Procedures for program testing associated with implementation of a large complex real-time system are discussed step by step.

The discussion includes testing both in a simulated environment and in real time.

Final testing and monitoring of system performance are also briefly considered.

Notes on testing real-time system programs

by M. G. Ginzberg

Pending appearance in the literature of a systematic and complete treatment, these notes are intended as a partial check-list for the systems engineer installing a real-time system. Although based on the implementation of an airline reservation system, the notes are generally applicable to large systems of similar complexity.

Real-time program testing involves additional problems caused by the time-dependent interaction of operational programs, control program, complex hardware, and random input to the system.

Since errors are difficult to isolate and correct once such systems are operational, adequate pre-installation testing is important. The effort of developing testing routines is commensurate with that required for the control program, and the volume of code in the routines can be expected to exceed that of the control program.

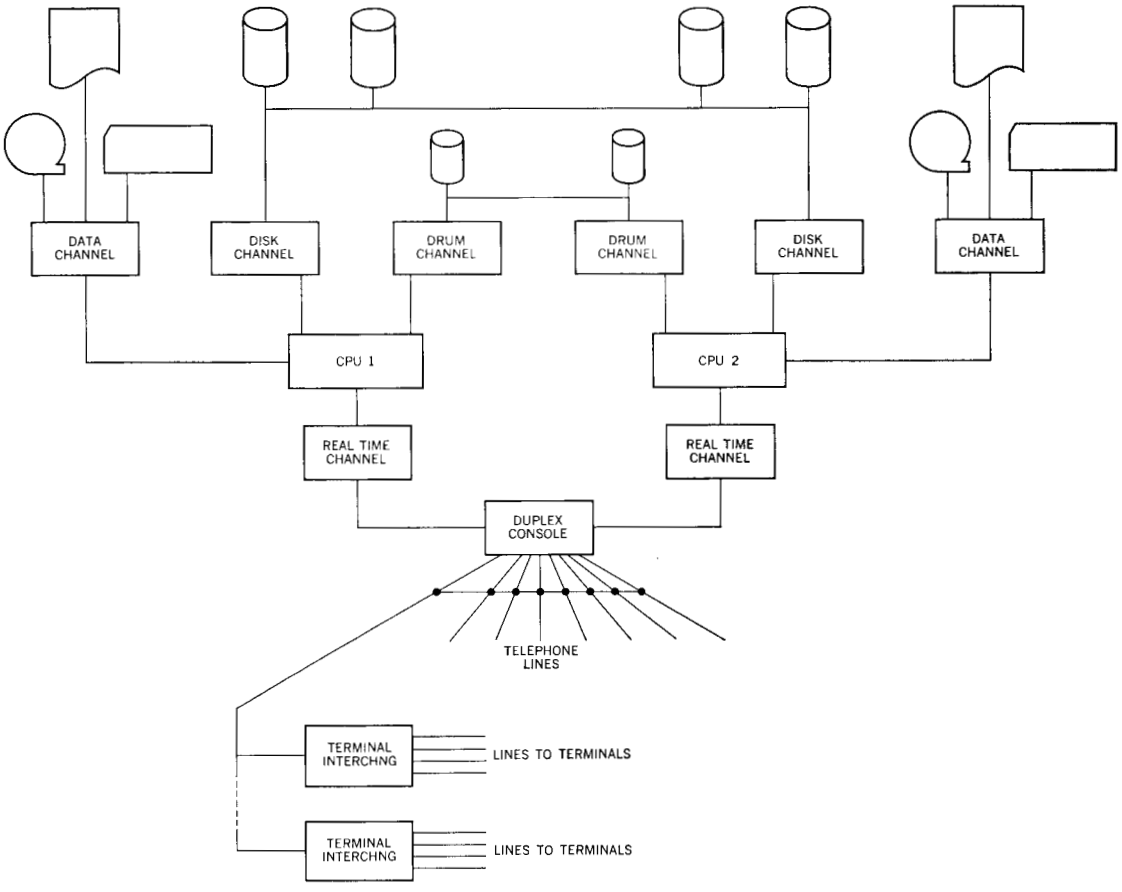
The type of system under consideration is suggested by Figure 1 and is assumed to have the following general characteristics:

system characteristics

- A large number of terminal devices have immediate access to the system, providing a high volume of random inputs.
- Most input messages require an immediate response to the terminal device originating the message.
- File records must be retrieved, updated, and in some cases created, while a message is being processed. Record retrieval generally requires a sequence of file events.
- The message volume necessitates several simultaneous inputs in core memory, each being processed in turn until further processing must wait for completion of a file event.

