W. C. Carter
H. C. Montgomery
R. J. Preiss
H. J. Reinheimer

Design of Serviceability Features for the IBM System/360

Abstract: This paper discusses the design of features that are intended to provide the IBM System/360 with a significant improvement in serviceability over that of previous systems. It was decided from the beginning to develop the System/360 as an integrated package of hardware, operational programs, and maintenance procedures.

The major problems to be solved in gaining this improvement and integration were (a) reducing the maximum duration of service calls; (b) reducing the median duration and mean duration of service calls; and (c) matching a single package of maintenance programs and procedures to a large variety of operational monitor programs and machine models.

These problems have been attacked by supplementing standard servicing facilities (both hardware and program) with (a) the ability to record automatically the complete, detailed, system environment at the instant of error discovery; (b) the ability to initialize the CPU to any arbitrarily specified state (either "legal" or "illegal"), to advance from this state by a specified number of machine cycles, and to compare the new state with a precomputed result state, much of this using circuits that are independent of those required for program sequencing; (c) a system of programs that can be integrated with the System/360 Design Automation to produce automatically the inputs, results, and location analyses that are required to exploit the capabilities described in (b); (d) a family of diagnostic monitor programs that attack directly the problem of matching maintenance procedures to machine models and operational monitor programs; and (e) a facility to retry failing CPU operations at the instruction level in the larger models, in addition to the usual retry at the program-segment level.

Introduction

During the early planning for the IBM System/360, the goal was to make a significant improvement in serviceability performance over that of existing systems. Improvements of the desired magnitude have necessitated a number of innovations and shifts of emphasis in system design. The purpose of this paper is to describe the more novel ways in which the response to the challenge of improving serviceability have been made in the System/360. The amount of hardware added, being model dependent, is not described. Because of the special attention given to the integration of features into the system models, and the multiple use of components wherever feasible, the amount of additional hardware was not excessive.

The process by which the general goal has been translated into specific objectives is outlined first. Next, the automatic fault-locating system developed to reduce the duration of service calls is presented. Then the combination of special hardware and programs that is provided to enable maintenance programs to be integrated into any machine/monitor program complex is explained. In the final section, the more significant serviceability advances in the System/360 are summarized.

The major serviceability objectives for the System/360

• *Basic definitions*

The system considered for serviceability consists of the System/360 hardware, systems programs, operational programs, operator, and environment. Most unscheduled maintenance calls are caused by *system failures*, which may be defined as the detected presence of an unwanted action or the detected absence of a wanted action. A *solid failure* is defined as one that always exhibits itself in the same manner to conditions that the user can control. An *intermittent failure* is defined as one over whose occurrence the user has little or no control.[1]

The operating period consists of time available for productive usage of the system, time for scheduled maintenance, and time for unscheduled maintenance. The time for the unscheduled maintenance is the product of the number of component failures and the mean time to repair these failures, plus the time necessary for the maintenance caused by operator mistakes, improperly acting programs, and the environment. Component failures are discussed by Davis, et al.[2] Although a number of System/360 facilities has been provided to facilitate the handling of program errors, the present discussion is limited largely to features that enable better isolation of failures in the hardware.

• *Developing the objectives*

The objectives to be set all assume that the initial effects of the learning curve[3] have passed, and the period of rapid change in maintenance effectiveness due to the introduction of new techniques and systems is over. This period frequently lasts approximately a year.

Current operating experience in the industry shows a distribution of unscheduled system down time of the form shown in Fig. 1. For most installations the most serious aspect of unscheduled maintenance is not so much the median or mean duration (although these are important) as the duration of the 90th and higher percentile failures. Grave inconveniences presently come from having a system down 4, 5, 6 or more hours at a time, because this disrupts the entire schedule of the installation. Hence, an immediate objective is to reduce drastically the maximum length of service calls.

In addition to the duration of the long service call, the mean and median durations of the service call are important. These three quantities are interrelated and depend upon the distribution of the duration of unscheduled maintenance calls as a function of the cumulative percentage of unscheduled maintenance calls. The distribution currently experienced is shown in Fig. 1. A working hypothesis is that the general shape of the distribution will remain relatively unchanged. If some mean duration

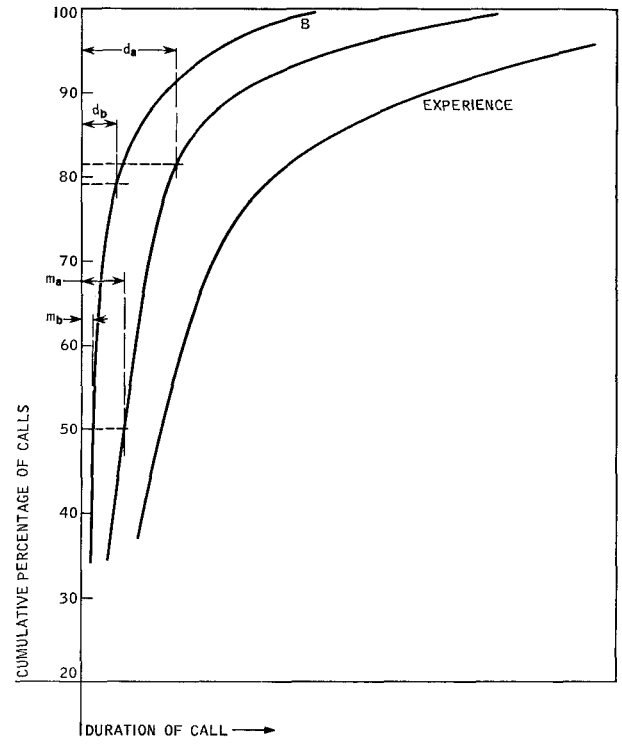

Figure 1 **Cumulative percentage of calls as a function of the duration of the call.** *The dimensions d_a and d_b represent mean durations, and the dimensions m_a and m_b represent median durations of calls for the anticipated distributions A and B.*

of unscheduled interruption is assumed, say d_a or d_b, then the curves labeled *A* and *B* in Fig. 1 indicate reasonable distributions of the duration of unscheduled interruptions. It can be shown from these curves that to reduce the mean for this empirical distribution it is necessary to reduce the median by the same percentage.

