should be validated. After that, automated tools can be used to analyse and test the system for security.

The maintenance engineer and console operator have direct access to the machine. Reliable personnel should be employed in these important positions (from a security point of view). The integrity of the hardware system can be checked by performing periodic diagnosis. Techniques for protecting the system from hardware malfunctions are well-known [Ram 74a], though rarely implemented, due to the high cost. Thus the fourth level of security can be satisfactorily protected.

4.3.2 Safeguards against active threats

4.3.2.1 Introduction

Hardware and software safeguards can be used in this level of security. Hardware safeguards can be used to make sure that privileged instructions can only be executed in the supervisor state. Privileged instructions prevent the user from interfering with the operating system or another user's file or program. Various types of memory protection schemes are useful to protect the integrity of user programs and data, such as relocation and bounds registers (in CDC 6000 series), lock and key scheme (in IBM 360 series), paging (in XDS 940), segmentation (in Honeywell 645), etc. The ring structure in Multices can also provide adequate protection of user privacy. Other types of hardware countermeasures include built-in identification codes for computers (such as the IBM 370) or terminals, microcode, etc.

Software safeguards can be provided by access management and threat monitoring. Access management deals with the methods of accessing information and service in the computer and determining who is going to get what. Different ways of authentication and identification of users can be used, such as passwords, authentication algorithms, and proferring of physical items like badges, fingerprints, etc. Threat monitoring keeps a record of all access or attempts to access sensitive data and service. A log of all sensitive operations can be kept, recording who got access to what. A review of this record periodically can detect unauthorized attempts to use sensitive information or service. All successful breaches of security are also recorded, allowing the system manager to close these trapdoors. Besides, it is an effective tool against intrusions based on a trial and error strategy. Threat monitoring should always be active as long as the computer is operating. Therefore, we must make sure that it will not be deactivated by a privileged instruction.

All these safeguards have been discussed extensively in current literature. [Hof 73]. However, all these techniques can only be used in the design of the hardware and software of the system. The user remains a helpless prey of loopholes in the security of the system. It is highly desirable for a user to have programming techniques which he can use to protect critical sections of his program. The "relay-runner" scheme in the next section is such a technique.

4.3.2.2 The "relay-runner" scheme

In order to prevent illegal execution, we can authenticate all entries into the system by means of passwords, etc. [Gar 70]. If the intruder enters the system masquerading as a legal user, the "relay-runner" scheme can be used effectively to neutralize his threats. The "relay-runner" scheme provides protection against illegal execution of the code by an infiltrator as well as prohibits illegal jumps and modifications that may be due to software errors or hardware malfunctions in the system. This is achieved by detecting all illegal

changes in the execution sequence.

Consider the case of a simple assembly language program with no branchings and no loops. This piece of code can be partitioned into blocks separated by relay checkpoints. These checkpoints are conditional statements to test if the program flow carries the valid, up-to-date relay code. The user, upon legal entry into the system, enters the program and executes from the first executable statement. Here it stores the first of a series of relay codes (baton) in some address, say, RYC1. Then the normal program begins execution. When program execution reaches the first relay checkpoint, the instruction compares the content at RYC1 with a preset code number. If the codes agree, the content of RYC1 is changed to some other number, and a new relay code is stored in location RYC2. If the codes do not agree during the test at the first relay checkpoint, a trap routine is invoked and execution is discontinued. This process is carried on at suitable intervals throughout the program, with the "baton" carried along. This is analogous to a relay-race, where the next relay-runner will not continue unless he receives the "baton" from the previous runner. This prevents the programmer from modifying the execution sequence. If he jumps ahead by one step the relay-point is not yet activated and so the program will not continue its execution. If the programmer backtracks in his execution, the old "baton" value is already lost and the relay-point check will also discontinue its execution. The "relay-runner" protocol also prevents illegal entry into the program for unauthorized code execution since the intruder will not possess the correct "baton", which is generated in real-time. The use of "relay-runner" checks therefore reduces the legal entry points into one, which can be tightly protected. Depending on how closely the relay checkpoints are installed, a varying degree of security is obtained. Closely installed checkpoints give tighter security. Fig. 25a gives a graphical representation of the Relay Runner concept.

To visualize how the program of Figure 25a is protected from illegal execution, assume the legal user has just executed the instruction p3 in Figure 25b and is in waiting state because of some resource request. An infiltrator enters this piece of codes and starts executing the instruction at P1. He will be successful until program control reaches RP2, at which point the content of RYC2 is compared with 15. Since the legal user has executed the instruction at CP21, RYC2 now contains 200. Therefore, the test fails and the trap routine will be invoked.

If branching exists in the original program, care must be taken in the placement of relay codes and checkpoints such that every possible path of program flow is covered, and that the setting and resetting of relay codes do not interfere. In general, the programmer should organize his program so that all branches should emerge from one common exit. In this case, the relay checkpoint can be placed right at the exit point. Figure 26 shows one way of achieving this.