- Only the more frequently used subroutines can be contained in core. Since many programs must be read from file on demand, their size must conform to some fixed multiple of a file record. Long programs must be segmented.
- A program segment used to process a particular message that contains an interruption for retrieval of a file record may be used to process several other messages before continuing to process the original message.
- A message may be read into any one of a number of core blocks (entry blocks), and similarly, a read-in program segment may be executed in any one of a number of areas in core.
- File protection must be provided so that only one message at a time can update a particular record.
- File maintenance programs must be designed to run in an on-line computer, permitting continued use of the file records for message processing while file updating is in progress.
- Deferrable processes must be initiated automatically when the system load permits. Other processes may require initiation at specified times.

Figure 1 System configuration



control program

These system characteristics, although not exhaustive, indicate the types of problems encountered and the complexity of system implementation. To maintain order in such a system, a control program performs the following functions:

- Control of input and output with respect to the remote terminal devices and the files
- Allocation of working storage in core to messages in process
- Maintenance of queues of work in process
- Control of program branching and of program read-in
- Dynamic control of file storage
- Monitoring the communications network

A set of control program macro instructions form the interface between the control program and the operational programs. The control program macro instructions, augmented by a subset of the regular op-code repertoire of the basic machine,[1] define the system within which an operational program is coded. An operational program merely executes the appropriate control program macro to retrieve or file a record, to obtain or release working storage in core, to obtain or release space in files (creating a new record or eliminating an existing one), to branch or return to another operational program, to output a message to a terminal device, to write a record on tape, or to terminate processing of a given message.

Within the framework provided by the control program, most operational programs can be written without considering them as part of a multiprogrammed system. However, the major restriction must be observed—program modification is not permitted across points at which processing is interrupted pending completion of a record retrieval. It is important to note that the same remarks are not applicable to file maintenance routines that run in real time, nor to subroutines that provide interlocks between file maintenance programs and programs that process normal input messages.

Testing in a simulated environment

Since testing of operational programs may have to commence prior to availability of all system hardware, of a working model of the control program, or of both, an *environment simulator* program is required.

environment simulator

The environment simulator program is written to permit testing of operational programs on a standard data processing machine. The operational programs run under simulated conditions as if they were operating within the real-time control program of a machine to which all of the devices constituting the ultimate system are attached. The simulator program contains expansions of all control program macros that modify the entry block and other working storage in the same manner as the macros in the actual control program.

Design of the environment simulator is dictated not only by the hardware and control program simulated, but by the computers available for pre-installation testing as well as by the computer at the ultimate installation. For example, if core memory is large but tapes and files are limited, it may be advantageous to simulate tapes and files in core.

The simulator should be coded for a maximal amount of debugging capability. First and foremost, it must be possible to obtain a snapshot or a panel dump at any point desired in an operational program. Dumps of the entry block, working storage chained to the entry block, or any records retrieved by the operational program should be readily available at any step.

The environment simulator should enforce restrictions implied by the control program. Improper use of control program macros should be prevented. The contents of any registers that would not be preserved in real time across a particular macro should be destroyed. It is highly desirable that the simulator detect illegal attempts at program modification, i.e., across interruptions in processing pending record retrieval, or across the boundaries of program segments. Violation of segment length restrictions should be detected but not prohibited; the programmer should not have to continually resegment his code in the course of debugging.

Finally, the means should be provided to run in one pass a series of cases through the program or programs being tested. In other words, a test deck should contain the programs to be tested plus one or more sets of input. Generally the input for one case consists of an entry block containing the input information expected by the first subroutine in the group being tested, plus any records that have supposedly been retrieved prior to entering the programs being tested or that are supposed to be retrieved in the course of the test run. Index registers, accumulators, and other registers must be preset for all cases, except when the first subroutine entered is the one that first receives the input message from the control program.

