Today's complex operating and computing systems make systems testing a difficult task. Major problems arise when one attempts to measure the performance of a system in a multiprogram environment and to evaluate the interfaces between computer elements, programs, and operator.

Historically, testing devices were first developed to monitor system activity and to produce test data. Separate computer systems were next used to permit on-line data reduction and generation of appropriate test conditions.

The purpose of this paper is to describe a hierarchical control program design which incorporates the major capabilities of the previous solutions without requiring a separate computer. This low-cost, flexible technique has been applied in the measurement of various performance characteristics, in generating simulated error conditions, and in simulating machine devices and features.

Hierarchical control programs for systems evaluation

by D. D. Keefe

The task of testing a complete operating and computing system has become increasingly difficult, primarily because of the growing complexity of today's systems. An operating system consists of a control program and a set of job programs; a computing system consists of at least one processing unit and various channels, control units, and input/output devices. Not only must each program and system device be individually tested to assure proper functioning, but the various interfaces between programs in the operating system and devices in the computing system must also be tested. Finally, all parts of the system must be pulled together for the complete systems test required to gauge the performance of the system and to check the functional relationship among operator, operating system, and computing system.

In this paper, the major problems encountered in a typical systems test are discussed, two established solutions are reviewed, and an experimental technique, called *hierarchical control*, is described. Applications of the technique are used to illustrate its advantages and disadvantages.

assumptions

The class of systems which the technique assumes is typified by the IBM SYSTEM/360.[1] Applicable features include:

- Several interrupt levels and a priority scheme for choosing among simultaneous interrupt conditions.
- A privileged mode of operation (called *supervisor state*) that can utilize the full instruction set of the computer and run in an uninterruptable state.

- A nonprivileged, interruptable mode of operation (called *problem state*) which can utilize an instruction subset that normally includes all arithmetic, logical, and internal data handling operations.
- A hierarchy of active programs consisting of a control program that operates in the supervisor state and one or more job programs that operate in the problem state. The job programs may have priority levels that further contribute to the hierarchy.
- Input/output error-detection operations performed by devices, control units, and channels.
- Input/output error-recovery operations issued by the control program with the aid of status information presented by the channels.

The control program is assumed to be an integral part of the computing system. That is, the control program is considered as necessary to the operation of the system as are the devices. The job programs use the control program and, through the control program, the computing system to accomplish application-oriented tasks.

testing problems

Let us consider some of the problems which the system tester must solve. In a one-job program environment, performance is principally determined by straightforward throughput timings, and all costs of operating the system are directly chargeable to the job program. Even in this environment, however, it can be difficult to derive performance figures for specific phases or routines. The difficulties are compounded when the control program manages two or more job programs, using a multiprogramming algorithm to allocate processing time to the individual programs. The job programs require differing amounts of supervisor time and other system resources. It thus becomes difficult to efficiently assess the effect of an individual program on system performance.[2] It is similarly difficult to establish a valid execution cost of operating any one program in the system.

A control program must be capable of handling any error condition, whether the error originates in a device or a program. As is well known, the errors destined to occur eventually, once the system begins production runs, are often difficult to produce in a test environment. Moreover, some means of assessing system activity is of great importance to the designer of an operating system in determining whether the system is functioning as anticipated, and in isolating problem areas if it is not. Inasmuch as a thorough test is vital to assure proper operation of a system once it is released, one desires a method that tests a system vigorously and at the least possible cost.

established solutions

Looking back through the short history of systems testing, we find that two main techniques have evolved. The first uses specially built equipment to monitor system activities and produce output on graphic displays or magnetic tape, while other devices are employed to create error conditions within the system.[3] The

special-purpose devices are characteristically one-directional, i.e., they either record data or provide input signals. This approach is accurate and reliable and has the advantage of not disturbing the operating environment of the system. A large amount of data can be collected in a short period of time, and most of the data reduction is performed later, either by visual inspection of results or by analysis on a computer. The disadvantages of this approach stem primarily from its high cost and lack of flexibility.

