AT&T Technical Journal
Vol. 64, No. 1, January 1985
Printed in U.S.A.

System 75:

GAMUT: A Message Utility System for Automatic Testing

By C. J. LAKE, J. J. SHANLEY, and S. M. SILVERSTEIN*

(Manuscript received July 11, 1984)

GAMUT is an automated testing tool that can verify message activity in message-based systems. This tool is unique because it can be customized for application to a variety of systems and used in all test phases of a development cycle. Using a single interface, rather than an array of port-specific interfaces, GAMUT can generate and verify load traffic through scripts, as well as provide an automatic record and playback mechanism for feature testing. This paper discusses the design philosophy and distinctive features of GAMUT as well as its architecture. It illustrates message definition and the script command language by building a sample test script of a station-to-station call on the System 75 office communications system and concludes with a discussion of challenges and problems encountered while using the tool to test System 75.

I. INTRODUCTION

The purpose of system testing is to ensure that a newly developed product performs according to established requirements and satisfies customer needs. One approach to testing is to manually exercise systems according to prescribed test plans; testers augment this approach with tools that automatically apply input and verify operations.

* Authors are employees of AT&T Information Systems Laboratories, an entity of AT&T Information Systems, Inc.

Such automatic tools frequently replace system peripherals and consequently are application-specific. However, the emphasis on structured design in new products has also changed the perspective of testing and consequently the architecture of the automated testing tools; a single test tool can simulate the activity of all peripherals.[1]

This paper describes an automated testing tool, GAMUT, that monitors and verifies message activity in message-based systems by simulating traffic at message interfaces within the system. GAMUT executes scripts, written in a programming-like language, that represent some feature operation of the system; sequences of stimuli are applied to the system under test and responses are verified automatically by the tool.

This paper discusses the design philosophy and distinctive features of GAMUT as well as the architecture of the tool. It illustrates message definition and the script command language by building a sample test script of a System 75 station-to-station call. Some alternative applications of the test tool are also suggested.

II. BACKGROUND: MESSAGE-BASED SYSTEMS

In the past, switching systems were designed with a well-defined, high-level message interface to peripherals or between processors; for example, in Automatic Call Distribution (ACD)* electronic switching systems there were message interfaces between the main switch and the local peripheral controllers as well as the management-information system.[2,3] However, the use of multiple-microprocessors and multiprocessing operating systems in real-time information processing systems has led to the incorporation of message-based architectures at internal implementation levels. Typically, such systems consist of a network of subsystems, implemented with hardware, software, or combinations of both, which communicate through message interfaces. Requests for service, called stimuli messages, are inserted into a subsystem by users or by other subsystems. The subsystem then produces response messages that may become input to other subsystems or that may be output to users on various system terminals.

In a complex system composed of many subsystems with many terminals, the combinations and sequences of stimuli and response messages are endless; an attractive method to exercise and test the design and implementation of such systems is to have a computer utility apply sequences of input messages to a selected interface and validate the resulting output.

In System 75, in addition to the internal interfaces between software

* Acronyms and abbreviations used in the text are defined at the back of the *Journal*.

subsystems, there is a complex interface between the system and communication terminals connected to it. Prior to connecting actual users to the system, it is necessary to test overall system performance under a wide variety of conditions and loads. Problems arise when these tests must be performed prior to the availability of either the interface circuits or the actual terminals.

Other problems, such as simulating actual communication connections or load conditions, occur when the system to be tested is complete but before it is practical to actually place heavy call volumes through the system. Another case is finding transient faults or faults that result from long sequences of state transitions.

Regardless of the application, systems designed with a message-based architecture contain independent subsystems and message interfaces that have the following characteristics:

- Control and data are passed in the form of messages.
- Activity at the message interface defines the operation of the subsystem.
- No subsystem possesses knowledge of how other subsystems perform their functions.
- Since the only way subsystems interact is through messages, the message interfaces form protective fire walls.
- By examining a trace of stimuli and response messages, one can determine if the system is operating correctly.