Based on the general goal mentioned in the Introduction and the considerations just discussed, the following priority of serviceability objectives for the System/360 was established:

a) Drastically reduce the maximum duration of unscheduled maintenance calls.
b) Reduce substantially the median duration of the unscheduled maintenance call.
c) As a corollary to priority b, reduce substantially the mean duration of the unscheduled maintenance call.

In a different vein, it was recognized that if program compatibility and component standardization were to contribute to serviceability to the extent they might, it would be necessary to ensure that the threads of similarity that pervade the several models remain visible to those who service the systems. The combinations of different

hardware implementations and operational-monitor programs must not be permitted to mask the fundamental sameness of different models. The determination that these potential benefits should not be lost led to the fourth serviceability objective:

d) Develop a single, comprehensive set of standard servicing programs and procedures that are effective for all of the important combinations of machine configurations and monitor programs.

Having set the objectives, the next task was to specify the tools with which the objectives would be accomplished and the environment in which the tools would be used. The first tool is additional circuitry that is designed to facilitate maintenance. The second tool consists of programs that use the new facilities and standard computer instructions to isolate malfunctioning components. These tools must be used in two environments. The environment most frequently encountered is that in which the system is failing repeatedly and symptoms of the failure are readily available. Diagnostic tools must be designed to use these symptoms directly. In the second environment, the system is failing only infrequently but its failure is interfering seriously with its desired use. In this case the problem is to obtain symptoms that can be analyzed. Once these symptoms are obtained, the infrequently failing computer may be used to process the data obtained concerning these symptoms.

Reducing the median and the mean duration of service calls

Reducing the duration of unscheduled service calls can be accomplished by the following general strategy:

a) For failures in units whose functions can be assumed by other units in the system, use a diagnostic program to gather symptom data, decouple the failing unit and continue operating the reduced system.

b) For failures in units whose functions cannot be performed by other units, improve servicing performance.

A method of implementing this strategy has been devised for the System/360. Failures of category a) will be discussed in later sections of the paper.

For failures of category b) it was decided to concentrate on decreasing the time required for diagnosis, since it is the largest single time component and the one that offered most promise for improvement. Two major obstacles that limited the reduction in diagnosis time in earlier systems are the following:

1) Only a minor fraction of the circuit elements was accessible to direct program control and interrogation. This has been called *the addressing-resolution problem.*

2) The interval between points at which the system status could be determined by a program was a complete instruction execution time. The system passed through many intermediate states, but these could be deduced only from what was observed at the end points. This has been called *the timing-resolution problem.*

Since earlier work[4] had suggested that an elegant solution to both of these problems could be achieved at modest cost, it was decided to attack them directly for some CPU's—and to do this from the inception of design in order to keep costs down.

For the smaller, read-only storage controlled models, a slightly different approach to accomplish the same end was provided. This will be discussed under the section entitled "Auto-Diagnosis."

- *Fault Locating Tests (FLT's)*

Diagnosis is accomplished by applying a series of stimuli and observing the corresponding responses. If the logic between stimulus and response can be uniquely controlled and identified, then a series of correct and incorrect responses can lead to the pinpointing of the failing circuit element. To provide better addressing resolution in the System/360, hardware is required to limit the participation of circuits to particular test stimuli. To perform the identification, a long and complicated program is required to calculate the necessary stimuli to test the logic, and to evaluate the responses. Since an ailing computer cannot be expected to perform this calculation, the data must be precalculated on a working system. The additional hardware needed by the ailing system is that required to apply the stimuli, to observe the responses, and to determine the next action to perform in order to arrive at a diagnostic conclusion.

How can the stimulus patterns be calculated? It is well known that the logical properties of primitive building blocks can be tested using a subset of all possible combinations of inputs.[5] The combinational logic network connecting any two storage elements in a computer consists of many levels of such elementary blocks connected as dictated by the computer logic. Any elementary block can be tested by an input pattern if an inversion of the correct state of its input or output lines will invert the observed logic network output.[6,7]

An IBM 7094 program complex (see Appendix) was developed to precalculate the sequence of stimuli necessary to identify a small number of replaceable circuit cards, one of which contains the failing component. The term *fault-locating pattern* is used to describe these stimuli. Each pattern represents a set of storage element states calculated to test a set of elementary blocks in the combinational logic network defined by an output and all of its inputs.

With the 7094 programs and the System/360 features, the following basic approach to rapid isolation of logic circuit failures is possible:

a) Improve the addressing resolution by manipulating directly the failing component at the combinational logic, sequential logic, or timing chart level.

b) Improve the timing resolution by providing the ability to record each sequential machine state at computer speeds.

c) Present information derived from this data in a way that requires a minimum of human analysis at the time of repair.

d) Provide immediate verification of repair.

To apply the stimuli, it was necessary only to add circuits to the System/360 to set the status of any storage element in the computer to any state and to provide the ability to determine the status of any storage element, i.e., to observe the response. This feature is also used in automatic environment recording, and is discussed later in this paper.