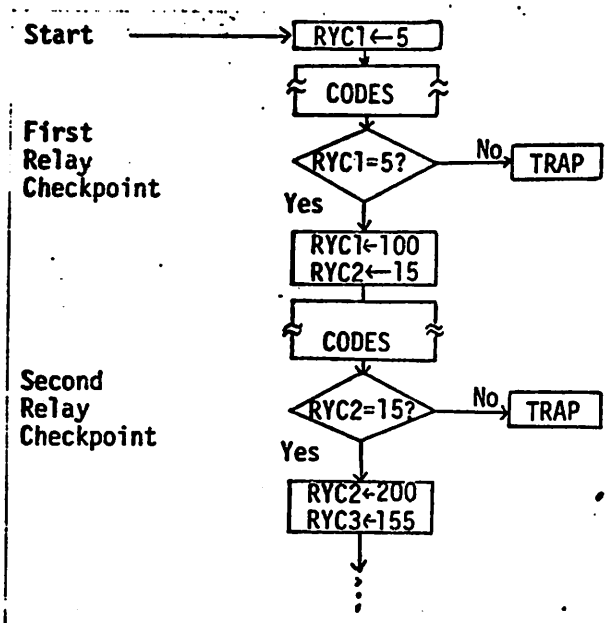


Figure 25a A Partial Flowchart for a program with the Relay Runner scheme implemented.

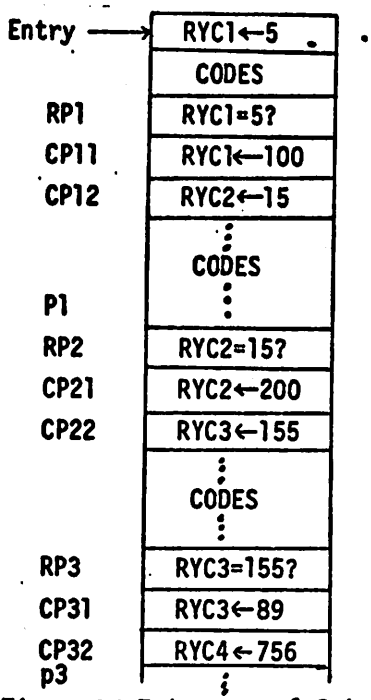


Figure 25b Existence of Critical Points within the Protected Program.

If loops exist in the program, relay checkpoints may be placed right before and after the loops if they are small. But if much input/output is done within the loop, the programmer may wish to put checkpoints inside the loop as well. It is advisable, in most cases, to organize the loops and branches in a single-entry/single-exit fashion. A tagging strategy may also be used to indicate the specific paths traversed so that the proper relay codes are addressed at the common checkpoints.

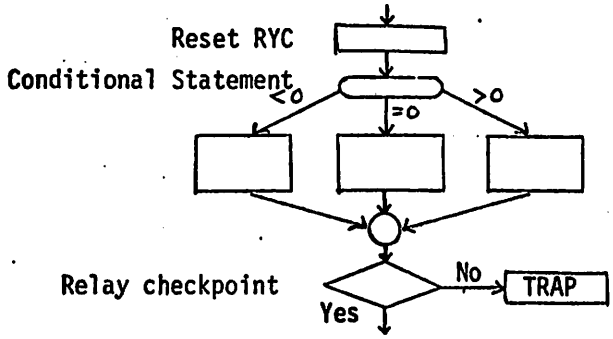


Figure 26 Configuration to handle Branching inside Program.

Some programs using this technique were run on a CDC 6400 system with COMPASS support. The objective is to obtain some overhead figures for various block sizes and sub-block sizes. The structure of the actual program used bears a close resemblance to the one described in Figure 27, [Ng 73].

With a mean program length of 80000 COMPASS instructions, run-time overhead

figures are obtained for programs with block sizes up to 5000. The results are summed up by the graphs of Figure 28.