GAMUT exploits these characteristics and thus provides a powerful test tool for performing feature, load, regression, and range testing.

III. OVERVIEW—GAMUT FEATURES

GAMUT is a message utility that exercises and tests message-based systems by replacing part of the system at an interface boundary or by operating in parallel with the system. It interfaces with such systems either by processing messages entered interactively or by executing test scripts. A script is a file that contains sequences of stimuli to insert into the system under test, response messages to verify, and control information. The content of a script, taken as a whole, represents some feature operation or system function.

Since GAMUT is based on an architectural property of a class of systems and not on the design of a particular system, it differs in many ways from other test utilities.[4,5] It can keep pace with a product's development and provide unit, integration, system, and field-test functions.[6] The tool can be customized for application to a variety of systems by selecting an appropriate access interface and by creating a message database. It accepts parameterized scripts, by using scalar, array, and don't-care variables, for repeated applications of test cases,

and can generate traffic load tests using a single interface rather than an array of port-specific interfaces.

GAMUT consists of two major components, common message-utility software and a message-access module. The first part, the larger of the two, generates, validates, and manipulates messages and essentially remains unchanged for any test application. On the other hand, the access module connects the common software to the system under test—inserting and recording messages passing through a message interface of the system under test—and is usually specially designed for each test application. Thus, GAMUT can be applied to different test applications by customizing the message-utility software and "plugging-in" the appropriate access module. Other test systems can be too closely married to the system under test to permit this.

Since the utility can be configured to simulate a subsystem, it can be used early in development as a unit test tool. In addition, the subsystem simulation capability allows GAMUT to be used as a temporary replacement for unavailable subsystems or terminals. As development proceeds and more subsystems become available, the tool can remain in place and be used as an integration test tool. When the development reaches its final stages, GAMUT can be moved to the major message thoroughfare and drive the design through system tests. This is impossible to achieve with a tool that is coupled to a particular system point, and development teams usually have to develop different tools for each phase of testing.

When GAMUT is connected to a major message interface, it can be used to insert multiple, near-simultaneous stimuli and verify system performance. Other test systems can be used for load testing as well, but usually these tools simulate actual user activity and require a large number of interfaces to achieve this and are, therefore, as complex as the system under test. This tool achieves load testing more economically by driving one key interface internal to the system under test. Furthermore, since it can exercise independent subsystems, GAMUT can apply subsystem load tests to identify bottlenecks that limit system performance.

GAMUT can be configured to operate in parallel with the system under test. With this arrangement, it can be used to monitor activity on an operational system for troubleshooting or for acquiring measurements. This arrangement also allows testers to inject both simulated and manual traffic on a laboratory system to study feature interactions and load performance.

The ability to record activity, store the messages, and play them back at a later time provides an additional benefit for system testers. Since systems are usually developed in phases for field release, both old and new features must be tested in each phase. GAMUT is used

to record manual test cases for automatic retest of features at a later date, thereby guaranteeing continued system quality.

Thus, by taking advantage of the properties of message-based systems, GAMUT provides a foundation for multiple testing perspectives, provides testers with leverage in applying a wide variety of tests, and encourages a phased, systematic testing philosophy consistent with the architecture of the system under test.

In addition, because of the tool's structure, its application is not limited to testing systems designed with message-based architectures. By developing an access module that taps onto a collection of key system signals—including perhaps a clock signal—a message interface can be presented to the message utility system where there was no interface before. Test scripts can be devised and applied to the access module, which in turn maps the messages to sequences of signals at the appropriate connection.

IV. GAMUT ARCHITECTURE

A block diagram of the GAMUT test system is shown in Fig. 1. Within a system under test, a key message interface between subsystems A and B is selected and made available to a GAMUT access module. GAMUT will apply test stimuli and monitor system responses at this point to determine if the system is performing as expected. Typically, in System 75, subsystem A was the central control complex and subsystem B was the switching network and port complex. The message interface between A and B was the control channel.[7]

4.1 GAMUT access module

The access module interfaces the system under test to the message utility portion of GAMUT. Depending on the application, the module can be as simple as an Electronic Industries Association (EIA) channel. However, usually a special-purpose, microprocessor-based access module with interface-specific circuits is required. The access module provides five major functions:

1. Inserting stimuli messages into a subsystem under control of the message-utility system.
2. Capturing response messages and transmitting them to the message-utility software.
3. Message time stamping.
4. Peak traffic buffering.
5. Real-time message filtering so that only requested messages are recorded.