The FLT generator program complex analyzes the logic design data stored on the Design Automation Logic Master Tape[8] and calculates or analyzes a set of fault locating tests depending on the System/360 model for which they will be used. These tests may be stored on an input medium, such as magnetic tape or disk, for direct reading by a System/360 model in the fault location testing mode, or, for some models, diagnostic tests may reside in read-only storage.

This program complex (a) produces the FLT data to test directly most System/360 CPU components in the larger models; (b) orders these data so that the results of testing can be easily analyzed; (c) keeps these data up to date with engineering changes in the hardware logic; and (d) produces up-to-date documentation, which in itself is sufficient to reduce the duration of many maintenance calls.

Test application for externally-stored tests

The FLT tests are stored on an external storage medium, such as magnetic tape or disk pack. When the *Load FLT* button is pushed on the maintenance console, a simplified sequence of channel control circuits transfers a record of tests from the external medium into main storage. Each FLT is divided into the several fields shown in Table 1.

After an FLT record has been read into storage, the subsequent action of the CPU is controlled by the additional special circuitry* designed to test the computer using the FLT patterns. The storage elements of the CPU are first initialized according to the values of the stimulus pattern. A special counter is also set to the value given by the clock advance field. When this initialization (scan-in) has been completed, the machine clock is allowed to advance synchronously with the counter. As soon as the counter reaches zero, the clock is stopped and the machine

Table 1 **FLT format**

Field	*Function*
a	Identification
b	Stimulus pattern
c	Clock advance
d	Control bits
e	Precomputed expected result and its location identification
f	Identification of the alternate FLT's

identifies the storage element it has to compare with the precomputed result. This comparison is made, the result stored, and the test repeated a fixed number of times, *N*. If *N* consistent results are obtained, a decision is made depending upon this result as to which of the two alternate FLT's identified in field *f* of Table 1 is to be run next. If the results are inconsistent, an intermittent failure has been found, the computer is stopped, and the test and reason for the halt are displayed on the maintenance console. FLT documentation, discussed later, may then be consulted.

The primary sequence of FLT's will be followed by an error-free machine. (One of the FLT's identified in field *f* of a test in the primary sequence is the next such test.) The precalculated FLT patterns are prepared by the FLT generator so that an extremely wide coverage of the computer's elementary logic blocks is given.

If a failure is detected by a primary test, then the machine's failure of this test either does or does not provide sufficient diagnosis to identify the replaceable failing cards in the machine. If it does, the machine stops and the test number is displayed on the maintenance console. FLT documentation then lists the suspected failing circuit cards. The control bits—field *c* of Table 1 in the test—provide the information to determine whether to proceed or stop after a consistent failure.

If the primary test does not provide sufficient resolution, further FLT's are applied to improve the resolution. The second FLT identified in field *f* of Table 1 checks a subset of the elementary logic blocks that failed the primary test. If the subset fails the secondary test, then that subset contains the failure; if it passes, the failure is in the complement subset. Testing continues until a termination point is reached as shown in the typical test sequence given in Fig. 2. The primary and secondary sequences have been precomputed so that resolution to a few replaceable suspected cards has been achieved when a termination point is reached.*

* These circuits exist only in the larger models.

* For a simple engineering model, the number of replaceable suspected cards averaged approximately three.

