AUTOMATED TESTING OF INTERACTIVE SYSTEMS

THESIS

Presented to the Graduate Council of the North Texas State University in Partial Fulfillment of the Requirements

For the Degree of

MASTER OF SCIENCE

By

Stephen C. Cartwright, B. S.

Denton, Texas

May, 1977

DwScott

Cartwright, Stephen C., Automated Testing of Interactive Systems, Master of Science (Computer Science), May, 1977, 86 pp., 2 illustrations, bibliography, 28 titles.

Computer systems which interact with human users to collect, update or provide information are growing more complex. Additionally, users are demanding more thorough testing of all computer systems. Because of the complexity and thoroughness required, automation of interactive systems testing is desirable, especially for functional testing.

Many currently available testing tools, like program proving, are impractical for testing large systems. The solution presented here is the development of an automated test system which simulates human users. This system incorporates a high-level programming language, ATLIS. ATLIS programs are compiled and interpretively executed. Programs are selected for execution by operator command, and failures are reported to the operator's console. An audit trail of all activity is provided. This solution provides improved efficiency and effectiveness over conventional testing methods.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER I

THE PROBLEM: RAPID DEVELOPMENT AND INCREASED COMPLEXITY OF INTERACTIVE SYSTEMS PRESENT NEW TESTING PROBLEMS FOR IMPLEMENTORS

The computer provides man with a tool which can be used to solve a variety of problems which would be impossible to solve or would require a great deal more time and effort to solve without the computer. On the one hand, computers provide man the ability to perform complex scientific calculations which have enabled space travel, and on the other hand, computers allow large volumes of information to be maintained, referenced, sorted, printed, etc. The first application would be impossible without computers, and the second would require enormous amounts of time and manpower.

The second type of computer application described above, saving time, effort and money, is the major objective of many computer system applications. A general area where this applies is the collection of information, in particular information provided by humans as opposed to hardware monitoring devices. If a computer is to use information, the faster the information can be placed into a form usable by the computer, the faster the information can be acted upon.

As an example, suppose information is received which must be read into a computer system. If this requires writing the information on a special form by one person, having the information punched onto a data card by a second individual, having the information read into the computer by a third individual and having the results returned by a fourth individual then the system is not efficient and is prone to errors. If the person receiving the information could sit at a keyboard and enter the information directly into the computer system, it would be a great deal faster, more efficient and less prone to errors.

An interactive system will therefore be defined as a computer system which interacts directly with a human user to collect information and provide results. Airline reservation systems are an example of interactive systems which are quite familiar to most people. The reservationist can determine what flights are available, the status of flights and make reservations for the customers by simply entering a series of commands on a keyboard and waiting for the responses. It is fast, efficient and reliable.

The use of computers for interactive systems has increased dramatically with the widespread use of minicomputers in the 1960's and early 1970's. The introduction of microprocessors and microcomputers in the mid-1970's broadened the use of computers even further. One applications area for mini/microcomputers has been in data collection. A

specific area of data collection involves entry of data by human operators through keyboards.

Prior to the late 1960's, the most widespread method of entering data was to have someone punch the information on cards, read the cards into a computer, perhaps save the information on disk or tape and throw the cards away. Cards served only as an intermediate storage medium which provided a method of transferring information from a form people could use to a form computers could use. In the late 1960's, the idea of transferring the information from the operator to tape was developed. This was followed by data being transferred to disk or flexible diskettes in the early 1970's. The information could then be read by or transmitted to a large mainframe. This eliminated the intermediate storage medium which could not be reused and replaced it with a medium that could be reused. The early systems were merely keypunch replacement devices. In other words, their capabilities were the same as those of a keypunch. Today the market and equipment are more sophisticated. The terms "source data entry" or "data entry" are now applied. The difference is in who uses the equipment and at what point in the information cycle it is used. Data entry systems allow the person generating the information to enter it into the system rather than rely on a specialist (keypunch operator) to enter the information. This means that the new systems must provide more flexibility and capability than previous systems.

New capabilities include the ability to detect errors as they are entered by requiring certain information to be alphabetic, numeric, of a fixed length, of a minimum or maximum length. The information can be automatically right-justified, left-justified, blank-filled or zero-filled.