It is important that the module operate in real-time and log messages without affecting the performance of either subsystem, as well as to insert messages indistinguishable from subsystem-generated messages.

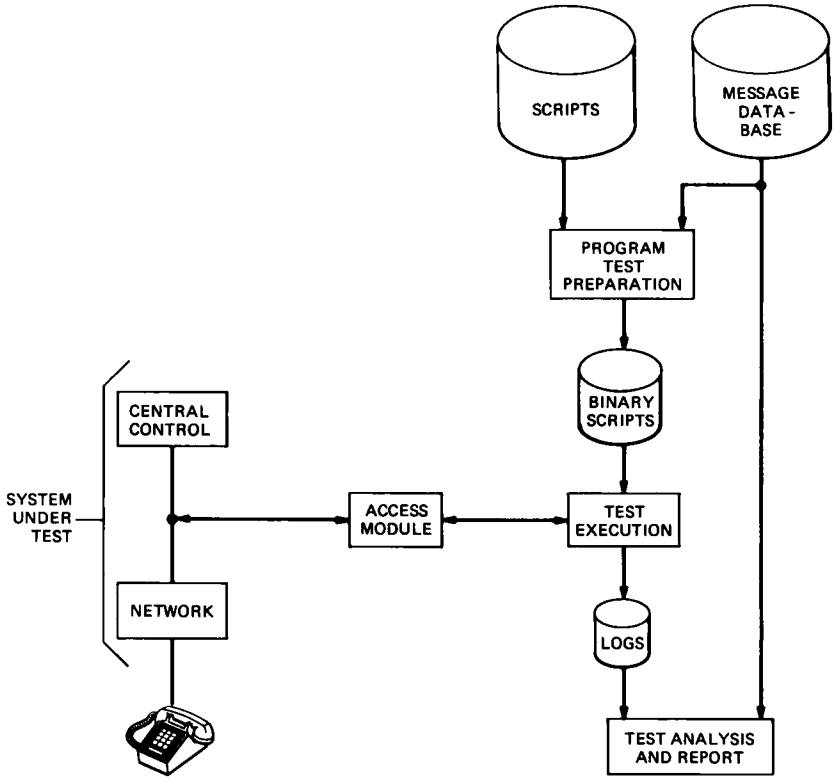

Fig. 1—GAMUT test system.

4.2 Common message-utility software

The common message-utility software runs under the *UNIX*™ operating system. It is driven by a database that contains a template for every message that flows between subsystems A and B. Isolating this format information in a database allows GAMUT to track message changes or additions within the interface and facilitates moving to another interface. This database allows users to communicate with the message utility software in a symbolic message language; it allows the software to convert this language into the format of the system under test; and it allows the software to report test results in the original format. Therefore, before any testing can begin, a description of every possible message that will flow through the message interface must be entered in the message database.

Test cases are entered into scripts that contain sequences of stimuli messages, expected system-response messages, and control information to specify how the messages are to be applied. Scripts are compiled by test-preparation software that converts the messages in the script

language into the format expected by the system under test. Control information is converted into a form easily interpreted by the test-execution software. The resultant file is called a binary script.

Binary scripts are then passed to the test-execution module, which can execute more than one binary script at a time to allow simulation of realistic test scenarios. The execution module reads the binary scripts and interprets script commands much like a computer CPU executes instructions; it cycles through multiple scripts in a round robin schedule much like a time sharing operating system. When the execution module encounters a stimuli message in a script, it instructs the access module to insert this message into the system under test. This module also monitors messages that return from the access module and verifies that these are valid responses as specified in the script or are exceptional conditions that were not specified. In either case, all stimuli and response messages are saved in log files as the test execution progresses.