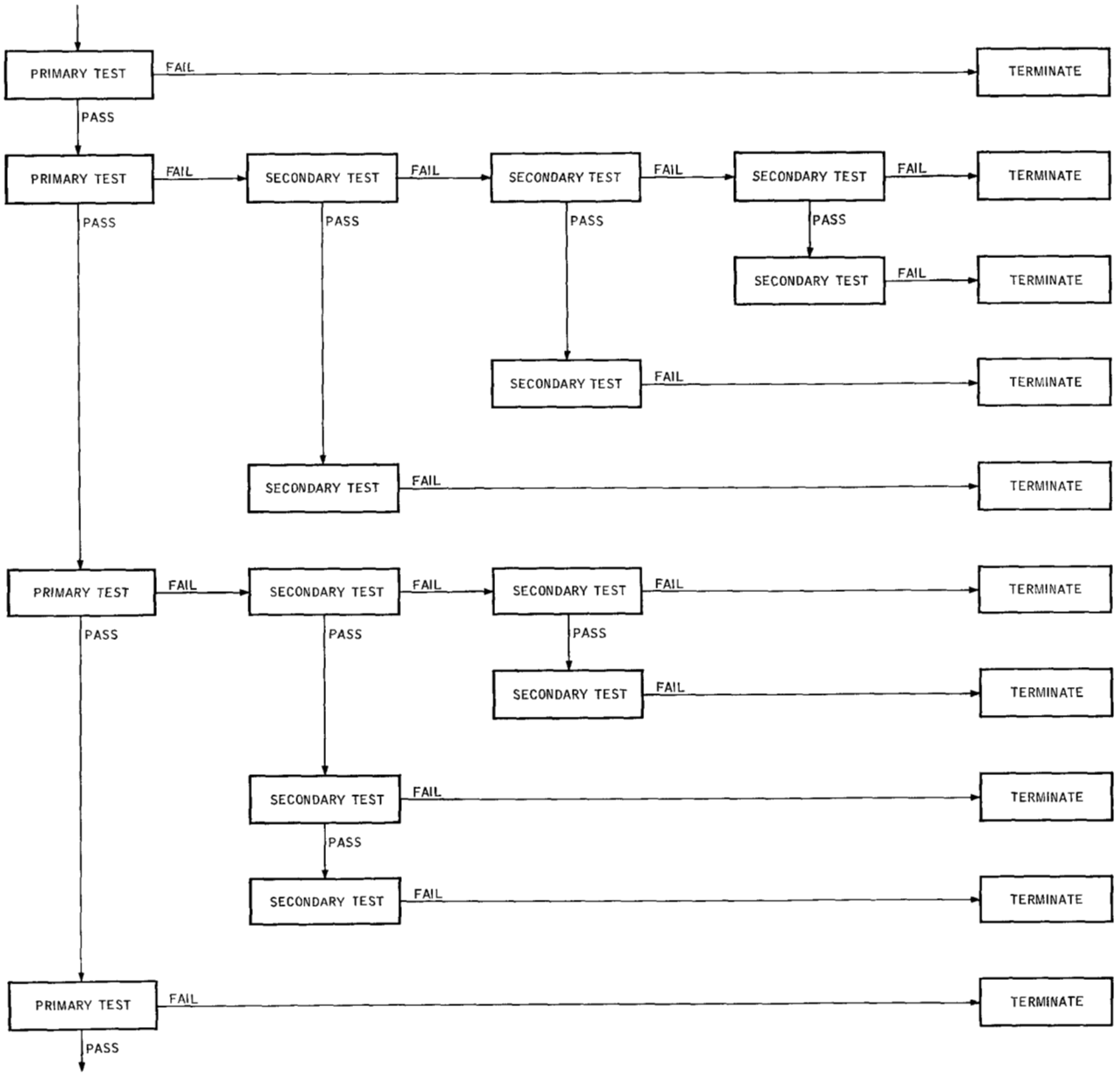

Figure 2 **Typical sequence of tests.** *Terminate blocks on the diagram indicate that sufficient resolution to allow repair has been achieved.*

To avoid erroneous conclusions caused by intermittent failures, the tests are designed so that the failure of a test is the condition that leads to a termination. If this does not happen naturally, the last test on a branch repeats the pattern of the first test in which a failure was indicated. If the last test does not indicate a failure, the machine is halted, and a special indication of an intermittent failure is given.

Test documentation

The FLT test documentation identifies all the machine logic that actively participates in producing the desired output for the test pattern input. A single failure in any one of the logic blocks described, or in the wiring between them, would cause an incorrect answer to be detected at the expected result location. (This does not mean that these are the only circuits used in the machine during the execution of this pattern. Instead, they are the only circuits that, with a single failure in them, can be detected at the output.) This listing also includes the test identification, all pertinent circuit pin numbers, together with the value that would be observed at these points with a dc probe or an oscilloscope, references to corresponding logic diagrams, and the particular cards that are suspects for specific termination points.

FLT features

Earlier work,[4] and experience with an experimental model, System/360 prototypes, and the IBM 1418, 1419, and 1428 have suggested that the following are the more significant FLT features:

a) The percentage of CPU hardware required to be operable for FLT's ranges between 3 and 10 percent (depending upon the model).

b) Fault location testing provides uniform procedures, documentation, and preplanned troubleshooting techniques.

c) Successive tests check out a controllably small part of the logic, thereby affording excellent location resolution.

d) The FLT's check small combinations of circuit actions, thereby making it possible to simulate logic conditions that would, under testing with conventional programs, be possible only with externally introduced unusual signals or with special computer instructions.

e) Fault location testing provides secondary documents that identify both the circuits that are tested and those that are not.

f) FLT's and their associated documents are automatically produced and updated.

g) FLT's, being generated by a production computer, require less debugging time on the prototype model, and less verification time to confirm that they do truly locate the failing circuits.

- *Auto-diagnosis*

For the System/360 Read-Only Storage (ROS) controlled computers, each instruction belongs to a compatible System/360 set, and is executed in one or more cycles. Each cycle determines a definite machine state by setting storage elements according to the contents of a word read from ROS, and by active logic as in conventionally controlled machines.

To the engineer who is single-cycling through an instruction, the computer controls seem conventional. In this method of design the direct use of microprogramming by the user is impossible. Both the advantages of the direct use of microinstructions, and the disadvantages of two levels of control are eliminated.

However, in machines with an ROS control element, it is easy to build in diagnostic facilities by using part of the ROS for diagnostic microprograms.* The computer circuits are diagnostically exercised on a cycle-by-cycle level, thereby maintaining both addressing and timing resolution. Error indications are given by the checking circuits. The local store and main storage tests exercise these units directly without reading instructions from them. The ROS also contains microinstructions that are used to invert parity bits so that the error indication circuits may be tested directly.

The serviceability attributes of ROS models of the System/360 are as follows:

a) Controls are easy to check; both the present ROS word and the possible next words are parity checked.

b) Documentation is complete on the single-cycle level, is presented in flow-chart form for ease of understanding, and is updated by computer programs.

c) ROS arrays in the smaller models can be easily checked by tests stored in the ROS.

d) ROS failures in the larger models can be diagnosed using FLT's.

e) Special features, as discussed later, can be included at modest cost and with little effect on existing control logic.

Reducing the duration of long service calls

Generally, the largest single factor contributing to long service calls is the time for diagnosis. Frequently the failures that offer the greatest resistence to rapid diagnosis are those called intermittents. The strategy for reducing the duration of long service calls in the System/360 is directed at three problem areas related to intermittent failures. First, the efficiency of intermittent failure diagnosis has been low because accurate and detailed symptom information has not been available. Second, in the past, the impact of intermittent failures in the CPU has been amplified by time-consuming recovery procedures. Third, many failures have been labeled and treated as intermittents simply because the tools available for inducing their symptoms have been inadequate.

- *Symptom data collection and analysis*

By definition, intermittent failures cannot be induced by the user in order to get symptom data. An alternative is to collect and preserve these data as they occur in real time. The System/360 provides a hardware/program system that does just this.