The use of these systems is not restricted to just data entry. Systems now provide the capability to update information kept in a data base accessed by the minicomputer system. In addition, inquiries of a data base are also allowed.

The idea did not originate with minicomputer systems. Large computer systems also support interactive capabilities, and mainframe vendors as well as service bureaus provide this capability. However, the usage would not be as widespread as it now is because of the high cost per terminal. The minis allowed the processing to be moved into the environment of the data being entered. Collect the data at its source, eliminate as many errors as possible as soon as possible and require only the minimum time and effort to enter the information.

This is a principle of the distributed processing theory. Move the processing, or part of the processing, of information to the point in the cycle where it is most effective. Spread a little grease along the entire track rather than have a large dose right at the end.

The interactive systems, although a great benefit for the user, provide the systems developer a unique set of problems which are not found in the traditional batch mode environment. An interactive system relies on relatively slow human input rather than a high speed card reader, tape drive or disk drive. In addition, the more sophistication which is programmed into these systems the greater the combinations of input that can occur. For example, assume that a data field can be the following:

a. maximum 6 characters
b. valid characters are + - , . Blank 0-9
c. blanks may only appear to the left of the left-most non-blank character or to the right of the right-most non-blank character
d. The + or -, if present, must be the left-most non-blank character.
e. The "." if present, may be only followed by zero or two digits or blanks, and it may only appear once.
f. The "," if present, may appear only to the left of "." and there must be three digits between itself and the next "," or "." to the right.

Testing the valid and invalid combinations of this feature alone would be quite time-consuming.

The problem which arises with this type of system is how to approach the testing of the system so that a high degree of reliability can be guaranteed and at the same time

not expend great amounts of money and manpower in doing so. Not only should the product be reliable on its initial release but on subsequent releases. This means retesting the system as new versions are prepared.

The type of testing which must be performed is functional testing. Functional testing is the practice of testing a product against its specifications. Does the product meet its specifications? Does it do what it is supposed to do? The important thing is that the user interface function as designed and as documented. As new functions are added, new functional testing must be added; however, the existing features must continue to be tested to be sure they still work properly.

Testing of interactive systems can prove to be a time-consuming, expensive operation. Because of this, there is often a tendency on the part of the development organization to sacrifice testing in order to meet schedules or use the manpower on other activities.

Thus the problem is that of effectively testing the functional capabilities of interactive systems. How can this be accomplished? There are many ideas on testing techniques, testing aids and approaches to total systems design. These are discussed in Chapter II and Chapter III. However, it will be shown that these ideas do not apply to interactive systems, are too theoretical in nature or only approach part of the problem.

The real solution to the problem of effectively testing an interactive system is to automate the testing procedure so that the functional correctness of a system can be determined quickly and easily. Chapter IV presents an approach to automated testing which involves the development of an automated interactive test system incorporating a high level interpretive language for writing test programs. This is an approach which is common in the testing of hardware systems but has received little attention as a method of testing interactive systems. The advantages of such an approach are many. It provides greater flexibility, requires less time, tests can be repeated, testing can be modular and test procedures can be easily generated. These ideas are presented in detail in Chapter IV.

CHAPTER II

DESCRIPTIONS OF SEVERAL TYPES OF TESTING

Two terms are often used for the process of determining if a computer system works correctly: debugging and testing. Gruenberger states that until 1957 there was little to distinguish between debugging and testing (4, p. 11). Even today the words are often used interchangeably. The distinction to be made here is that testing is the process of determining if errors exist, and debugging involves isolating and correcting those errors. In the initial stages of systems development, there is generally a little testing and a great deal of debugging. As a system progresses, there is more and more testing and less debugging, implying that the further a project progresses, the fewer the number of problems that will be encountered. It is also generally true that this trend will tend to flatten out at the end, and that the problems discovered by testing will become more and more difficult to debug.

For the purpose of discussion, it is assumed that testing works in parallel with debugging, and when testing is referred to, it implies that debugging is also included. Finding problems is not an end in itself; they must be corrected.