When test execution is completed, the tester uses test analysis software to examine the logs. The analysis software consults the message database to convert the raw messages back into the form used by the test designer in the script language. Three types of reports are available. The first is a trace of all messages inserted during the test. The second is a trace of all the inserted messages and responses that provide a step-by-step record of every transaction occurring during the test. The third report is an exception report, which contains messages returned from the system under test that were not specified in a script as valid responses, as well as all messages that were specified in the script but were not received. The last report is interspersed with any diagnostics generated by the script execution. By reviewing these reports, the tester can determine how the test progressed, if the system performed correctly, and if not, what went wrong. A log-filter program is also available to help the tester focus on a particular type of message or portion of the log.

V. MESSAGE DEFINITION

GAMUT is customized to a particular application by defining a message database, called the ASCII Message Definition Table (AMDT). The AMDT is an ASCII file that defines the I/O interface language for the system under test; i.e., the file contains field descriptions of every message that passes through the message interface.

The AMDT contains both general message identification (e.g., op-code, message length, number of fields) and specific field-description information (e.g., field name, default values, and conversion operations). Data are entered in the file with any *UNIX* system text editor according to a prescribed format. Once a template is defined for each

possible message in the system under test, a tester can design tests that combine the messages with GAMUT script commands.

In the next few sections, we consider a script that tests a station-to-station call on System 75. In System 75, circuit packs communicate via messages with call-processing software over a time-division-multiplexed bus.[8] For example, when a user pushes a button on a telephone set, a message is sent to the call-processing software, which generates such responses as application of dial tone and activation of a call-status lamp. The messages are defined by the switch architecture; the off-hook message might be defined as follows:

```
offhook  station_id
```

A typical use of this message might be

```
offhook    1
```

which represents "go off-hook on station number 1." When converted according to the template in the message database, the resultant binary message is ready for insertion in the system under test. Conversely, a binary message received by the test-execution software is converted back to a "script language" message by comparing the messages in the database for field matches and selecting the best match.

VI. SCRIPT LANGUAGE

GAMUT allows a test designer to generate messages, apply them as stimuli, validate the resulting response messages, as well as control test execution. To achieve this, the script language must combine aspects of a traditional programming language, like C,[9] with a language focused on message manipulation. To simulate a particular feature operation, the tester combines a number of scripts into an experiment that GAMUT will execute.

In a script, stimuli messages are combined with the `send` command; the test execution software will insert each `send` message into the system under test. Expected response messages are combined with the `wait` command and for each `wait`, the test execution software will verify that the specified message actually occurred. There are several variations of the `wait` command that provide capabilities for error detection, for verification of a set of messages independent of their order of arrival, and for specification of several possible responses, only one of which will be matched. The GAMUT script language also includes several commands that provide flow control, assignment and arithmetic or logical operations, and subroutine capability similar to that of a programming language. Thus, a test designer can use the script language to specify a feature operation and validate that operation in a variety of scenarios.

6.1 Script execution

An arbitrary collection of scripts can be run together to simulate a field environment and create a realistic test. When these scripts are executed, the operations are verified independently; this can also occur while the system is manually generating traffic. Random characteristics can be added by using a pseudo-random number generator programmed into the script. Frequently one script will be assigned a control function, initializing and dispatching foreground and background tests, waiting for test case completion, and summarizing results. The tests can be executed sequentially or simultaneously. Another script can be assigned an administrative function, varying system translations[10] and corresponding GAMUT variables so that tests can be reapplied over various translation ranges and combinations.

6.2 Sample script

To describe features of the script language, we build below a sample script that checks the ability to place a call from one station to another within System 75. The station numbers and the call duration are established external to the script. The script defines the features of the script command language as necessary; so that the script can be reapplied to a variety of system states, field values—which may vary from one execution to another—will be parameterized.