The requirements imposed by the problem on the hardware part of the system are (a) detection of errors immediately upon their occurrence; (b) recording of the detailed state of the system at the time of error; and (c) switching to a different instruction stream.

In the System/360, interruptions caused by checking-circuit alarms are automatically separated by the hardware from interruptions arising from program-caused conditions. Since the disposition of program failures is obviously at the discretion of the routines to which the interruption

* These tests are used extensively in the smaller models.

system leads, we shall limit the comments that follow to circuit failures.

Requirement (a), detection of errors immediately upon their occurrence, is satisfied by distributing checking circuits* throughout the system, including areas of control circuits, so that failures are discovered close to both their spatial and temporal points of occurrence.

Requirement (b), recording the detailed state of the system at the time of error, introduces an interesting problem that is reminiscent of the Uncertainty Principle: How can a machine record its status without changing what is being recorded? For CPU failures and failures in some I/O channels, the System/360 solution is the straightforward one of providing a special set of independent circuits to control this function. When a checking circuit detects an error in the CPU or in some parts of the I/O channels of larger models: (1) normal sequencing in the affected unit is halted immediately (CPU errors do not affect channel operations, and vice versa, for models in which the CPU and I/O channels operate autonomously); (2) under control of the special circuits the states of the control and data path storage elements of the affected unit are recorded as a pattern of bits in main storage; and (3) normal sequencing is then resumed.

Requirement (c), switching to a different instruction stream, is met by forcing a special program interruption, called Machine Interruption, upon resumption of normal sequencing.

Requirement (b) is satisfied somewhat differently in the I/O units. There are two basic differences here:

(1) A checking-circuit alarm causes a halting of the control unit for the device, but not necessarily of the device itself (because of mechanical motion). For instance, magnetic tape proceeds to the end of the current record, on reading. In some control units this halting is immediate, preserving the status at that point in time; in other control units only a partial status is preserved, while the unit proceeds to normal termination.

(2) The status information must be retrieved from the affected unit by a program.

For input/output units, requirement (c) is met by coded status information made available upon a normal I/O interruption, which occurs after termination of the command.[9]

Now the automatically recorded data give an accurate and detailed picture of the status of the affected unit itself (i.e., CPU, channel, or control unit), but they do not provide complete information about the total system environment. It is important, for this class of failures, to know more than just the local conditions of failure, because interference from sources external to the affected units is a common source of failure. The information to complete the environment record for each incident is gathered by a special program. Included in these supplementary data are: the I/O units that were active at the time of error, time of day, program identification, instruction operands, instruction addresses in storage and, where relevant, tape or disk-pack label. This information is combined with the automatically recorded data to form an entry in the environment record in secondary storage. The result is a complete, detailed, chronological maintenance history of the system, in machine-readable form.

To aid the customer engineer to use this information effectively, a program is provided to edit in a number of ways the entries in the environment record. Included are routines that print entries in a format convenient for reading, and retrieval routines that search the record and extract entries that satisfy the parameter keys specified in standard retrieval requests. For example, the customer engineer may ask for all entries related to a unit having a specified unit address, or for all entries triggered by a specified checking circuit.

• *Bypassing the effects of error—instruction retry*

The user of a system is largely indifferent to machine failures if they do not interfere with his operations. For instance, if all failures are discovered and repaired during periods when he is not using the system, the user is usually satisfied. Experience suggests, however, that failures are not always timed so fortuitously. When machine failures occur during normal operating periods, the user can either terminate his use of the system or he can (and often does for failures in tape units, for instance) try to continue operating the system by bypassing the effects of the failure.

Bypassing the effects of failures can be realized either by avoiding the system area in which errors have been observed or by repeating the processing that has been contaminated in the hope that the cause of the error no longer exists. The first alternative is clearly the choice for solid failures, and the second is preferable for errors whose symptoms are not persistent. This second alternative has been called *checkpointing*.

Avoiding an ailing system area can be done gracefully if the system contains sufficient redundancy and processing capacity so that the function of the suspected area can be assumed by a different area, or if the function performed by the suspected area may be temporarily eliminated. Both of these techniques have been applied with some success to I/O equipment. The widely varying I/O requirements of different programs have made redundancy a by-product of accommodating the I/O needs of the more demanding programs.

The avoidance technique can also be applied successfully, but less so, to the area of main storage. Avoiding suspected areas of main storage is practical only if there

* A variety of checking techniques are used in the System/360 but they are model dependent and so will not be discussed here.

are multiple storage subsections on line and the entire subsection in question can be deleted. Even here the impact on system operation is considerable since in most cases the affected program will have to be reintroduced into a different storage area and restarted from either the last checkpoint or the absolute beginning.

But it is in the CPU area that the avoidance technique can be used least successfully. Solid failures in the CPU have usually led to a halting of system operation in single-CPU systems and at least to a drastic revision of the operation of systems having more than one CPU. Reconfiguration of the system to continue processing with one less CPU requires specialized planning and programming.

In the past, the handling of intermittent CPU errors has not been very effective. Program segmentation coupled with checkpoint dumps and checkpoint rollback does indeed work for intermittent failures but with considerable waste of time. Also, it takes a long time to unmask a solid failure with this approach.

Avoidance is not required for intermittent errors in I/O operations. I/O control programs have for some years had an iteration resolution at the operation level rather than at the program-segment level. By proper programming, all of the information required to re-execute the I/O operation is still available at the end of the operation.

The hardware facilities included in the System/360 to solve other problems offer a new ability: the opportunity to re-execute a CPU instruction in the event a failure is detected. For retry to be effective for CPU operations, a first condition is that all the information required to re-execute the operation is still available after the failure is detected. A second condition is that the effects of the failure be nonpersistent.