Testing has always been a part of the development of computer programs and computer systems. However, in the past there was very little if anything in the way of test methodology, either in theory or practice. Vander Noot stated in 1971 that "Out of the millions of words written over the past few years about EDP, only one article could be found that was devoted to testing ..." (8, p. 60). In the past, a programmer would develop his code, execute it, go through some sequence of tests that satisfied him that the program worked and put it into use.

In recent years the attitude toward testing has changed. More and more emphasis is being placed on software reliability and testing methodologies. As software systems become larger and more complex, the need for better testing becomes essential. Huang states that 50 per cent of the man hours used in today's software industry are spent in program testing (5, p. 127). When the cost of software exceeds ten billion dollars annually (1, p. 48), any efforts to provide better efficiency in software testing could result in substantial dollar savings.

There is currently a great deal of information available about types of testing and testing tools. Are these ideas practical, and can they be applied to testing of interactive systems? Do they improve the quality of the software, and do they save time over manual testing techniques? This chapter examines the types of testing which can be performed. These

include module testing, system testing, alpha testing, beta testing, functional testing, regression testing and performance testing. The purpose and advantage of each is discussed, and in some cases these types of testing are virtually the same and others overlap.

Module, System, Alpha and Beta Testing

As testing of a software system moves from its initial stages toward completion, its characteristics change. Initially there are a large number of problems to be corrected, and at the end, all problems are believed to be solved. The concept of module, system, alpha and beta testing reflect this transition. These are a series of tests used to establish the correctness of a computer system.

Module test is the first stage. Components, or modules, of a software system are tested by the individual programmer to determine if they operate correctly. This may involve writing test drivers to test the modules. The next step, system testing, is sometimes called integration testing. It involves testing the combination of modules developed by different programmers or groups of programmers which together form a computer system. The implementors go through this phase of testing by running their own tests to determine if the system functions correctly.

Alpha testing is conducted when the system designers and implementors feel the product is ready to go into production. At this time formal tests are run against the

system. These tests can be developed by the developers or, more preferably, by an independent test group. The testing is designed to test all capabilities of the system in a formal, step by step procedure.

If the system passes its alpha test, it is considered ready to go into a customer or production environment. Beta testing is the first production system and represents the system's introduction into the real world. The product is used, rather than tested, but it is observed closely in order to evaluate its performance.

Module and system testing have always existed and are what was considered debugging in the past. Alpha testing represents a formal testing step to determine a system's correctness. Alpha testing and beta testing are also given the terms verifications and validation (6, p. 9). Verification means the logical correctness of a system based on execution in a test environment, and validation means the logical correctness in an external (or real) environment.

All these stages apply to the testing of an interactive system. Automation is not really necessary in the first stage (module testing) and in the early part of the second stage (system testing) since the system is not yet in a stable state. However, whatever automated test tools are available can be useful in the later stages of system test and alpha test. This is where exhaustive testing is necessary, and automation can save valuable time since this

is where projects often begin to run late or else it is noticed that they are late. Beta testing, since it is production testing, is not appropriate for automated testing.

Thus automated testing can be a valuable tool during the later part of systems test and during alpha test for interactive systems.

Functional Testing

This term is often used interchangeably with alpha testing, however, there are some differences. Functional testing means testing the system to determine if it does what the design specifications say it is to do. Alpha testing should include functional testing and may in effect be functional testing. Alpha tests may include other tests which are not functional tests. These may include tests designed to determine if the method of implementation is correct rather than if the end result is correct.

Functional tests are based on the product specifications and do not take into consideration the method or techniques of the implementation. It is not concerned as much with how it works as with whether it does what is is supposed to do. Preparation of functional tests involve the following (2, p. 359):

- Identify test conditions, including unique cases.
- Develop test cases, i.e. each test case is a single condition.

- Prepare test script which state how the test is to be performed. A test may be one or more cases.
- Prepare test data base if necessary.
- Prepare test driver(s) if necessary.

The idea of functional testing is receiving more attention as the discipline of software engineering develops. More emphasis is being placed on determing if a product does what the specifications say it is supposed to do rather than does it work.