A second approach uses a separate computer to monitor operations in the system under test. In this case, most of the data can be evaluated as soon as collected. The ability to modify the monitoring program gives this approach a great deal of flexibility, and false status information can easily be sent to the monitored system to force simulated error conditions. However, unless the monitor computer is very much faster than the monitored computer, it is difficult to prevent the loss of some data. This can be avoided by halting the processor of the monitored system until the monitor is ready to accept more data—at an additional cost in elapsed testing time. Regular control program and job program timings can be kept in normal relationship by using separate clocks to monitor the processor and channel operations. The monitor program inhibits attempts to pass interrupts to the monitored system until the job program reaches the point where the interrupt would have normally occurred. This we call "modified-time-base" processing. With a programmed monitor, data gathered from the test system can serve as feedback to subsequent operations, thereby providing the effect of "bidirectional" operations. External devices are often useful in providing the monitor computer with additional inputs that may be used immediately within the system. The major disadvantage of this technique is the additional cost of using a separate computer.

hierarchical solution

Let us now consider an approach that provides the main capabilities of the previous two methods at a much lower cost. Essentially, we take greater advantage of the multiprogramming capability of the computer systems by transforming the previously described two-computer technique into one that requires no separate computer.

An example of a hierarchy of programs in a normal system is given in Figure 1. We have a control program operating in the supervisor state and three job programs operating in the problem state. Job Program A is at a higher priority level than Programs B and C, that is, B and C are given the system only when A is unable to use it. Since B and C are at the same level, they alternate in using the system whenever activity drops to their level. Now let us expand the hierarchy by introducing two higher-priority programs, as shown in Figure 2.

The monitor control program, or monitor, operates in the supervisor state and takes control at all interrupt conditions. Appropriate indications are passed to the test analysis program for further processing. The conditions to be monitored are specified

Figure 1 Hierarchy of programs in a typical normal system

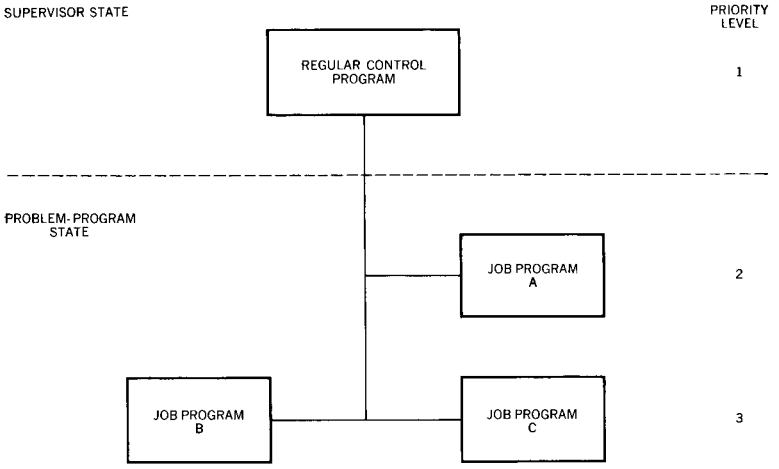
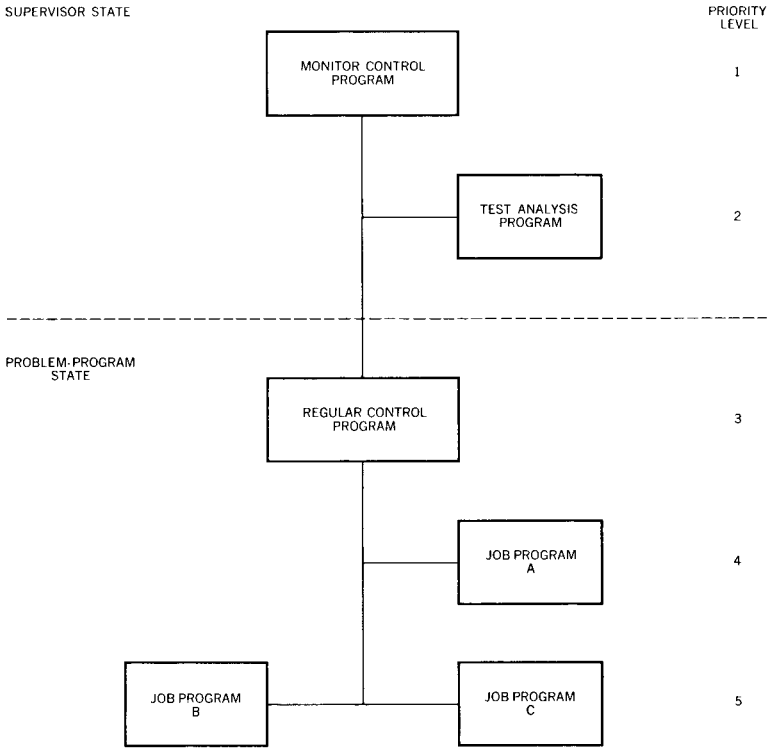