For example, in an instruction that loads a register from storage, the first condition is satisfied since the operand is still available in main storage. However, if core storage failed, the effects of the error would be persistent and the retry would fail. The majority of instructions cannot be retried after their execution has been completed; their retry threshold* is usually after instruction-fetch sequencing, at some point in the execution phase.

Re-executing instructions imposes two requirements on the machine:

a) Execution must be halted immediately upon error discovery, so that if the retry threshold has not yet been reached no operands will be destroyed.

b) Sufficient information about the state of the machine when it was halted must be available to the program that decides whether or not re-execution is possible.

As was observed earlier, circuitry to satisfy these requirements has been provided for the System/360 environment-recording facility. Following a checking circuit alarm the computer state is frozen, and special circuits record the detailed status of the CPU as a pattern of bits in an area of main storage reserved for this purpose. The routine that stores the environment record also analyzes the stored status information, determines if retry is possible and counts the number of retry attempts. If the retry threshold is passed or the failure effects are persistent, the retry is unsuccessful. This routine takes a fraction of a second compared to the minutes required to perform checkpoint restart with its accompanying rerun.

The total effectiveness of this facility depends on the proportion of all failure incidents for which the retry threshold is not reached and for which the failure effects are not persistent. For a given machine the retry threshold varies from instruction to instruction, so that the relative success of retry is very sensitive to the instruction mix over which it is measured. Using the gross assumptions that failures in all cycles are equally probable, and that instruction usage is uniformly distributed, more than 50% of retrys will be successful for the System/360 models in which the facility works best. Variations in the instruction mix could lower this success percentage to perhaps 35% or raise it to perhaps 80%.

- *Reducing the number of intermittent failures*

If failures that would otherwise be treated as intermittent can be treated as solids, then the average time for diagnosing failures can be reduced. This is done by more thorough and better-controlled exercising of primitive building blocks. As discussed earlier, the procedure for generating fault locating tests is done by a complex of IBM 7094 programs. These programs produce lists of all logic blocks that are directly tested and those that are not. This permits a greatly increased coverage of directly tested components. This increased coverage allows many failures to be treated as solids that otherwise would be labelled intermittents.

In addition, if any FLT indicated failure only once, a list of suspected circuits would be immediately available. The cards containing these circuits can be replaced as a preventive maintenance precaution.

In summary, the number of intermittents is significantly reduced by increasing the control a person may have in the face of a symptom indicating suspicious behavior, and by providing better documentation to use when a failure is discovered.

Integrating maintenance programs into the machine/monitor program complex—standardizing servicing procedures

System/360 maintenance programs will be used in what may be called a two-dimensional application environment.

* The retry threshold for an instruction is that point in its execution after which some initial operands are no longer available.

In the simplest case, a maintenance program has all system facilities available for the duration of its running. In a more demanding environment the program must restrict itself to a subset of the host system and be prepared to yield control to the operational monitor immediately upon the occurrence of events, external to the subset, that require a fast response. So the first dimension is the presence or absence of a monitor. The second dimension is whether there is one or more than one CPU in the system.

Program compatibility, the use of one set of I/O devices and control units in all models, and having basic storage units appear in several models, suggest the goal of a single set of maintenance programs as a basis to support these devices for the entire line. To leave the programs for a given unit basically unchanged from one environment to another, it was decided to interpose a buffer program between the programs for each unit and the external environment, and at the same time to provide in this program, called a *diagnostic monitor*, those facilities (utility functions) that are common to many of the unit programs. The problem of integrating the maintenance program package into the machine/monitor program complex has in this way been reduced to the problem of matching the diagnostic monitor to this complex.

The necessities of operating both in a stand-alone environment and in a monitor-controlled environment, and the decision to centralize common functions have led to providing the following facilities in the diagnostic monitor: (a) self loading, through the initial-program-loading operation; (b) system initialization; (c) initial handling of interruptions and their distribution to the unit diagnostics; (d) decoding of externally-originated messages and taking appropriate action on these; (e) recognition of priority interruptions;* (f) loading unit diagnostics; (g) control of the sequence in which the units are treated, including the cases in which several units are handled simultaneously; and (h) run option (e.g., error printouts) control.

Primarily because of the severe size constraints imposed on the diagnostic monitors by the smaller System/360 models, and to minimize the diagnostic monitor storage demands when sharing the system with an operational monitor, several diagnostic monitors are provided. They constitute a family in the sense that diagnostic monitors have upward-compatible facilities. The size reduction is achieved by the expedients of reducing the number of features and providing some in simpler form.

Initial versions of three members of the diagnostic monitor family have been in operation on System/360 prototypes since the latter part of 1963.

* A priority interruption is one that arises outside the domain defined for the diagnostic monitor by the operational monitor. Without exception, a priority interruption causes the diagnostic monitor immediately to return control of its domain to the operational monitor program.

From the viewpoint of a diagnostic monitor program, the problem of adapting to changing environments is largely the problem of how to communicate with the outside world. In this context the outside world falls naturally into four classes: the unit diagnostic programs, operational monitor programs, other diagnostic monitor programs (in other CPU's), and operators and customer engineers.

All communication by a System/360 diagnostic monitor with its environment is through standard program interfaces. As used here, a program interface consists of a means for transmitting intentions (passing control information) and a means for transferring parametric information and other data. These interfaces are standard in the sense that they are identical in all diagnostic monitors of the family.

The *diagnostic monitor/unit diagnostic interface* provides a direct solution to the problems of integrating into one system a collection of unit-diagnostic programs that have been produced by independent design groups, and using the same unit diagnostic with different monitors. Common agreement is first reached on the definition of this interface, and then all coding conforms to this standard.