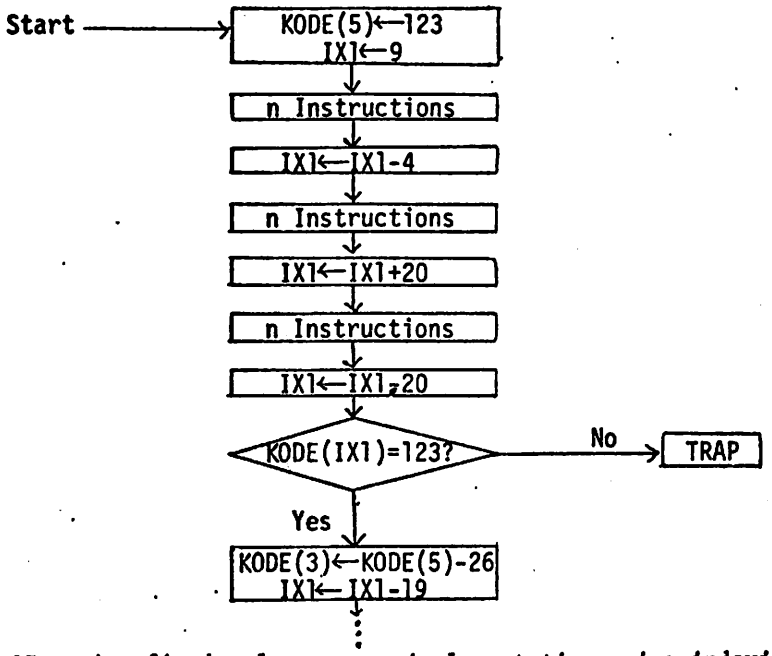


Figure 27 A refined relay-runner implementation using indexing

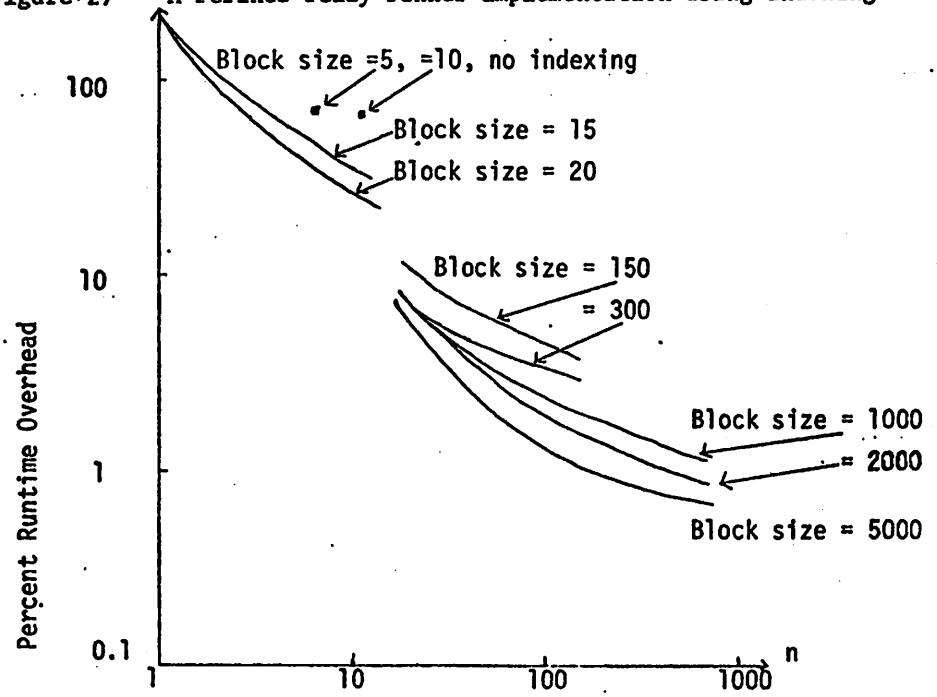


Figure 28 Run-time overheads for some simulation runs

For sizes of n ranging from 3 to 1000, the run-time overhead varies from 55% to 0.89%. Since these are average figures, the user has the freedom to protect the critical parts of his codes with a small block size.

While overhead figures of 25% or 50% may sound high, typical programs do not

have large critical areas, so that large block sizes can be tolerated. Note that overheads between 1% and 10% can be achieved by using block sizes of 2000 and smaller. Therefore, the Relay Runner scheme is a valuable tool for controlling illegal executions due to unwarranted intrustions or unpredicted errors in real time programs.

The above discussions and schemes are equally applicable to protecting pure procedures and shared codes provided that users have their own data storage.

4.3.2.3 Integrity checks

Even with all these safeguards implemented, there is still no guarantee that the system is absolutely secure. A real subversion is usually caused by a penetration that has not been detected. The execution of a program that has been illegally modified can be disastrous. It is desirable therefore to be able to check the integrity of the system periodically or just before critical programs are executed. In real-time systems, the detection of damage is very important. In many cases, we would rather stop a process than allow a wrong operation to be performed since many of the results are inevocable.

To ensure that the code to be executed is not illegally modified, one could validate the code just before execution. A simple scheme for this will be to develop a check sum of the contents of the code to be executed and compare it against the correct check sum that is stored at a secure place. If the computed check sum agrees, the sub-program is allowed to be executed. It is important to recompute the check sum for validation of the sub-program if some authorized modifications are made during execution.

The integrity of the hardware of the system can be checked by periodic diagnosis. The design of systems which can tolerate hardware faults has been investigated intensively. [Ram 74a]. However, in systems employing stored program control, software problems can also destroy the integrity of the system, and then the user programs. Software errors can come from residual design errors, incorrect maintenance, or intrusions from external sources. In order to protect the integrity of the system for continuous operation, different kinds of software defenses can be used. These defenses are highly specialized for the environment in which they are used. Some techniques used by the ESS system of the Bell System are presented here. [Con 72]. They include circuits that monitor program operation, in-line program checks, and audits. Circuits can be designed to monitor the proper sequence of operations of the program and trigger recovery action if an error is detected. An example is the use of an external "watchdog" timer which have to be reset periodically by program action. A failure to reset the timer will be interpreted as a software error. In-line program checks can be provided by the program itself to check for "impossible"conditions or abnormal states of the program. These will also include defensive programming techniques discussed in the last section.