An environment simulator program is a valuable test tool not only during pre-installation testing, but even after the real-time hardware and programs are available. If properly constructed, the program provides much more debugging flexibility than during testing in real time.

record generator

Construction of test data, which can be exceedingly time consuming, is greatly facilitated by a fairly simple *record generator* program. Essentially, the program contains a library of data formats, including the location, size, character (alpha or numeric), and normal contents of each field of each record type from which it generates records required for testing. To obtain a record, a set of cards is prepared that contains the record type identification, the name and contents of all fields for which an input is mandatory (or for which the user wishes other than the normal contents), and the file address of the real or simulated record. The record generator should also be able to generate entry blocks containing

either unprocessed or partially processed inputs. As an added refinement, the record generator might be programmed to compute the contents of certain tally fields in a record by referring to specified contents of other tally fields.

The program should be coded to facilitate use with the environment simulator and with the file loader discussed later in this paper. When testing with the environment simulator, it would be highly desirable that record generation and testing take place in a single pass.

The environment simulator and the record generator should be ready for use when operational program coding commences. As an alternative, special test drivers must be written and records constructed manually for programs that are coded before the test tools are available. This procedure is time consuming and less reliable than testing with the environment simulator. In general, a more automated test environment reduces the possibility of "validating" individual programmers' misconceptions (of record formats, for example) by tests.

unit test

The *unit test* of individual subroutines is the first step in debugging a system of operational programs. In this respect, a real-time system does not differ from any other program system; coding and debugging should proceed from the simple to the complex. The innermost subroutines should be coded and debugged first, testing them with all possible valid inputs and with a sufficiently large sampling of erroneous inputs to check all of the error branches. This process eliminates the need for dummying internal subroutines, a procedure which can only serve as a possible source of errors that have to be eliminated when the actual subroutine replaces the dummy.

A *library tape* should be maintained of all debugged programs. Provision should be made in the environment simulator for calling programs from the library tape. This provision reduces the size of the test decks and ensures that all programs using a particular subroutine actually use the same version. When planning the structure of the library tape, possible use of the tape for generating a real-time system tape should be considered.

package test

The *package test* is the final stage of testing in the simulated environment. A package is a logical, functional subdivision of the operational program system, consisting of the complete processing of a particular input message or of a related group of inputs. A long and involved normal path for processing a particular type of message should constitute a basic package. The exception paths, which generally add programs to those of the basic package, should be set up as additional packages. On the other hand, a group of messages whose processing is fairly simple and that uses a substantial amount of shared code may constitute a single package.

Specifications for these tests are best prepared by persons familiar with both the programming and the over-all functions of the system; the most likely candidates for this task are those

who prepared the program specifications. All possible variations in the inputs and in the conditions encountered in processing them should be specified. The more complete these specifications are, the less debugging is necessary during field tests and dual operation.

Testing in real time

As soon as the first package has been completely tested in the environment simulator and a working model of the control program is available, testing should commence in real time. These test runs cannot be run on a standard installation, but require the real-time hardware to be ultimately installed in the system's data processing center. This phase of debugging should be scheduled to start while the hardware is still in the test cell. The early start is of particular importance to the prototype system in a machine line.

Testing of the first package in real time provides the first meeting of the control program and the real-time test tools with actual operational programs. Prior to this test, sections of the control program have been debugged by the use of specially written driver programs, and the assembled model of the control program has been tested by the use of pseudo-operational programs that function as "macro exercisers." The real-time test programs have been similarly tested. No matter how elegant testing of the control program has been and regardless of how adequately operational program packages have been checked out in the environment simulator, program interaction in real time is the critical test. For the first time, actual system hardware and programs process actual inputs and retrieve, create, and update actual file records. In debugging the control program, no test tool is quite as rigorous as the actual system of operational programs.

It is obvious, then, that the real-time testing of the first operational program package serves to determine any modifications that may be required in either control program or the real-time test tools. Once the latter are operating properly, the orderly testing of other real-time packages can proceed. The functions of the test tools are more readily comprehended if the nature of real-time testing of the operational programs is examined first.

single-thread testing

Single-thread testing is the first step in real-time debugging. In this phase of testing, a series of terminal device inputs is processed in serial order, hence the designation "single-thread." The specifications for these tests have already been prepared for the package tests described in the previous section.

A test run generally starts with set-up messages to create and/or modify records needed for the test run. The main part of the run consists of a series of inputs as specified in the test specification. Whenever an input causes the updating of a record, this record should be displayed before and after the message causing the updating. Similarly, when the response to a message de-

pends on the status of a record or some of its fields, the record should be displayed prior to input of the test message. The documentation of a test run should include all inputs and their associated responses, as well as the record displays just mentioned.