Frequently-used functions are provided as subroutines in the diagnostic monitors where they are accessible to the unit diagnostics through this interface.

Control of run options is exerted by the diagnostic monitors through this interface. The diagnostic monitor exerts a normalizing influence on commands directed to the unit diagnostics from the outside world, so that regardless of their origin, messages are always presented to the unit diagnostics in a standard way.

The *diagnostic monitor/operational monitor interface* provides a solution to the problem of matching several diagnostic monitors to an even larger number of operational monitors. Just as important from a practical point of view is the fact that operational monitor design and diagnostic monitor design need not be so carefully phased in time. But perhaps the most attractive advantage is that users who provide their own operational monitor can integrate a suitable IBM maintenance program package into their system simply by including in their program a section that provides the operational monitor side of this interface.

It is through this interface that the operational monitor passes job requests to the diagnostic monitor. Included in such job requests are the definition of a diagnostic monitor's domain and the system units to be tested. The control of this domain is returned to the operational monitor through this interface on the normal completion of job-request servicing and priority interruptions. Results of requested runs are given to the operational monitor through the interface at the time control is returned.

The *diagnostic monitor/diagnostic monitor interface* is

required for running several unit diagnostics under the coordinated control of more than one CPU. For these applications there must be a diagnostic monitor in each of the participating CPU's. These diagnostic monitors communicate with each other through the diagnostic monitor/diagnostic monitor interface.

The *diagnostic monitor/human interface* provides a single set of procedures and messages that the operator and customer engineer may use to request and control the running of maintenance programs on any System/360 machine.

It is through this interface that run options may be manually specified and output messages and data printed. Because of this capability there is no requirement for the human user to learn and compensate for the inevitable individual differences that appear in the unit-diagnostic programs. This standard interface facilitates the user's learning of the procedures applicable to any System/360 installation, regardless of which model is installed. It is especially valuable for learning the procedures applicable to a system configuration in which several different models are present.

Summary

Among the serviceability features for the System/360, the following were designed to supplement standard hardware:

a) Error detection circuitry (data flow and controls).

b) Freezing and then transferring to main storage the pattern of the CPU's internal elements which defines the present machine state.

c) Branching under program control to a specific monitor location upon given signals.

d) Transferring a pattern from main storage to the CPU's internal elements.

e) Advancing the machine clock a specified number of cycles under independent control.

f) Special comparison circuits to compare machine states with the actual states.

g) Control of this hardware by a special instruction or by independent circuits.

As indicated in the text, not all models have all features.

These features were supplemented by the following new programs and procedures:

a) Use of a 7094 to (1) produce bit patterns to test almost all of the System/360 CPU components (for the larger models), (2) sequence the tests on magnetic tape or disks to allow resolution of circuitry failures to a few replaceable circuit cards, and (3) produce documentation, updated by engineering change level, to specify the suspected cards.

b) Use of the ROS as storage for diagnostic routines (on the smaller models) to (1) exercise the circuitry in increments of single steps at full machine speed, (2) isolate the error after detection by use of checking circuitry, and (3) allow these routines to be used by setting maintenance console switches.

c) Use of FLT (in the larger models) or ROS test procedures to diagnose a system with frequently recurring error indications.

d) Programs to operate as part of the operational monitor to (1) produce the error environment record, (2) use the machine state pattern to permit instruction retry, and (3) use the diagnostic monitor to test parts of machine systems, as directed by operator or operational monitor.

e) Programs to operate independently to (1) analyze the error environment record to aid the customer engineer in the diagnosis of intermittent failures, and (2) control diagnostic procedures efficiently by diagnostic monitors.

After the initial effects of the learning curve have passed, this planned combination of hardware and software procedures is intended to (a) reduce the maximum duration of service calls, (b) reduce the median duration and mean duration of service calls, and (c) match a single package of maintenance programs and procedures to a large variety of operational monitor programs and models.

Appendix

Testing and diagnostic procedures that rest upon the analysis of computer circuits is feasible only if this analysis is performed automatically. The first programs to do this analysis were produced by Roth, et al.[7] and Forbes, et al.[6] The techniques of doing the analysis for an entire computer system (as opposed to logic without feedback) and the definition of the hardware requirements to implement these tests for an entire system were first described by Maling and Allen.[4] These techniques have been used very successfully to analyze such systems as the IBM 1418, 1419, 1428, and such machine features as storage protection for the IBM 7094 at MIT. The knowledge gained from designing and using these first systems is being used to produce improvements in these systems. The new system uses as input the Design Automation Logic Master Tape.[8] It also can analyze more than twice as many logic circuits as the previous program and even with its increased capabilities takes less computing time on the IBM 7094.

The general flow of the FLT generating programs complex is shown in Fig. 3. The information defining the logical structure of the computer nets is extracted from the