Numerous papers presented at the Second International Conference on Software Engineering deal with the problem of systems which do not do what the customer expects them to do (7). Between the time the specifications are written and the implementation is completed, the product may get off target and as a result, it may not perform the functions the specifications say it is to perform. Although functional testing cannot solve this problem in and of itself, it can be a great aid in determining if this has happened especially if the tests accurately reflect the specifications. This assumes that the specifications are not ambiguous and are complete and that the test cases adequately cover the specifications. If this is true, the implementors have a valuable tool for determining if their implementation is correct, i.e. meets the specifications.

The ideas of functional testing apply to interactive systems development. Like any other system, if they do not

perform their intended function, they are not successful. In addition, automation of the functional testing is very desirable. Functional testing is meant to be complete and exhaustive. As a result, it may be extremely time-consuming to execute the functional tests unless steps are taken to automate them. Therefore the automation of functional testing for interactive systems is a desirable objective, and these tests can be used during the later stages of system testing and during alpha testing.

Regression Testing

Suppose a computer system supports only printed output. Changes are made to the program so that it also supports punched card output. The implementor of the punch code tests his code very carefully and is confident it works correctly. All testing performed against the punch code is successful. Then tests are run against the print capability and problems are found. When the programmer is asked if he tested the print code, he responds "Test it ... why should I test it ... I didn't make any changes to it?" Perhaps not, but something he did adversely affected the existing code.

Regression testing is testing which insures that what used to work still works. It means making sure that new features have no adverse affect on existing code and the current test results match previous test results (2, p. 355).

Customarily, regression testing is simply the alpha tests or functional tests re-executed for a new version or release. Ideally, a test procedure allows new tests to be added for new features and the entire test procedure is rerun for each release or version of a system. Thus, regression testing is the insurance that a product maintains its correctness from version to version.

This type of testing applies to all types of systems including interactive systems. The advantages of automation which apply to functional testing and alpha testing are even more important for regression testing. As stated before, regression testing insures that a system maintains its correctness. After the first few releases of a system, quite often the original implementors and designers are no longer associated with the system. As a result, those testing subsequent releases of a system are not as familiar with it. For this reason automated testing is beneficial because investigation into the system is only necessary when a problem occurs. Those tests which execute correctly do so without requiring detailed knowledge on the part of the tester.

Performance Testing

Performance testing answers the types of questions: how many, how fast and how long. For example, how many communications lines will a system support, how fast will it support

them (at what point will it degrade) and how long will it support them before failure? Specifications often state how many of this device or that device a system will support. How many concurrent jobs, how many concurrent input/output activities or how many data entry terminals will the system support? Specifications may also state how fast a job is to be executed, how fast a device is to be responded to or how fast information is to be transferred to a device. Specifications may also state how long a system must run between problems (mean time between failure). Performance testing can be used to determine what the performance limits of a system are. Does a system meet its performance specifications? Does it exceed the specifications and, if so, by how much?

How does performance testing differ from functional testing? Elmendorf states that performance testing involves space and time measurements of system utilization, and functional testing involves the measurement of system quality (3, p. 137). This separation may be too strong. The specifications for a system may state functional and performance objectives, but it may be difficult to distinguish one from the other. If a system is to support concurrent input and output, that is a functional capability. If the system is to sustain a certain rate of input and output, that is a performance capability. They are closely related.

Part of alpha testing may involve performance testing, just as it involves functional testing. Performance testing may also be included in regression testing. Depending on the design specifications, performance testing may be required for an interactive system. If so, the advantages of automation are the same for performance testing as for functional testing.

Summary of Types of Testing

Functional testing relates to the quality of a system, and performance testing involves the space and time measurements of system utilization. Both are used in determining or validating that a system meets the design specifications. These two types of testing are formalized into test procedures and executed during the later stages of system testing, during alpha testing and during regression testing. Alpha testing is generally associated with the development of new systems or new features, while regression testing is concerned with the continued correctness of existing features.

All of the above apply to the development and implementation of interactive systems. Automation of these testing techniques is a desirable objective because of the greater efficiency and thoroughness which can be realized during the testing process. The next two chapters seek to find a method of automating the functional testing of an interactive system during system testing, alpha testing and regression

testing. Performance testing is not given direct consideration.