Figure 2 Hierarchy of programs in a test analysis system

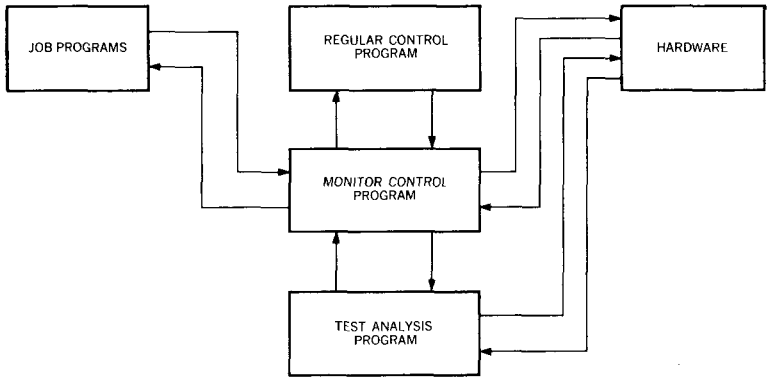


in the test analysis program at the time it is loaded. Although the test analysis program serves as a job program for the monitor, it is run in the supervisor state to give the test programmer maximum flexibility in altering the system to suit the requirements of his test. Moreover, the functions of the monitor and the test analysis

Figure 3 Control flow diagram

A. NORMAL OPERATION

B. MONITORED OPERATION



program are identical to those performed within the separate monitoring computer discussed in the second approach. The monitor must ensure that all information presented to the regular control program and its job programs appears exactly as it would in normal operation. The feedback capability of the two-computer system is also retained, adding greatly to the utility of the technique.

Note that the regular control program operates in the problem state. Consequently, when it issues a privileged instruction, a program-check interrupt occurs. The monitor takes the interrupt and is free to execute the instruction as issued, ignore the instruction, or modify the instruction before execution. This gives the monitor two key advantages:

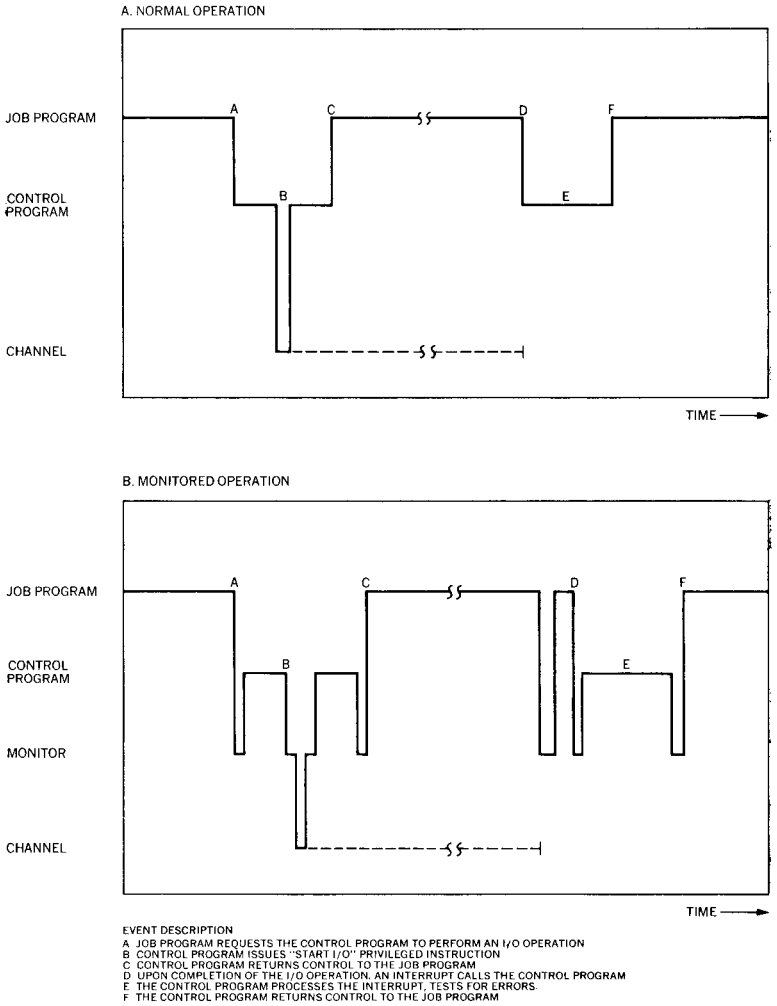
- The monitor can bar the regular control program from usurping control.
- The monitor can detect and handle all privileged instructions issued by the regular control program.