The package tests with the environment simulator should probably use individual records, or small groups of records, generated as required for the specific packages. For real-time testing, a system of records is needed that is independent of the programmer running the tests.

A *test record system*, consisting of a complete and mutually consistent miniature of the ultimate record system, should be extensive enough to cover all variations in record length, chaining, etc. Representations of all records that would be present in an initialized system are required. Those records for a test run that are normally created in real time as part of the processing of an input message should be created in that manner by set-up messages. These records should not be part of the test system except where prevented by the order in which programs are checked out.

multi-thread testing

Multi-thread testing is needed to eliminate multiprogramming errors that have escaped detection during the single-thread runs. During this phase, it should be determined, for example, that the programs being tested indeed process different inputs simultaneously and that file protection is functioning properly (i.e., several "simultaneous" inputs, all of which update a particular record, have the correct cumulative effect on the record). One of the more complex situations to be tested involves the simultaneous input to the system of several messages all of which would, if input alone, cause the creation of a particular file record which should only be created once. File maintenance programs that run in real time should be tested while a typical transaction mix is being processed.

It should not be expected that many errors are uncovered in multi-thread testing. However, errors detected are of a particularly complex nature; hence they are exceedingly difficult to find if they are not forced to occur in a systematic manner. Even though this is accomplished, involved errors will still be detected later during volume testing.

All of the real-time testing discussed above could be accomplished simply by message input from terminal devices. However, this is both excessively time consuming and error prone.

terminal simulator

A *remote terminal simulator* program is an effective tool for expediting real-time testing. The simulator's function is to present input messages to the control program so as to appear to have been input from an actual terminal device. If the system contains different types of terminal devices, it should be possible to simulate the input from any and all of them.

The remote-terminal simulator, reading a series of messages from tape, should be capable of simulating the existence of a reasonable number of terminals. For purposes of multi-thread testing, the simulator should be able to present to the system several input messages simultaneously; for more realistic and

more efficient testing, it should also be possible to deliver messages in an asynchronous manner. When running in the former mode, the simulator gives the system one message from each of the terminal devices being simulated and then waits for a response to each of these inputs before entering any additional messages; in the latter mode, the response to each individual message triggers the input of the next message from the same simulated input device.

The input to the simulator should consist of a series of messages identified as to the simulated originating terminal. It is desirable to be able to split one run into several phases so that, whether running synchronously or asynchronously, all messages in one phase are completely processed before the program proceeds to input messages of a subsequent phase. This permits separating the set-up messages from the actual test as well as from any other desirable divisions of the test run.

The simulator output should be a listing of the input messages and the responses to them, grouped by simulated input device.

It should be fairly obvious that the terminal device simulator just described permits remote testing in real time for both single-thread and multi-thread testing—the former being merely a special case of the latter. The simulator also greatly expedites retesting and volume testing, which are discussed later.

A *simulator input pre-processor* program, used to prepare the simulator input tape, should edit the input messages for correct format, missing characters, etc. If this is done on peripheral equipment (e.g., a 1401), much valuable system time can be saved.

A *real-time systems tape* is needed to load the operational and control programs. The function of this self-loading tape should be to write the file program segments (the bulk of the operational programs and the low-usage portion of the control program) to files, and to set up the core load (the major portion of the control program, certain high-usage operational subroutines, system control cells, and constant pool). The operational program portions of this tape should be generated from the library tape previously mentioned. Tight control should be exercised to keep both of these tapes consistent.

A number of utility programs facilitate real-time testing. Those described below do not necessarily constitute a complete list. The utility programs should be initiated by a terminal set message. Most of these programs will be needed in the course of simulator test runs during which simulated terminal set initiation can proceed without operator intervention. In general, it is highly desirable to have operator communication with the system via a terminal device message rather than by console manipulation. This arrangement permits programmed editing of the operators' actions and greatly reduces the system's vulnerability to operator error.

macro trace

The *macro trace* program is the prime real-time debugging tool. The necessity for such a routine arises from the difficulty

in using the usual varieties of snapshot and trace routines in real time. For example, use of the type of debugging macros available in 709/90 SOS can easily cause a lengthy program segment to overflow. It is difficult to see how the instruction substitution type of snapshot routine could be implemented for operational programs that may be executed in any one of ten or twenty different core areas.