Audits are a collection of independent check programs which detect and correct errors in memory content. These techniques are especially necessary for systems employing stored program control, such as the ESS. Audits can use the redundancy in the software structure to perform logical checks to locate errors and inconsistencies. Sometimes the redundancy is inherent in the data structure, such as a linked list. In other cases, it may be necessary to expand the data structure for audit programs. In these systems, audits have to be run periodically and frequently to ensure the integrity of the memory content. They are also run during system reinitialization during recovery from an error. Audit programs can be used to check that a linked list structure is intact or to verify the integrity of data constants and parameters stored in a writable memory by consistency checks with a less volatile backup record. Integrity checks can be performed by comparing some redundant data stored in different parts of the system to detect state discrepancies of facilities involved. Timing checks can also be used to ensure that no facility is being used beyond its maximum allowed time-limit. This prevents the loss of facilities due to an error. Audits are integrated into the system software. They can be run in an interleaved fashion with normal system operations. After detection of errors by audit programs, a recovery procedure follows. Backup information can be used to return the software to a "safe" state. If the error is serious, a system reinitialization may be performed.

4.3.3 Safeguards against passive threats

Passive threats consist of different means to obtain information illegally, including wiretapping, unauthorized access to data in removable files, etc. The effective countermeasure against these threats is encryption. Encryption is a form of privacy transformations, which takes data in its natural form and transforms it by scrambling so that it is hopefully unrecognizable by unauthorized people. Therefore even if a person is able to obtain the information, it will be meaningless to him.

An encryption system can be visualized in Figure 29.

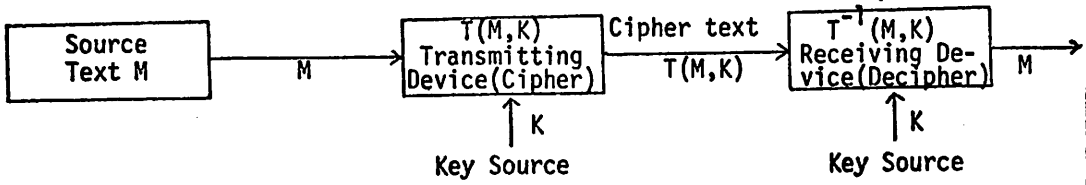


Figure 29 An encryption scheme

The operation done on the source text M depends on an input parameter K, called the "key". This key is essential for us to perform the decipher process. Only authorized people possess this key. Some simple encryption procedure includes substitution of one character strings for another, algebraic addition of key characters to message characters to form encoded messages, or rearrangement of the

ordering of characters in a word. For different encryption schemes , the reader is referred to the famous book "The Codebreakers". [Kah 67].

4.3.4 Conclusion

We can see that the reliable operation of a large program depends on the integrity of the program, which in turn hinges on the integrity of the system. A number of security measures to protect the integrity and privacy of the system are discussed here. These techniques can be applied to protect the system against simple hardware and software errors internal to the system as well as intrusions and threats from outside the system. However, a user must always bear in mind that these security measures do impose a cost on the user, in the form of longer execution time, larger memory storage, hardware redundancy, software redundancy and inconvenience. The extensiveness to which these techniques should be applied depends on the sensitivity of the information to be protected and the penalty-cost of a successful intrusion. A cost-effectiveness analysis of each of these techniques is urgently needed in the near future. Unfortunately, however, many of these techniques are tailored to certain specific environments. It is very difficult to generalize many of these techniques for analysis.

After the safeguards have been implemented, it is desirable for us to be able to evaluate the effectiveness of the protection that they provide to the system and different means to improve the protection economically. This can be achieved by measurement and modelling. Several models have been proposed, including Weissman's ADEPT-50 Model [Wei 69], Turn's Model [Tur 72], etc. Although these models are far from perfect, they represent some significant efforts towards the performance evaluation of security systems. Of course, all these safeguards would be wasted if we do not have a good administrative and physical security. The techniques required to achieve this goal are discussed in the book by Van Tassel [Van 72].

5. Conclusion

From the discussion in the previous sections, we have surveyed different problems of developing large reliable computer programs and different techniques to improve the reliability of a program. All of these techniques have their advantages and their weakness. Fortunately these techniques complement each other and a combination of these techniques can enable us to produce a piece of reasonably reliable medium-sized software. For really large computer programs, a lot of work still needs to be done, as indicated throughout the text.

We are proposing the following scheme as a reasonable approach to developing reliable large-scale software:

(1) Specifications of the system.

(2) Design of the structure, decomposition, and modularization of the system, with the specification of each module.

(3) Coding of the system in a suitable programming language.

(4) Debugging, integration and check out of the system.

(5) Software evaluation and partial validation with the help of automated tools.

(6) Software fail-safe fail-secure instrumentation.

(7) Validation of protection and security measures.

The specification of the program has to be concise and precise, providing the implementor all the informations that he needs in order to complete the program. A uniform specification language should be used for modules at different levels. The language should be sufficiently formal and yet descriptive, allowing a programmer to understand all the exterior properties of the module easily. Some formal languages for the specification of software modules have been proposed, such as by Parnas [Par 72]. However, the usefulness and effectiveness of specification using a formal language still need to be evaluated since not enough experience has been reported. The investigation of the relationships between the external properties and the internal attributes of a program is also needed.