All variables must be declared before use and can be either scalars or arrays. GAMUT uses the macro capability of C; therefore, `#include` statements can be used to access standard header files and `#define` statements provide simple or parameterized text replacement. Once the variables have been declared, most scripts require some start-up procedure—either to synchronize execution with other scripts in the experiment or to synchronize message flow with the application. Scripts are executed in a prespecified order according to a round robin schedule. For this example, we assume that a control script is executing and that it will initiate execution of the call script by setting the `Run_mode` variable to `RUN`, after defining the originating and terminating stations as well as the call holding time.

Thus the first lines of the script include variable declarations and run-time synchronization:

```
#include   "field_codes"
#include   "error_codes"
#define    RUN 1                 # Run-time synchronization
#define    CALL_COMPLETE 2       # Run-time synchronization
#
global     Run_mode              # Script execution state
global     Orig                  # Originating station id
```

```
global     Dest              # Destination station id
global     Extension[4]      # Digits of extension array
global     Wi                # Wait timeout interval
global     Holdtm            # Call duration
global     Good_calls        # Number of successful calls
global     Bad_calls         # Number of call failures
global     Busy_calls        # Number of busy calls
#

# Idle while Run_mode is not equal to RUN
#
:idle    if Run_mode != RUN    :idle
```

The values of the global variables are assumed to be set in the control script and, since they are global, will be known to all scripts in the experiment. The line with the `if` statement tests the value of `Run_mode`, and as long as that variable is not equal to the value `RUN`, control remains at the line labeled `:idle`. Once `Run_mode` equals `RUN`, control transfers to the next statement.

Next the station identified by the value of `Orig` will originate a call to the destination station. A message will be sent by the simulator to the call-processing software indicating that the `originator` station went into an off-hook state. The script expects, in response, a message for dial tone to be applied to the originating station. If the response does not occur within a specified time, control will be transferred to the statement `:no_dialtone`, where an error subroutine will be called. The subroutine will be included at the end of the script and will output an error message, increment an error count, and terminate the call.

```
  send                      offhook  Orig
  wait  Wi  :no_dialtone    tone     Orig  DIALTONE_ON
                            .
                            .
                            .


:no_dialtone
  call  :error (NO_DIALTONE)
  goto  :reset
```

On the `wait` command line, the variable following the `wait` is the amount of time (in seconds) the script will wait for a message from the application before logging a time-out error. As soon as the expected message arrives, the simulator processes the next instruction so the wait time should reflect the maximum expected delay between messages. If a time-out occurs, the simulator transfers control to the

statement with the label indicated. To make the script more general, we will assume the wait interval `Wi` is initialized in the control script.

The originator can now dial the destination extension; the digits to be dialed are stored in the array `Extension`. After the first digit is dialed, dial tone should be removed, and the script verifies call processing's action with a `wait` command.

```
  send                dial  Orig  Extension[0]
  wait  Wi  :no_quiet tone  Orig  DIALTONE_OFF
  send                dial  Orig  Extension[1]
  send                dial  Orig  Extension[2]
                      .
                      .
                      .

:no_quiet
  call  :error (NO_QUIET)
  goto  :reset
```

The three digits of the destination extension have now been dialed. If a dialing error occurs, control will be transferred to the statement labeled `:no_quiet`. Note that the same `tone` message is expected when origination occurs and after dialing the first digit; only the field value changes to control the tone.

Two different responses are now possible: the destination alerts or the originator hears busy tone. The `swait` allows the script writer to specify a set of alternative responses, only one of which will be correct. When that response is selected, control is transferred to the statement indicated. If no response is matched, control is transferred to the statement indicated on the `swait` line `:no_call`.

```
  swait  Wi  :no_call
  <
    :hangup  tone  Orig  BUSY_ON
    :alert   tone  Orig  RINGBACK_ON
:alert
  wait  Wi  :no_ringing  ringing  Dest  RINGER_ON
                         .
                         .
                         .

:no_call
  call  :error (NO_CALL)
  goto  :reset
:no_ringing
  call  :error (NO_RINGING)
  goto  :reset
```

A busy response will cause the script to transfer to statement `:hangup`; otherwise, the originator will hear ringback and the script will branch to `:alert` where the application of ringing at the destination is verified.