The closest analogy to a macro trace program is a branch trace. The macro trace, however, can be designed to yield much more information. This trace program is intimately associated with the control program whose macros are actually traced. Various options should be provided. For a simple flow trace, for example, the program should trace all entries to, and returns from, program segments, giving at each of these points the macro traced, the program segment in which it was executed, its relative location in the program segment, and a panel dump. Additional options should be provided to trace larger subsets of control program macros, up to and including a trace of all macros, in order to make available more detailed flow traces. It should also be possible to obtain a dump of the entry block and all associated working storage at each program entry and exit. Other options would allow a trace of all I/O events, complete with a dump of the record being retrieved or filed.

Terminal set inputs should be used to start the trace, specifying the desired trace options, and to stop the process. At first, it may be adequate to initiate a trace of the processing of all messages received from the terminal device that is entering the start trace message, since usually only one message is traced. Later, it becomes desirable to add the ability to limit tracing. Limited tracing reduces the amount of trace output when testing the routines that do appreciable looping (such as file maintenance). Once operational, the system is useful in tracking down obscure errors that may obtain a trace of a specified program segment, or segments, each time the program is initiated, regardless of the source of the message that caused entry of the segment.

Trace output and remote terminal simulator output should be written on the same tape. This arrangement is more useful, since the trace can immediately follow the message being traced. Also, the likelihood of operator error (failure to mount a trace tape when needed) is reduced.

utility programs

A *post-processor* program should be provided to format and edit the output of the remote set simulator and the macro trace. Like the input pre-processor, this function should be assigned to peripheral equipment if possible.

A *file loader* program is needed to load the test record system. It is exceedingly important that this program be initiated by a terminal set message, since most test runs are preferably started with a reload of files to ensure a clean record system, rather than taking a chance on the condition in which the files were left by the preceding test run.

A *record dump* program is required for the record displays mentioned earlier in the discussion of single-thread testing. The ability to address records indirectly must be provided to permit the display of records created in real time and assigned a random file address by the control program. The number of levels of indirect addressing required depends on the system file organization. The record dump program should also be able to display core words.

A *record change* program is needed for those tests in which it is impossible or inconvenient to use normal terminal device messages to set up the record condition required. The same level of indirect addressing should be provided as in the record dump routine. The ability to alter core words should be included.

A *programmed delay* routine may be needed. For example, consider the case of a test that is run to check out, by a normal terminal set entry, the retrieval of certain records set up, filed, and indexed in real time. The program that sets up these records checks the input message for correctness and sends a response to the input device before proceeding with editing, filing, and indexing the record. To ensure that all of this relatively time-consuming processing is completed before the simulator initiates the record retrieval message, a delay device is needed. This delay can be in the form of a message that receives no response until all other operational program activity in the system has ceased.

retesting

Retesting eliminates the introduction of errors in already checked-out packages while correcting errors in newly tested packages. As the single-thread testing of each package is completed, the test deck should be added to a master simulator deck. This deck must be rerun periodically, and the results of each run should be compared with previous runs.

Retesting can be expedited by a *simulator output tape compare* program. This routine compares the output tapes of two simulator runs expected to be identical and lists any discrepancies discovered. To simplify inspection of this listing, the simulator post-processor should number the lines of its output listing sequentially; then the listing can be reduced to lines that differ from the original. In test cases that display records, some words (e.g., those containing chain addresses) may vary from run to run. For this reason, it is preferable that only the line containing the discrepancy be displayed, rather than the entire message.

The above utility routines are basic. Other utility routines required vary from system to system. Some functions to be considered are the communication between a remote terminal and the personnel at the data processing center, and the insertion of comments in simulator runs to ease interpretation of the output.