The design of the program involves the decomposition of the program into smaller modules and the organization of these modules. Some design methodologies have already been discussed in section 3.1.2. Liskov has also proposed some guidelines for the design of reliable software systems [Lis 72b]. Abstraction is a very useful concept to simplify and order the compexity of the system. The abstraction specifies what is being done without the details of how it is done. The identification of abstractions, however, depends very much on the designer and the concept that he wants to support and clarify. In some systems, abstractions of resources are used to structure the system. In systems for supporting a data base, the characteristics of data structure may form good abstractions. In order to make the modules for different levels of abstractions to be logically independent, the combined activity of the functions in a level of abstraction should only support that abstraction and nothing else. [Lis 72b] The system should be designed for maintainability and adaptability. The levels of abstractions should be arranged in a heirarchy fashion (as discussed in section 3.1.2.) and data used by 2 different levels should be passed as explicit arguments only. The distribution of system resources in this hierarchy should also be determined before implementation starts. The design of the program is still an art although much discipline have already been introduced by the concept of structured programming. Since the development of the program depends very critically on the initial design, much research work is urgently needed for methodologies to evaluate the design and propose improvements before the actual implementation of the program takes place.

The implementation of the system should be carried out with discipline. The philosophies of structured programming should be enforced. The program should be written for readability and understandability. Wherever the restrictions of structured programming (as presented by Dijkstra and Mills) seriously affect the productivity of the programmer and the efficiency of the program module, these restrictions

should be relaxed at a local level, i.e., the program module should behave like a structured program when used externally although it may be "unstructured" to a certain extent internally. Attributes for a structured programming language have been discussed in section 3.1.1.3. Some equally important areas of research may include "structure documentation" and "structured" techniques for an informal proof of correctness of each module.

The debugging of the program should be carried out in a bottom-up fashion, starting from the individual modules. The compiler can locate syntax errors as well as structural flows. A compiler with powerful diagnostics should be used at this stage rather than one with high efficiency. An optimizing compiler may be used to generate efficient code after the debugging stage. (This optimizing compiler, of course, has to be very reliable.) The development of verifying compiler for potential structured programming languages are also very valuable, since it will also help us to design such languages. Assertion languages will also enable us to establish our confidence on the program by an informal proof.

When the modules are integrated together, automated evaluation and partial validation systems can be used effectively. Since the modules are written to be logically independent, the number of relevant test cases for modules in 2 levels of abstraction is the sum of the relevant test cases for each level, not the product. The testing of combinations of modules requires only the validation of the interface (input and output parameters) of the modules since this is the only interaction between modules. There are no complicated interaction in control or implicitly shared data because the program is written with structured programming. Therefore, we can see how design can simplify the testing and check-out of the program. The automated evaluation and partial validation systems also will build up a maintenance data base so that modification of the program is very easy. Automated generation of test inputs is still far from satisfactory at present, although computerized assistance is very helpful for the programmer's synthesis of test data. When an error is discovered and corrected, the AEVS can also help us to avoid introducing undesirable side-effects to other parts of the program. Different monitors can be introduced into the system to collect program behavior statistics and provide run-time analysis of the program. This piece of self-metric software may also enable us to quantify some reliability measures of the system. The development of a useful reliability model for software development is urgently needed.

After all this careful development and extensive validation, the program should be quite reliable. However, for some real-time systems, such low error-rate still cannot be tolerated. Software fail-soft instrumentation can then be applied to detect and contain software errors in real-time using software defenses. Hopefully, undesirable actions due to software errors can be avoided and damages minimized when such errors are detected as early as possible. Fail-secure instrumentation are designed to protect the integrity and security of the information in the system against active intrusions or system software and hardware errors. These instrumentations depend

very much on the operational environment of the system. Therefore, these protection and security measures have to be validated in the appropriate environment.

Appendix A. Techniques for the manipulation of the graph model

For any model, the essential requirement is that the manipulation techniques must be amenable to automation. Various techniques for manipulating the graph model are available. The most representative approach is based on the use of a connectivity matrix. The graph is represented by a connectivity matrix C in which one row and one column correspond to each node and $C_{ij} = 1$ if and only if there is a directed arc from node i to node j in the graph. Fig. A.1 is the connectivity matrix of the program graph of Fig. 6.

	1	2	3	4	5	6	7	8	9
1	0	1	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0
4	0	1	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	0	1	1	0
7	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	1
9	0	0	0	0	0	0	0	0	0

Fig. A.1 The connectivity matrix of the graph of Fig. 6

On the basis of this representation, various manipulations can be performed in a convenient way. The suitability of this machine representation mainly comes from its structural resemblance to the storage structure of most computers. Some of the basic techniques are mentioned here. A node j is said to be reachable from node i if there is at least one directed path from node i to node j. All nodes reachable from each node can be easily found. A reachability matrix R is a matrix in which one row and one column correspond to each node and $R_{ij} = 1$ if and only if node j is reachable from node i. It was shown in [Pro 59][1] that $R = \lim_{N\to\infty} (C+I)^N$ where I is an unit matrix. A more efficient algorithm was developed in [Ram 66]. A column j in R represents all nodes which can reach to j. By using R, all MSC subgraphs can be simply identified as follows. First a new matrix $M = R \cap R^T$ is obtained. The number of MSC subgraphs is given by the number of distinct nonzero row vectors of M. Moreover, if $M_{i.} \neq 0$, then the nodes of the MSC subgraph correspond to the nonzero columns of $M_{i.}$. In Fig. A.2, $M_{2.} \neq 0$ and the nonzero columns of $M_{2.}$ are (2,3,5). So, (2,3,5) are all nodes in that MSC subgraph.