Also, linkages to the test analysis program may be formed for most types of privileged instructions and all interrupt conditions.

control flow

The flow of control between job program, control program, and equipment is depicted in Figure 3A for normal operation, and in Figure 3B for monitored operation. To better understand this, consider the time sequence of a typical input/output operation. In normal operation, a job program requests the control program to perform an input/output operation. This is shown in Figure 4A.

Figure 4 I/O request to completion



The control program initiates the operation and returns control to the job program. When the operation has been completed, the control program takes an input/output interrupt and checks for error-free operation. Control is then passed back to the job program. Figure 4B shows the same functions under monitor operation. The monitor does not pass the interrupt directly back to the control program. Rather, the interrupt is passed when the job program reaches the point of normal interruption. Thus, a normal timing relationship between the job program and the control program is maintained.

In designing a monitor program, two of the goals are high speed and low cost. In some respects, these are complementary, since increasing the speed of the monitor reduces the overhead associated with monitor execution. However, any costs associated

with achieving increased speed must be balanced against the savings produced.

Because the main storage required by the monitor and test analysis program makes up the major cost in a control program hierarchy, the basic monitor should be designed to be as small as possible, considering the desire for increased execution speed. The remaining storage requirement is largely a function of test analysis program complexity. Because registers used by the monitor have to be stored and loaded for every interrupt condition, execution speed tends to be inversely related to the number of index and general-purpose working registers used by the monitor.

monitor functions

The monitor must maintain an indication of the state (supervisor or problem state) that the regular control program has set for the system. For example, a privileged instruction causes an interrupt whether issued by the regular control program or by a job program. In the first case, the monitor ignores the interrupt and performs the instruction; in the second case, the monitor passes the interrupt to the regular control program. The monitor can normally use built-in storage protection features to prevent the regular control program from infringing upon the monitor storage area, thereby maintaining the proper "sphere of protection" for the control program and job programs.[4] The monitor must, of course, simulate all protection functions used by the control program, and it must ensure that the proper protection indications are forwarded to the control program.

Finally, linkages to the test analysis programs must be provided at relevant control points, which include all interrupt conditions as well as most privileged operations. These linkages enable a single control program to operate with all test analysis programs, much as in the case of a normal control program with a set of job programs.

system control

Given a monitor, system control can be established and maintained in one of several ways. The regular operating system can be relocated to an area of main storage other than that which it normally occupies. The facility for operating in this relocated area can be provided by a "relocation register,"[5] such as the Preferential Storage Base Address register of the IBM 9020,[6] or it can be achieved by using a relocatably assembled supervisor, providing that the computer and supervisor are designed for this type of operation.

A second possibility is to relocate the interrupt control area from the regular control program area to the monitor control program area. This is equivalent to the first method, except that only one operating system may be accommodated. The relocation can be achieved either by a special hardware feature or by a microprogramming modification.