CHAPTER BIBLIOGRAPHY

1. Boehm, Barry W., "Software and Its Impact: A Quantitative Assessment," DATAMATION, May, 1973, 48-59.

2. Duke, M. O., "Testing In a Complex Systems Environment," IBM Systems Journal, Volume 14, Number 4, 1975, 353-365.

3. Elmendorf, W. R., "Disciplined Software Testing," Debugging Techniques In Large Systems, edited by Randall Rustin, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1971, 137-140.

4. Gruenberger, F., "Program Testing: The Historical Perspective," Program Test Methods, edited by William C. Hetzel, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1973, 11-14.

5. Huang, J. C., "An Approach to Program Testing," ACM Computing Surveys, Volume 7, Number 3, September, 1973, 113-128.

6. Montgomery, George Wynn, System Test Methodology, NTIS AD/A-012 461, June, 1975.

7. Proceedings Second International Conference on Software Engineering, IEEE Catalog Number 76CH1125-4 C, October, 1976.

8. Vander Noot, T. J., "System Testing....A Taboo Subject," DATAMATION, November 15, 1971, 60-64.

CHAPTER III

DESCRIPTIONS OF SEVERAL TEST TOOLS AND TECHNIQUES CURRENTLY AVAILABLE OR CURRENTLY BEING DEVELOPED

As computer systems have become more complex and as the importance of testing has been given more emphasis, numerous techniques and approaches have been developed and are being developed. Some are techniques and methodologies while others are specific tools. This chapter evaluates the advantages, disadvantages and applicability to interactive system testing of these techniques and tools. Each one is evaluated to determine if it can aid in the automation of interactive system testing.

Synthesis Techniques

Miller uses the term "synthesis techniques" to summarize the various ideas for improving program development so that bugs cannot get into the system initially (11, p. 2). Synthesis techniques include the generally accepted ideas of structured programming which require the programmer to organize his program into logical blocks which are executed through the use of control structures. The ability to branch or execute a "GOTO" type instruction is not provided for his

use. The ideas of top down design (12) and the chief programmer team concept (1) are examples of other synthesis techniques.

Although this is not truly a testing technique, the ideas of synthesis techniques and testing are so closely related that they are discussed here. The argument is that no matter how comprehensive the testing procedure, code which is poorly designed, poorly structured or poorly coordinated may never complete (pass) testing. Elmendorf states the relationship between design, development and testing very well (2, p. 138):

> If either the design or implementation is sloppy, then the testing bogs down in a morass of problems, difficult to identify, expensive to fix, and impossible to schedule.
>
> If the testing is sloppy, then there is little motivation to practice quality in the design and implementation phases. Managers favor the criteria of excellence against which they're being measured. Testing is quality measurement and thus, indirectly, quality motivation.

Synthesis techniques prevent errors from getting into a system. These ideas are applicable to all types of software systems including interactive systems. Although their use is important, they can not aid directly in the automation of interactive systems testing.

Symbolic Execution and Program Proving

Program proving is a fascinating idea and one that appeals to any programmer. The security of knowing that a program or system has been proven to be correct would be very

reassuring. King has written several articles (5, 9) which deal with symbolic execution as a method for proving correctness. This involves breaking down a program into a symbolic execution tree which represents all the branches within a program and symbolically executing each branch to prove its correctness. However, most programs do not have a finite number of branches so King proposes the trees be traversed inductively rather than explicitly. Other papers by Elpas (3) and Good (4) also approach program testing through formal proofs.

Although this technique is very desirable, how practical is it for interactive systems at this time? Huang (6, p. 113), Miller (10, p. 51), and Ramamoorthy (14, p. 383) all state that program proving is currently only useful for small programs and is still many years away from being useful for large software systems. Ramamoorthy states several reasons why this is so, including: 1) it is too costly, 2) it is not useful for systems with parallel processing, and 3) most programmers do not have adequate mathematical background to use it (14, p. 384). Program proving, therefore, does not appear to be a technique that can aid in the testing of interactive systems at the current time.

Simulation

Simulation has application to testing as well as to other areas. Simulation is useful in modeling a real world

situation. Simulation can be used to model a proposed system in order to evaluate it. Simulation can be used to simulate part of a system, and this is where it is useful in interactive system testing. To test an interactive system, simulation of the human interface to the system can be used. Therefore, in discussing the automation of testing, simulation comes into play, i.e., the simulation of the human interface to the system.