When the call is answered, several responses (ringback off, ringing off, connect) should occur and be verified; however, the order in which they occur is not important. The `mwait`, multiple message wait, allows the tester to specify all those responses. If at least one of the specified messages does not occur, control will be transferred to the statement specified on the `mwait` line.

```
:answer
  send              offhook      Dest
  mwait  Wi        :bad_ans
  <                 tone         Orig       RINGBACK_OFF
                    connect      Orig       Dest
                    ringing      Dest       RINGER_OFF

  >
  delay  Holdtm
                              .
                              .
                              .
:bad_ans
  call  :error (BAD_ANSWER)
  goto  :reset
```

When the destination answers, ringback and ringing are discontinued and the two stations are connected. The connection will be maintained for `Holdtm` seconds, at which time the call will be torn down.

```
  send                        onhook       Dest
  wait   Wi    :bad_onhk      discon       Dest
:hangup
  send                        onhook       Orig
  swait  Wi    :bad_onhk
  <
               :g_done        discon       Orig
               :b_done        tone         Orig  BUSY_OFF
  >
:g_done
  Good_calls = Good_calls + 1
  goto  :reset
:b_done
  Busy_calls = Busy_calls + 1
:reset
```

```
  Run_mode = CALL_COMPLETE
  goto  :idle
:bad_onhk
  call  :error (BAD_DISCONNECT)
  goto  :reset
endscript
```

The destination hangs up first and then the originator. The originator hang-up is labeled by `:hangup` so it can be referenced whenever a call attempt results in busy, as well as for normal call termination. At the completion of the call, the `Run_mode` is reset, and the script will idle again until reinitiated by the control script.

The error subroutine can be added to the end of the script file or included as a separate file.

```
# Error Subroutine
:error  sub (error_code)
  switch (error_code)
    {
    case DIALTONE:
      pstring "Call origination failure"
      break
    .
    .
    .
    case NO_RINGING:
      pstring "Destination alerting failure"
      break
    case BAD_ANSWER:
      pstring "Answer failure"
      send  onhook Dest
      break
    }
:terminate
    send  onhook Orig
    Bad_calls = Bad_calls + 1
  endsub
```

For each specified error code an appropriate error message is output and the originator hangs up. In the case of an answering failure, the destination station is also hung up.

VII. USING GAMUT WITH SYSTEM 75

GAMUT was developed in parallel with System 75 by the Product Test group of AT&T Information Systems. Extra effort was expended

so that operational milestones of the tool and the product were synchronized to allow early testing. This paid off by helping the test group to find protocol, feature, and performance affecting bugs very early in System 75's design cycle. In addition, developers used the tool to diagnose and isolate problems on early models of the switch.

One problem the Product Test group had with the tool was the oversensitivity of the test scripts to message ordering. The early version of the GAMUT script language had no `mwait` capability, and the test scripts explicitly verified message ordering by sequences of `wait` commands. Very often this ordering is not important (e.g., the ringer off, ringback off, and connect sequence from the sample script): as System 75's software evolved, these sequences changed and regression scripts would fail needlessly. Therefore, the product test group added the `mwait` feature to keep GAMUT in step with the product.

Early test scripts were developed by test engineers who had a detailed understanding of System 75's architecture, particularly the control-channel messages. However, as the feature list grew, the number of scripts required to test the system grew. Soon there were not enough skilled testers to develop the scripts. To solve this problem, we built a number of software tools into the utility that provided a record and playback capability that allowed less experienced testers to automate regression test cases in a timely fashion. A feature tester could execute a test that operates correctly on the system peripherals and record the control-channel messages. Next a script generator would read the logged messages and output a parameterized script. When executed, the script would reverify that test case automatically on subsequent releases of the system, as well as provide a basis for a class of related tests (range, load, etc.).