[1] [Pro 59] Prosser, R., "Applications of Boolean Matrices to the Analysis of Flow Diagrams" Proc. Eastern Joint Comp. Conf. 1959.

$$R = \begin{matrix} 0&1&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1&1 \\ 0&1&1&1&1&1&1&1&1 \\ 0&0&0&0&0&1&1&1&1 \\ 0&0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0 \end{matrix} \qquad M = R \cap R^{T} = \begin{matrix} 0&0&0&0&0&0&0&0&0 \\ 0&1&1&1&0&0&0&0&0 \\ 0&1&1&1&0&0&0&0&0 \\ 0&1&1&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \end{matrix}$$

Fig. A.2 R and M matrices of the program graph of Fig. 6

Other manipulation techniques such as opening the loop, etc. are available and the details are referred to in [Ram 66, 67]. All these tools are the background for the implementation of automated validation discussed in section 2.2.

REFERENCES

[Ale 69] Alexander, T. "Computers Can't Solve Everything," Fortune, May 1969.

[Ash 71] Ashcropt, E and Manna, Z. "The Translation of 'go to' Programs to 'while' Programs," Stanford AI Memo AIM-138, STAN-CS-71-188, January 1971.

[Bak 72a] Baker, F. T., "Chief Programmer Team Management of Production Programming," IBM Syst. J., 1972, pp. 56 - 73.

[Bak 72b] Baker, F. T., "System Quality Through Structured Programming," Fall Joint Computer Conference, 1972, pp. 339 - 343.

[Bak 73] Baker, F. T. and Mills, H. D., "Chief Programmer Teams," Datamation, December 1973, pp. 58 - 61.

[Bas 72] Baskin, H. B., Borgerson, B. R. and Roberts, R., "PRIME - A Modular Architecture for Terminal-Oriented Systems," Spring Joint Comp. Conf., 1972, pp. 431 - 437.

[Ben 73] Benson, J. B., "Structured Programming Techniques," Record of the 1973 IEEE Symposium on Computer Software Reliability, May 1973, pp. 143 - 147.

[Boe 71] Boehm, B. W., "Some Information Processing Implications of Air Force Space Missions: 1970-1980," Astronautics and Aeronautics, January 1971.

[Boe 73] Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," DATAMATION, May 1973.

[Boh 66] Bohm, C. and Jacopini, G., "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," Comm of ACM, May 1966, pp. 366-371.

[Bro 72a] Brown, J. R. and Hoffman, R. H., "Evaluating the Effectiveness of Software Verification - Practical Experience with an Automated Tool," AFIPS FJCC, 1972.

[Bro 72b] Brown, J. R. and Hoffman, R. H., "Automating Software Development: A Survey of Techniques and Automated Tools," TRW Tech. Rep., May 1972.

[Cha 73] Chang, H. Y., "Topics in Designing Maintainable Real-Time Systems," Proceeding of the 2nd Texas Conference on Computing Systems, Nov. 1973.

[Che 74] Cheung, R. C., Kim, K. H., Ramamoorthy, C. V., and Reddi, S. S., "Automated Generation of Self-Metric Software," 7th Hawaii International Conference on Systems Sciences, January 1974.

[Cla 73] Clark, R. L., "A Linguistic Contribution to GOTO-less Programming," DATAMATION, December 1973, pp. 62 - 63.

[Con 72] Connet, J. R., Pasternak, E. J., and Wagner, B. D., "Software Defenses in Real-Time Control Systems," Digest of Papers of the 1972 International Symposium on Fault-Tolerant Computing, June 1972, pp. 94-99.

[Dij 65] Dijkstra, E. W., "Programming Considered as a Human Activity," Information Processing 65, W. A. Kalenick, (ed.). Proc. of IFIP Congress 65, VI, Spartan Books, Inc., Washington, D.C., 1965.

[Dij 68a] Dijkstra, E. W., "GO TO Statement Considered Harmful," Comm. of ACM, March 1968, pp. 147 - 148.

[Dij 68b] Dijkstra, E. W., "The Structure of the "THE" - Multiprogramming System," Comm. ACM, 1968, pp. 341 - 346.

[Dij 69a] Dijkstra, E. W., "Structured Programming," Software Engineering Techniques Report on a Conference sponsored by the NATO Science Committee Rome, Italy, J.N. Buxton and B. Randell (eds), 1969, pp. 84 - 88.

[Dij 69b] Dijkstra, E. W. Notes on Structured Programming, Teshnische Hogeschool, Eindhoven, Netherlands, August, 1969.

[Dij 73] Donaldson, J. R., "Structured Programming," DATAMATION, December 1973, pp. 52 - 54.

[Els 71] Elspas, B., Green, M. W., and Levitt, K. N., "Software Reliability," Computer, January 1971, pp. 21 - 27.

[Els 72] Elspas, Levitt, Waldinger, and Waksman, "An Assessment of Techniques for Proving Program Correctness," Computing Surveys, June 1972, pp. 97 - 147.