The entire interactive system could be simulated and there are benefits in doing so, but these benefits are in the area of debuggings as discussed by Supnik (17). In the final analysis, however, the real system itself must be tested and not a model of it. Thus simulating the entire system does not aid in the final testing. However, if the human interface to testing the system can be simulated then simulation is valuable.

In this way, simulation will be part of the solution of automating the testing of interactive systems. In this case it is an approach and not the tool itself. The tool to implement the simulation must be found.

Automated Test Case Data Generation

Given a program which accepts input, how does one go about selecting the input to be used to comprehensively test the program? For example, assume that a data field can be the following:

a. Maximum 6 characters.

b. Valid characters are + - , . blank 0-9.

c. Blanks may only appear to the left of the left-most non-blank character or to the right of the right-most non-blank character.

d. The + or -, if present, must be the left-most non-blank character.

e. The "." if present, may be only followed by zero or two digits or blanks. There may be only one ".".

f. The "," if present, may appear only to the left of "." and there must be three digits between itself and the next "," or "." to the right.

The possible valid and invalid combinations are too numerous to test. How then does one select a valid set of test data? In 1971, Vander Noot suggested criteria which would be taken for granted today such as testing upper and lower limit values, placing alphabetics in numeric fields, division by zero, etc. (18).

Today there are more sophisticated approaches to test case data generation. These approaches include the analysis of program paths and selection of data to test these paths. This analysis can be performed by a program designed to analyze other programs and select test data. Miller (10) and Ramamoorthy (13) both present approaches to accomplishing this.

As with program proving, it would be desirable to be able to let a program specify the types and amount of test case data to be used and know that all the unique paths would be executed using that data. However, as with program proving, the state of the art is not to that point yet. There is little of practical value as far as automated generation of test cases for systems programming applications is concerned. The work which has been done deals with high level languages such as FORTRAN.

A deficiency with this type of test case data generation is that the test case data is based on the structure of the program and not the function of the program. This is all right as long as the data generated actually tests all the functional capabilities, but there is no guarantee that it will do so. Determining that a program executes correctly is only part of the job. The program must also perform the function it was designed for.

Automated Verification Systems

In his article, "Testing Large Software with Automated Software Evaluation System," (14) C. V. Ramamoorthy describes tools which are currently available including automated verification techniques. There are two approaches to verification: static and dynamic. Static systems analyze code and detect logic errors, poor construction, etc. They point out problem areas which should be corrected. Dynamic systems

involve the insertion of probes into the system which are then monitored to determine if the correct paths are taken and that correct values occur at a given point. A great deal of this can be done interactively so that a program can be "watched" as it executes. This is testing interactively rather than testing automatically.

Again these tools are complex. They do provide analysis of code and can point out logic problems, but it is questionable if they can be executed effectively by a novice programmer or someone not familiar with the code. They are perhaps best suited as a design or debug technique and not as a formal functional test.

In addition, as described by Miller (11), these systems are based on looking at the actual code rather than the specifications. There is a danger in this approach that correct programs will be produced which do not meet the design specifications.

Automated Testing Using a Test Language

Testing of hardware systems through the use of test equipment which is programmable is an idea which seems reasonable and is done quite often. Can the same idea be applied to a software system? If so, should there be a language developed to accomplish this task?

The "Roster of Programming Languages for 1974-75" published in the December, 1976, edition of Communications of

the ACM (16) includes a list of 167 known programming languages. To be listed, a language has to (a) have been developed or reported in the United States, (b) have been implemented on at least one general purpose computer, (c) be in use by someone other than the developer (16, p. 655). Five languages are listed as dealing with equipment checkout: ATLAS, DETOL, DMAD, RATEL and SVDSS. Two are listed as dealing with real-time process control: COMTRAN and RTPL. SVDSS stands for Space Vehicle Data System Synthesize, and its description as "a very simple language for modeling discrete or analog systems" (16, p. 668) implies it is highly specialized and has little application to automated testing. It is not discussed here. RTPL stands for Real Time Procedural Language and is described as a "language to provide real-time control of software models written in SVDSS.. " (16, p. 667). In other words it is used with SVDSS and therefore has the same limited scope.