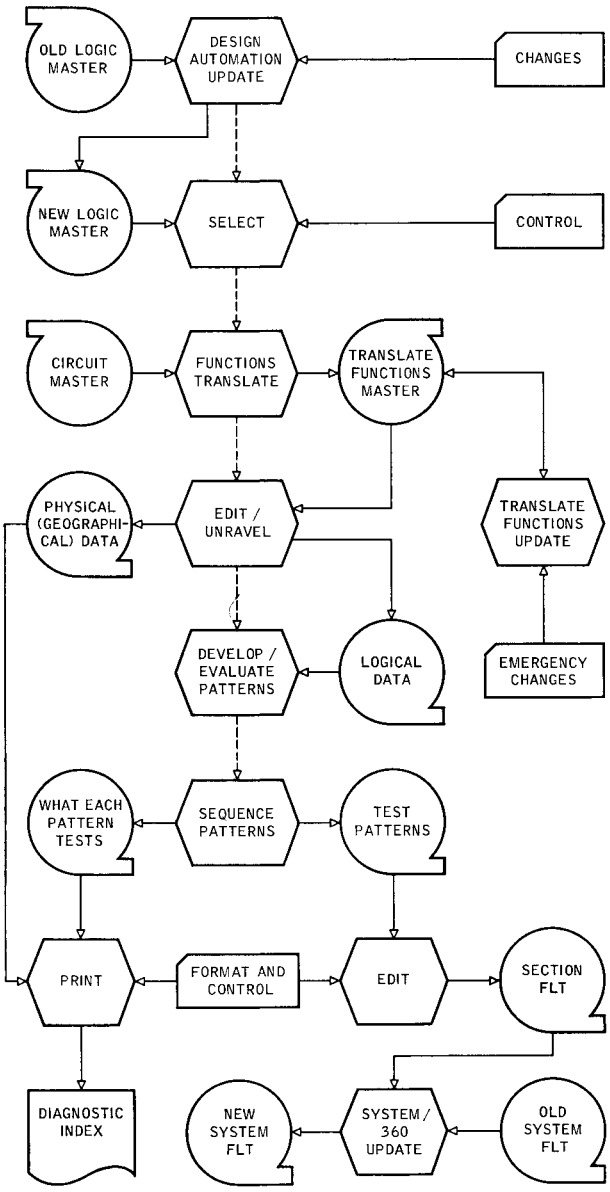

Figure 3 **Fault Locating Test generator program complex.** *This complex is for use on the IBM 7094 to precalculate test patterns.*

Design Automation Logic Master Tape. Then the combinational logic nets are unravelled from the sequentially connected nets, the physical location information is separated from the logical, and the latter is put into a form that is convenient for analysis. (Not mentioned in this outline are the many necessary routines for printout and for error analysis.)

The next part of the program consists of the routines to analyze heuristically a logic tree network. The routines trace forward from an assumed line failure to the tree output, and backward to the net inputs. At each step, logical values as determined by the circuit elements are assigned to the lines. This may lead to logical impossibilities and abandoning this attempt. The net inputs define a combination of input conditions that should determine if the particular line tested is working correctly. Each input and output line to a circuit element is analyzed.

After a pattern has been developed, it is evaluated to determine what lines it actually tests, and if it tests any lines not previously tested. The development and evaluation continue until a set of tests that will indicate the malfunctioning components collectively is obtained.

The tests themselves do not necessarily provide enough resolution to locate the malfunctioning component. However, after they are organized into a sequence in the form of a decision tree, this resolution is obtained.

The final routine of the program complex translates the decision trees and associated test patterns into pattern test data. These data are then transformed into the form necessary for the particular pattern testing circuits that are associated with a specific IBM System/360 model.

Acknowledgments

Because of the paucity of published material related to maintainability and reliability, it has been impossible to document in the usual way (by citing references) the contributions of all the people who contributed to this project.

The detailed engineering implementation of serviceability hardware for the various models was under the direction of N. E. Beverly, W. S. Demmer, M. E. Homan, and E. W. Miller for the larger systems. A. Peacock and L. Mulock were in charge of implementing serviceability design for the smaller systems.

K. Maling directed the programming for the complex of programs that produces the FLT patterns.

R. B. Ormes, A. Heineck, H. Morrow, H. C. Oppeboen, and D. C. Burnstine worked on the total maintainability plan.

F. A. Carlson, R. C. Williams, and G. R. Wilmot contributed heavily in the design of the diagnostic monitors. The first multiprogramming diagnostic monitor was implemented in the IBM British Laboratories.

To these and to all others who helped, the authors render their thanks and apologies for any unintentional omissions.

References

1. K. Maling, "Classes of Component Failures and a System Design to Facilitate Their Location," IBM Data Systems Division Technical Report TR 00.1029, (Rev. March 10, 1964).
2. E. M. Davis, W. E. Harding, R. S. Schwartz, and J. J. Corning, "Solid Logic Technology: Versatile, High-Performance Microelectronics," *IBM Journal* 8, 102 (1964).

3. C. A. Bennett, "Application of a Learning Curve to a Maintenance Problem," 2nd Annual Quality Control Symposium of the Dallas-Fort Worth Section of the ASQC, March 16, 1961.
4. K. Maling and E. L. Allen, Jr., "A Computer Organization and Programming System for Automated Maintenance," *IEEE Trans. Elect. Comp.* **EC-12**, No. 6, 887–895 (December, 1963).
5. R. D. Eldred, "Test Routines Based on Symbolic Logic Statements," *Jour. ACM* **6**, No. 1, 33–36 (January, 1959).
6. R. E. Forbes, C. B. Stieglitz, and D. Muller, "Automatic Fault Diagnosis," AIEE Conference on Diagnosis of Failures in Switching Circuits, May 15–16, 1961.
7. J. M. Galey, R. E. Norby, and J. P. Roth, "Techniques for the Diagnosis of Switching Circuit Failures," Proc. 2nd Annual Switching-Circuit Theory and Logical Design Symposium, AIEE, September, 1961.
8. P. W. Case, H. H. Graff, L. E. Griffith, A. R. LeClercq, W. B. Murley, and T. M. Spence, "Solid Logic Design Automation for IBM System/360," *IBM Journal* **8**, 127 (1964).
9. G. M. Amdahl, G. A. Blaauw, and F. P. Brooks, Jr., "Architecture of the IBM System/360," *IBM Journal* **8**, 87 (1964).

Received January 24, 1964