Finally, under a third approach, the interrupt control area can be program-modified to link to the monitor program. This approach has the advantage of requiring minimum special features or modifications. However, the monitor may be vulnerable to loss of control if the control program inserts new values into its interrupt

locations. To prevent this, the monitor may either be tailored to a specific control program or use storage protection to guard the interrupt area from alteration by the control program.

In the course of full-scale, practical experimentation, the latter two techniques have been successfully applied at both the Operating System/360 (OS/360) and the levels of the Basic, Disk, and Tape Operating Systems (BOS, DOS, TOS/360).

To effectively measure performance, one needs high-resolution timers. The timers may be built into the processing unit, or they may be external devices. Several suitable timers are now commercially available with resolution times of less than one microsecond. The time spent in the regular control program and the job programs should be maintained, as this is the "process time" to be considered. Real-time data for input/output operations should be gathered, so that the appropriate interrupt points may be chosen. It may even be necessary to compensate for the storage cycles used by the channels while running in the monitor and test analysis programs, depending on the accuracy desired. Direct access devices may require synchronization loops to ensure that rotational delays are the same in the monitored system as in the original. As the degree of realism increases, so do the storage requirements and the actual execution time of the system being tested. Thus some tradeoff is normally called for, and a degree of timing approximation is accepted to hold the system cost at a realistic level.

experimental use

A brief look at an actual implementation of the technique should help to place the operating cost in proper perspective. (The system described is only operating in an experimental mode in a laboratory environment.) Our computing system is a modified IBM SYSTEM/360 Model 30. The modifications consist of special microprograms that perform the following functions:

- Relocate the interrupt control area from the regular control program area.
- Store a selected register into monitor program fixed location.
- Load Program Status Word (PSW) from monitor program fixed location.
- Switch from normal SYSTEM/360 operations to monitor-mode operation.

The latter three functions are program-initiated via the DIAGNOSE instruction. Control over an external timing unit, implemented with the aid of the READ DIRECT and WRITE DIRECT instructions, permits such functions as start timer, stop timer, reset, and read into storage.

The monitor program, written as assembly-language macroinstructions, requires a basic 500 bytes of storage. Linkage to each test-analysis routine requires four additional bytes, plus the storage required for the routine itself. To use the monitor, a test-analysis programmer issues the macroinstructions and parameters specifying the points at which he wants control. This is followed by the coding for his analysis routines.

The monitor is normally entered into the computer system via initial program load (IPL). Once loaded, the monitor issues the diagnose instruction to escape the normal SYSTEM/360 mode of operation. It then performs an IPL procedure to load the regular control program and commences operation. The execution times of the various monitor routines are summarized in Table 1.

A more meaningful description of the execution times is shown in Table 2, which presents actual run times for several jobs under DOS. The time for each job is shown for both normal execution and execution under the monitor. Total size of the monitor and test analysis program was 2,066 bytes, and these programs used the highest-address 4K block of storage. The test analysis program in this case counted and classified the various interrupts by type, and the increase in time compared to total test time is seen to be well under ten percent. This shows that nontrivial test data can be garnered at acceptable cost levels. On the other hand, to be sure, a highly elaborate test analysis routine could raise the execution time by several thousand percent.

applications

In speaking of applications of hierarchical control, one thinks of a given application in terms of a particular test analysis program running under the monitor control program. To formulate an application, appropriate control program linkages are selected and the corresponding routines are written. The test analysis programmer, like the job programmer in normal operations, is spared all of the considerations involved in actually controlling the system.

Table 1 Sample overhead times for SYSTEM/360 Model 30

Routine in monitor	*Execution time (microseconds)*
Supervisor call interrupt	464
I/O interrupt	431
External interrupt	478
Program interrupt	549
Load PSW instruction	742
I/O instruction (SIO, TIO, etc.)	666*

* Plus channel response time.