[Elm 71] Elmendorf, W. R., "Disciplined Software Testing," Courant Symposium on Debugging Technique in Large Systems, 1971.

[Fab 73] Fabry, R. S., "Dynamic Verification of Operating System Decisions," Comm. of ACM, November 1973, pp. 659 - 668.

[Flo 67] Floyd, R. W., "Assigning Meanings to programs," Mathematical Aspects of Computer Science, Vol. 19, 1967, pp. 19 - 32.

[Gar 70] Garrison, W. A. and Ramamoorthy, C. V., "Privacy and Security in Data Bank," Technical Memorandum No. 24, Information Systems Research Lab., University of Texas at Austin, 1970.

[Gol 63] Goldstine, H. H. and von Neumann, J., "Planning and Coding Problems for an Electronic Computer Instrument, Part 2, Vol. 1 - 3," John von Neumann collected works, Vol. 5, Pergamon Press, New York, 1963, pp. 80 - 235.

[Goo 68] Good, D. I. and London, R. L., "Interval Arithmetic for the Burroughs B5500: Four Algol Procedures and Proofs of Their Correctness," Computer Sciences Technical Report No. 26, University of Wisconsin, June 1968.

[Goo 70] Good, D. I., "Toward a Man-Machine System for Proving Program Correctness," Ph.D. Thesis, Dept. of Comp. Sci., Univ. of Wisconsin, Madison, Wisconsin, 1970.

[Har 65] Harary, T., Norman, K. Z. and Cartwright, D., "Structural Models: An Introduction to the Theory of Directed Graphs," John Wiley & Sons, 1965.

[Hof 73] Hoffman, L. J. (ed.), Security and Privacy in Computer Systems, John Wiley & Sons, 1973.

[Ing 71] Ingallo, D., "The Execution Time Profile as a Programming Tool," Courant Symposium on Compiler Optimization, 1971.

[Ito 73] Itoh, D. and Tzutani, T., "TABEBUG-I, A New Tool for Program Debugging," IEEE Symposium on Computer Software Reliability, 1973.

[Jel 73] Jelinski, Z. and Moranda, P. B., "Application of a Probability-Based Model to a Code Reading Experiment," Record of the 1973 IEEE Symposium on Computer Software Reliability, May 1973, pp. 78-81.

[Kah 67] Kahn, D., The Codebreakers, MacMillan Co., 1967.

[Kel 73] Kelley, R. A., "APLGOL, An Experimental Structured Programming Language," January 1973, IBM J. Res. Develop., pp. 69-73.

[Ker 73] Kernighan, B. W. and Plauger, P. J., "Programming Style for Programmers and Language Designers," Record of the 1973 IEEE Symposium on Computer Software Reliability, May 1973, pp. 148-154.

[Kin 69] King, J. C., "A Program Verifier," Ph.D. Thesis, Carnegie-Mellon Univ., Pittsburg, Pa., 1969.

[Kin 67] King, P. J. H., "Decision Tables," Comput. J., August 1967.

[Knu 70a] Knuth, D. E. and Floyd, R. W., "Notes on Avoiding 'go to' Statements," Computer Science Department, Technical Report No. CS 148, Stanford University, January 1970.

[Knu 70b] Knuth, D., "An Empirical Study of FORTRAN Programs," CS-186, Dept. of Computer Science, Stanford Univ., 1970.

[Kra 73] Krause, K. W., Smith, R. W. and Goodwin, M. A., "Optimal Software Test Planning Through Automated Network Analysis," IEEE Symposium on Computer Software Reliability, 1973.

[Lin 72] Linden, T. A., "A Summary of Progress Toward Proving Program Correctness," Fall Joint Computer Conference, 1972, pp. 201-211.

[Lis 71] Liskov, B. H. and Towster, E., "The Proof of Correctness Approach to Reliable Systems," The MITRE Corporation MTR 2073, Bedford, Massachusetts, 1971.

[Lis 72a] Liskov, B. H., "The Design of the Venus Operating System," Comm. ACM, 1972, pp. 144-149

[Lis 72b] Liskov, B. H., "A Design Methodology for Reliable Software Systems," Fall Joint Computer Conference, 1972, pp. 191-199.

[Lon 70] London, R. L., "Bibliography on Proving the Correctness of Computer Programs," Machine Intelligence, 1970, pp. 569-580.

[Mad 60] Madnick, S. and Alsop, J. W., II, "A Modular Approach to File System Design," AFIPS Conference Proceedings 34, 1969, pp. 1-13.

[Man 69a] Manna, Z., "Properties of Programs and the First-Order Predicate Calculus," Journal of ACM, April 1969, pp. 244-255.

[Man 69b] Manna, Z., "The Correctness of Programs," J. of Computer and System Sciences, May 1969, pp. 119-127.

[Man 71] Manna, Z. and Waldinger, R. J., "Towards Automatic Program Synthesis," Comm. ACM, March 1971, pp. 151-165.

[McC 62] McCarthy, J., "Towards a Mathematical Science of Computation," Proc. IFIP Cong., 1962, pp. 21-28.

[McC 63] McCarthy, J., "A Basis for a Mathematical Theory of Computation," Computer Programming and Formal Systems, N. Holland Publ. Co., Amsterdam, 1963, pp. 33-70.