Scheduling and quality control

A realistic, detailed schedule and an adequate scheme for assuring

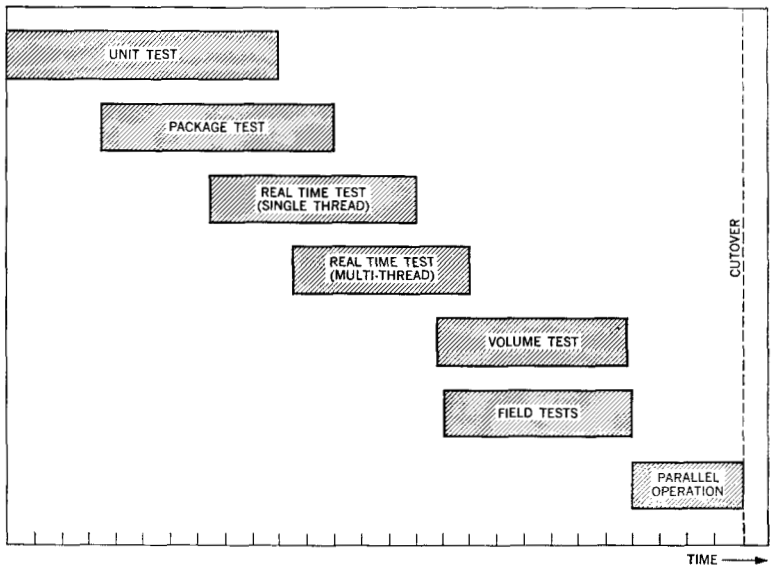
the quality of the programs produced must be instituted when programming begins.

An over-all schedule, as suggested by Figure 2, must be augmented by detailed scheduling down to the level of the smallest subroutine. Intermediate targets must be established, the dates of program arrival at each of these steps must be forecast, and responsibility for attainment of these goals established. A suitable set of intermediate steps in unit testing may be: program coded, test data complete, first test case run successfully, all test cases done. A similar breakdown should be used for scheduling package test and single-thread real-time test. If adequately detailed schedules are prepared and continuously re-examined, delays can be detected early enough to add personnel and take other corrective measures (or in the worst case, to adopt a more realistic schedule) before a major difficulty arises.

Control over the quality of the code produced and of the adequacy of testing should be established early. The code should be spot-checked by experienced personnel to detect loose code, poor programming practices, and deviation from program specifications. Prompt correctional action benefits the individual coder as well as the system. Unit test documentation should be thoroughly reviewed for adequacy. The package test specifications and the output of the remote-terminal simulator fulfilling these specifications must be audited with extreme care. It would be well to consider separating the control responsibility from that of producing and testing code.

It cannot be overemphasized that the amount of difficulty encountered at each step varies inversely with the thoroughness of the preceding phases of testing. For this reason, tight control

Figure 2 Testing schedule



over the quality of coding and debugging pays dividends in terms of the total effort to install the system.

Final testing

The operational program system should be in a reasonably healthy state by the time the single- and multi-thread real-time tests are completed. However, some program errors that evaded even the most conscientious detection efforts remain at this point. These errors include exception cases not anticipated, unexpected interaction of simultaneous entries, as well as the results of plain oversights in previous debugging.

Experience indicates that the most serious errors remaining in operational and control programs are associated with file record utilization. Two types of error predominate: (1) returning the address of a no longer needed record to the control program more than once, and (2) failure to return such records. The control program can be designed to render the former type of error completely harmless and to minimize the difficulty in recovering records lost due to the latter. Nonetheless, a major effort should be made to eliminate errors of this nature during the previously discussed phases of testing, as well as during the final testing.

volume testing

Volume testing involves the simulated operation of the entire system. The record system, either in total or a fairly large and representative subset, is initialized as though the system were about to become operational. A large sample of the normal transaction mix, gathered from actual transactions in the non-mechanized (or partially mechanized) environment that the real-time system is to replace, is input via the remote terminal simulator. As the transactions are input, the system clock and system calendar are advanced on an accelerated basis. The test should span a sufficient length of simulated time to include all periodic file maintenance runs that are part of the operational program system.

While conducting volume testing, the simulator should be run in the asynchronous mode, so that the runs simulate the real system as closely as possible. Transactions that would have occurred simultaneously with the file maintenance runs should be run in that manner. Frequent checkpoints should be taken and the system records audited to ensure that the cumulative effect of the transactions on the contents of the various records is correct. These record audits may be simplified by some simple off-line programs that process the terminal simulator input to compute its effect on the record system.

It may be deduced from the above discussion that a very tight audit should be made of file record usage during the course of volume testing. The number of records obtained from and returned to the control program between each pair of checkpoints should be predetermined and then verified during the actual test runs. Off-line programs, similar to those suggested above to determine record contents, should be used to ascertain the

correct record usage. Provision should be made in the control program to maintain a count of the number of file records available at any time. This count is needed not only for the test runs presently being discussed, but also to monitor performance of the system when operational.