Table 2 Sample execution time for a job stream

Job Type	*Time Normal*		*Time Under Monitor*		*Percentage Difference*
	min	sec	min	sec	
COBOL (Compile and GO)	4	44	4	57	4.6
FORTRAN Compile	3	00	3	14	7.7
RPG Compile and GO	1	41	1	47	5.9

Applications that concern timing of various conditions have already been mentioned; these conditions were said to be those associated with interrupts or privileged instructions. It is also possible to time routines that are not interrupt-oriented by inserting "pseudosupervisor call" instructions at the start and end of the routine to be timed. Job programs can use supervisor call instructions to request action by the control program. Such an instruction causes a coded interrupt which specifies the called-for action. A pseudosupervisor call uses a code that is invalid to the normal control program, and the test analysis program employs the supervisor call linkage in the monitor to watch for these special codes. When one occurs, the time is recorded and control is passed back to the job program. The normal control program is not entered; it need not know that the interrupt occurred.

Other feasible applications of this technique are: traces of input/output operations, of all interrupts, and of privileged instructions; timing of program phases; and summaries of interrupt activities.

As mentioned earlier, the feedback capabilities of the technique can support other interesting applications, some of which are as useful in component testing as in systems testing. For example, consider the problem of writing a program to test a particular input/output device that has never been attached to a computer. To hold test time at a minimum, the test program should be ready to run as soon as the hardware is available. How then debug the test program? A test analysis program to simulate the operation of the missing devices is a ready-made solution. The simulator is entered whenever a test program tries to operate the device. Simulation can range from substituting a similar device to total generation of all status information and delays within the program, Applied at the systems test level, a whole communication network can be simulated, with messages arriving according to statistically distributed frequencies and responses analyzed for validity as they are issued.

Error recovery procedures have been completely tested by generating every possible device error condition in a test analysis program. In this approach, a device error bit is forced ON when an input/output interrupt indicates that the channel has finished an operation. To help pinpoint the specific error cause, normal control program operation interrogates the device in a "sense" operation. As soon as the sense data has been read in, the test analysis program modifies it to yield the desired error data. The reaction of the control program is then evaluated against the expected reaction. In this way, error recovery procedures can be validated in a few minutes, as contrasted with several hours or days in previous methods.

Another useful application has emerged in the form of program-testing techniques for getting more value out of each test run. In normal operation, when a programming error causes a program check interrupt, the control program cancels execution of the

program and goes on to the next job. It is possible, however, that the test analysis program be given a list of the starting addresses for each routine in the program to be tested. When a program check occurs, the test analysis program intercepts it, sends out appropriate diagnostic messages, and returns to the next routine in the list. This ensures that each routine in the job program is attempted. Without the technique, the probability of getting any useful data out of the test run after the first failure is zero. With the technique, it is usually close to one.

summary comment

A hierarchical control program structure proves to be a feasible and effective systems testing tool. The monitor program permits data to be gathered on almost any type of program activity and to be reduced in an on-line fashion. Applications of the technique are beneficial for component testing at both the computer and program level. In fact, the applicable range of systems testing applications appears to be limited primarily by the ingenuity of test designers.

CITED REFERENCES

1. G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM SYSTEM/360," *IBM Journal of Research and Development* **8,** No. 2, 87–97 (April 1964).
   G. A. Blaauw and F. P. Brooks, Jr., "The structure of SYSTEM/360, Part I, Outline of the logical structure," *IBM Systems Journal* **3,** No. 2, 119–135 (1964).
2. P. Calingaert, "System performance evaluation: survey and appraisal," *Communications of the ACM* **10,** No. 1, 12–18 (January 1967).
3. C. T. Apple, "The program monitor, a device for program performance measurement," *Proceedings of the ACM National Conference* **P-65,** 65–75 (August 1965).
4. J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computation," *Communications of the ACM* **9,** No. 3, 143–155 (March 1966).
5. J. B. Dennis, "Segmentation and the design of multiprogrammed computer systems," *Journal of the ACM* **12,** No. 4, 589–602 (October 1955).
6. G. R. Blakeney, L. F. Cudney, and C. R. Eickhorn, "An application-oriented multiprocessing system, Part II, Design characteristics of the 9020 system," *IBM Systems Journal* **6,** No. 2, 80–94 (1967).