[McC 67] McCarthy, J. and Painter, J. A., "Correctness of a Compiler for Arithmetic Expressions," Mathematical Aspects of Computer Science, Vol. 19, 1967, pp. 33-41.

[McC 73] McCracken, D. D., "Revolution in Programming - An Overview," DATAMATION, December 1973, pp. 50-52.

[McG 71] McGonagle, J. D., A Study of a Software Development Project, James P. Anderson and Co., September 21, 1971.

[Mee 73] Meeker, R. E. and Ramamoorthy, C. V., "A Study in Software Reliability and Evaluation," Tech. Memo No. 39, Electronics Research Center, The University of Texas at Austin, February 1973.

[Mil 71] Mills, H. D., "Top-Down Programming in Large Systems," Debugging Techniques in Large Systems, R. Rustin (ed), Prentice Hall, 1971, pp. 41-55.

[Mil 73] Mills, H. D., "On the Development of Large Reliable Programs," Record of the 1973 IEEE Symposium on Computer Software Reliability, May 1973, pp. 155-158.

[Mil 74] Miller, E. F., Paige, M. R., Benson, J. P., and Wisehart, W. R., "Structural Techniques of Program Validation," Proc. COMPCON, 1974.

[Nau 66] Naur, P., "Proof of Algorithms by General Snapshots," BIT, 1966, pp. 310-316.

[Ng 73] Ng, F., "Run-Time Protection Schemes for User Software," Master Thesis, University of California, Berkeley, 1973.

[Pai 73] Paige, M. R. and Balkovich, E. E., "On Testing Programs," IEEE Symposium on Computer Software Reliability, 1973.

[Par 71] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," Technical Report CMU-CS-71-101, Carnegie-Mellon University, 1971.

[Par 72] Parnas, D. L., "A Technique for Software Module Specification with Examples," Comm. of ACM, May 1972, pp. 330-336.

[Pet 67] Petersen, H. E. and Turn, R., "System Implications of Information Privacy," Spring Joint Comp. Conf., 1967, pp. 291-300.

[Ram 66] Ramamoorthy, C. V., "Analysis of Graphs by Connectivity Considerations," JACM, 1966.

[Ram 67] Ramamoorthy, C. V., "A Structural Theory of Machine Diagnosis," AFIPS SJCC, 1967.

[Ram 71a] Ramamoorthy, C. V. and Chang, L. C., "System Segmentation for the Parallel Diagnosis of Computer," IEEE TC, March 1971.

[Ram 71b] Ramamoorthy, C. V., "Computer Program Models," Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, April 1971.

[Ram 71c] Ramamoorthy, C. V., "Fault-Tolerant Computing: An Introduction and Overview," IEEE TC, November 1971.

[Ram 71d] Ramamoorthy, C. V. and Mayeda, W., "Computer Diagnosis Using the Blocking Gate Approach," IEEE TC, November 1971.

[Ram 73a] Ramamoorthy, C. V., "Error Control, Protection and Security of Real Time Computer Programs," Invited paper at the InternationalComputer Conference in Taiwan, August 1973.

[Ram 73b] Ramamoorthy, C. V., Meeker, R. E., and Turner, J., "Design and Construction of An Automated Software Evaluation System," IEEE Symposium on Computer Software Reliability, 1973.

[Ram 74a] Ramamoorthy, C. V. and Cheung, R. C., "Design of Fault-Tolerant Computing Systems," to be published in Applied Computation Theory (ed. by R. Yeh), Prentice Hall, 1974.

[Ram 74b] Ramamoorthy, C. V., Kim, K. H., and Chen, W. T., "The Blocking Gate Approach to Software Testing," in preparation.

[Ran 69] Randel, B., "Towards a Methodology of Computer Systems Design," Software Engineering, January 1969, pp. 204-208.

[Rus 71] Rustin, R. (ed), Debugging Techniques in Large Systems, Courant Symposium, 1971.

[Sac 70] Sackman, H., Man-Computer Problem Solving, Auerback Publishers, Inc., 1970.

[Sho 73] Shooman, M. L., "Operational Testing and Software Reliability Estimation During Program Development," Record of the 1973 IEEE Symposium on Computer Software Reliability, May 1973, pp. 51-57.

[Tur 72] Turn, R. and Shapiro, N., "Privacy and Security in Databank Systems: Measures of Effectiveness, Costs, and Protector - Intruder Interactions," RAND Corporation Memo P-4871, July 1972.

[Van 72] Van Tassel, D., Computer Security Management, Prentice Hall, 1972.

[Wei 69] Weissman, C., "Security Controls in the ADEPT-50 Time-Sharing System," Fall Joint Comp. Conf., 1969.

[Wei 71] Weinberg, G. M., The Psychology of Computer Programming, New York, Van Nostrand Reinhold, 1971.

[Wil 70] Williman, A. O. and C. O'Donnell, "Through the Central 'Multiprocessor' Avionics Enters the Computer Era," Astronautics and Aeronautics, July 1970.

[Wul 71] Wulf, W. A., Russell, D. B., and Habermann, A. N., "BLISS: A Language for Systems Programming," Comm. of ACM, December 1971, pp. 780-790.