A *record usage trace* proves valuable when the record usage audits show that losses are occurring. This trace actually consists of several programs. A special real-time trace will record on tape every file address removed from, and every address restored to, the pool of available records. The special trace also records the program segment that requested or returned the record. The tape is processed off-line, pairing file storage requests with subsequent releases, to maintain a cumulative tape of those records that are in use. The tape shows the date when each record was obtained from the record pool and by what program segment. Further processing of this tape can develop statistics by program segment and age of records in use. These statistics should pinpoint any programs that do not return obsolete records to the record pool. The tape should also be able to determine multiple returns of the same record address.

A *file dump* program is necessary to capture the record system at checkpoints. When errors are discovered, this process permits starting from the last good checkpoint, rather than returning to the beginning. With a suitably designed control program, only the records actually in use have to be dumped. The dump should obviously be in the form of a self-loading tape.

field tests

Field tests should be scheduled concurrently with volume testing. The volume test input provides a suitable set of data for use during field test. The primary purpose of the field tests is to check out the communications network. Since data are entered by individuals, the tests also provide a check on the ability of the system to cope with variations and errors in input format introduced by human operators.

parallel operation

Parallel operation is the final step prior to system cutover. Depending on the number of terminals and stations involved, all or part of the system is placed in operation. To avoid system interruptions, records are maintained external to the system for backup. The external records may be provided by the real-time system itself. The parallel operation continues until system performance attains such a level that this precaution against system failure is no longer needed.

The necessity of testing new and modified programs continues when the system becomes fully operational. Some obscure program errors may be discovered even after many months of operation. Throughout the life of the system, functions are added to improve those already programmed. For testing such programs without disturbing the operating system, the test record system must be maintained. An area in the files should be reserved for these records, and it should be possible to access these records, rather than the normal record system, during debugging runs. Means

should be provided for easy alteration of existing routines, and for substitution or addition of new routines, for test purposes only. The programs needed to accomplish these functions vary with system design and mode of operation (simplex or duplex system, operated continuously or on specified shifts only). In the area of post-installation testing, therefore, this paper can only indicate the need for test tools, rather than describe them with a significant amount of detail.

Studying and monitoring system performance

The topics of studying and monitoring system performance are generally outside the scope of this paper. However, in view of their importance, and since they must be simultaneous with other efforts described in this paper, some brief comments are included.

System simulation provides the prime tool for studying system performance. Simulation studies are run during the pre-proposal stage to verify the ability of various proposed systems to handle the anticipated load. Data regarding input volume, transaction mix, and peak load should be as reliable as possible. On the other hand, the model of the system itself, particularly of the program components, may be quite vague at this stage. At best, a fairly good guess is available as to the number of retrievals and filings of records involved in each of the various transactions. The estimate of file events due to program read-in and of execute time in the control and operational programs may be extremely sketchy.

As the system programs evolve, more and more detailed information can be incorporated into the simulation. When complete, the operational program and data format specifications should be used as input to the simulator. At this point, the control program should be well enough defined to provide the execution time of all of its macros. The actual execution time in each of the operational programs and the number of segments in any program can only be estimated. When these items become known during the coding of portions of the operational program system, the true values should replace the earlier estimates.

In addition to a model of the system hardware and programs, the input to the simulator consists of the transaction mix and the input message rate. The output consists of the following items which are functions of the input rate:

- Core utilization—for entry blocks, working storage, and program segments
- CPU utilization
- Length of file and I/O queues
- Response time

The purpose of simulation studies that are run during the implementation phase is (1) to verify that the proposed system indeed suffices to handle the anticipated load, and (2) to determine

the optimum allocation of core between control program, permanent core operational subroutines, constant pool, working storage, and read-in program segments. The simulation runs can also be used to ascertain which operational program subroutines should be kept in core.

Once the system is actually in operation, it provides the best possible source of data regarding its load and performance. An analysis of the input can be made to obtain the actual transaction mix. An ability should be provided to gather system performance data dynamically, specifically the items mentioned earlier as output of the simulator.

A continuing, coordinated effort in system simulation and data collection helps to ensure an optimum system organization and provides sufficient lead time for system modification necessitated by load growth.

ACKNOWLEDGMENT

This paper is based on the efforts and analyses of many individuals associated with the airline reservation system referred to in the text.

FOOTNOTE

1. Operational programs do not use any I/O instructions, halts, or certain other instructions that are reserved for the use of the control program.