Sometimes, as features were finalized, there were sufficient design changes to invalidate previously recorded scripts. To avoid manually rerecording a test case, we developed a tool that extracted stimuli from the original script and automatically created a regeneration script. This script, with no verification capability, is executed and a revised regression script can be generated with the same software described above.

To test the fault detection and recovery capability of System 75,[11] we replaced the standard GAMUT access module with a hardware fault inserter. Then we wrote scripts to inject faults, execute system maintenance commands, and log test results, thereby extending the tool to System 75 maintenance.

GAMUT's System 75 access module is totally passive in the recording mode and is very portable. We were able to move some of the low-level message handling software from a minicomputer running the *UNIX* operating system to a portable PC, which resulted in a powerful

field tool. Our field-support team has found the portable version of GAMUT to be one of their main tools. It has helped them trace and isolate problems during our controlled introduction, as well as to measure traffic and customer use of features.

We found testing with GAMUT to be automatic, repeatable, and easily documented because test scripts, exception logs, and result logs are automatically saved in *UNIX* system files in the tester's language. Use of this tool facilitated problem isolation and encouraged phased, systematic testing, thus improving the testing process and system quality.

REFERENCES

1. G. J. Myers, *Software Reliability, Principles and Practices*, New York: Wiley, 1976.
2. H. A. Lanty, D. J. Morgan, and H. Oehring, "No. 1 ESS Furnishes ACD Service," Bell Lab. Rec. (March 1978), pp. 76–82.
3. R. J. Jakubek and S. M. Silverstein, "Microprocessor Control of Customer Premises Telecommunications Equipment," Proc. Ann. Conf. ACM (October 1976), pp. 270–4.
4. T. A. Dolotta et al., "The LEAP Load and Test Driver," Proc. Second. Int. Conf. on Software Eng., 13–15 October 1976, pp. 182–7.
5. D. A. Bennett and J. D. Walker, "System Testing the Small Communications System," unpublished work.
6. G. J. Meyers, *The Art of Software Testing*, New York: Wiley, 1979.
7. L. A. Baxter et al., "System 75: Communications and Control Architecture," AT&T Tech. J., this issue.
8. W. Densmore et al., "System 75: Switch Services Software," AT&T Tech. J., this issue.
9. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall, 1978.
10. H. K. Woodland, G. A. Reisner, and A. S. Melamed, "System 75: System Management," AT&T Tech. J., this issue.
11. K. S. Lu, J. D. Price, and T. L. Smith, "System 75: Maintenance Architecture," AT&T Tech. J., this issue.

AUTHORS

Carole J. Lake, B.A. (Mathematics), 1968, Gettysburg College; M.S. (Numerical Science), The Evening College of The Johns Hopkins University, 1970; National Security Agency, 1968–1974; Bell Laboratories, 1979–1982; AT&T Information Systems Laboratories, 1983—. Ms. Lake was initially involved in network operations support system development and for the past four years has worked in system testing for System 75. Member, ACM, IEEE.

James J. Shanley, B.E.E., 1961, Polytechnic Institute of New York; M.E.E., 1963, New York University; Bell Laboratories, 1955–1982; AT&T Information Systems Laboratories, 1983—. Mr. Shanley's development experience includes both hardware and software design on a variety of projects including T1 carrier, Subscriber Loop Multiplexer, *Horizon*® CMS, E911, *Dataphone*® II, and System 75. His current assignment is System 75 Field Support Group Supervisor. Member, Eta Kappa Nu, Tau Beta Pi.

Steven M. Silverstein, B.S. (Electrical Engineering), 1973, M.S. (Electrical Engineering/Computer Science), 1974, Polytechnic Institute of Brooklyn; Bell

Laboratories, 1973–1982; AT&T Information Systems Laboratories, 1983—. Mr. Silverstein has been engaged in business communication and management information system development. More recently he has been involved in system testing for System 75, and he currently supervises the Product Test and Evaluation group. Member, IEEE, ACM, Tau Beta Pi, Eta Kappa Nu.