Of those remaining, no information could be obtained concerning DMAD and COMTRAN. DMAD is described as "An engineering user oriented test language for functional testing of digital devices. Allows device description in terms of registers and signal names and functional operators, such as logical and Boolean operations" (16, p. 660). COMTRAN is described as "A language containing statements to permit easy programming of on-line, real-time communications hardware tests. Heavily column oriented with assembly language style" (16, p. 659).

This leaves ATLAS, DETOL and RATEL. It is not anticipated that these languages can be used for interactive system testing, but there is the possibility they contain ideas or techniques which can be useful.

ATLAS.--ATLAS (Abbreviated Test Language for All Systems) was developed originally by Aeronautical Radio, Inc. It is a standardized test language for expressing test specifications and procedures independent of the test equipment it is to be used with. It was initially developed under the direction of the Airlines Electronic Engineering Committee in 1968 but since then has found uses with airlines, industry and government. As its acceptance expanded, the Department of Defense approved it as an interim standard language for automatic test equipment.

In September of 1976, authorization for the distribution and maintenance of ATLAS was turned over to the Institute of Electrical and Electronics Engineers (IEEE). This resulted in the publication of IEEE/ARINC Standard ATLAS Test Language (7) which describes the language.

ATLAS is a high level, English-like language with many structured programming characteristics such as IF-THEN-ELSE, WHILE-THEN, FOR-THEN and BEGIN-END statements. However, it also allows an unconditional and conditional GOTO statement although the manual advises against their use. It has input/output, data and arithmetic operations comparable to PL/I. In addition it has 150 - 200 specialized instruction such

as DOPPLER, STEP SIGNAL, WAVEFORM and others which are hardware test oriented.

Aside from the structured statements, there is very little about ATLAS which is directly applicable to the development of a test language for interactive systems.

DETOL.--DETOL (Directly Executable Test Oriented Language) is a much more basic, lower level language than ATLAS although both are designed to function in an automated test equipment environment. "Improved DETOL Programming" describes the language (8). It does not offer the structured control instructions that ATLAS provides and allows only a "jump" instruction for transfer of control. It is FORTRAN-like in its arithmetic capabilities and has bit manipulation capabilities. It has a number of hardware oriented instructions such as the set, apply and release stimuli instructions but not nearly as many as ATLAS.

There is little if anything in DETOL that can be applied to interactive system testing.

RATEL.--RATEL stands for Raytheon Automatic Test Equipment Language which was developed by the Raytheon Corporation for use in testing component systems of the PATRIOT missile system. It is a FORTRAN-like test language which includes nearly all capabilities of Raytheon 704 FORTRAN IV and SYM II assembly language (15).

RATEL is an interpretive language. This means that the RATEL compiler does not generate executable code but instead generates interpretable code. There is a run time system interpreter which executes the code generated by the compiler.

RATEL allows program segmentation with up to fifteen segments. The segments need not be executed in a certain order. Global and local variables are allowed. Global variables may be accessed by any segment and must reside in the first segment. Local variables can be referenced only by the segment in which they appear.

Like ATLAS and DETOL, RATEL contains many instructions which are hardware oriented. Transfer of control statements and control structure statements were not obvious from reading the available documentation (15).

The characteristics of RATEL which could be applied to an interactive system include the idea of an interpretive language, segmentation and the global/local variable capability. The run time system also allows the operator monitoring the system to cancel tests (programs) and select others. This, too, is viewed as desirable.

Summary of Current Test Tools and Techniques

Of the currently available test tools and techniques, few of them can be applied directly to the automated testing of interactive systems. Program proving and automatic test case data generation, although desirable, have not been

developed to the stage where they are applicable to testing of large systems. Automated verification systems have the limitation that they determine if the program works but do not determine if the program meets its specifications.

Synthesis techniques can be of indirect help in testing because they aid in preventing errors from getting into the system. Currently available test languages cannot be applied directly to interactive systems, however there are several characteristics of RATEL which are applicable. Simulation offers an approach to interactive system testing but is not a solution itself.

The next chapter presents the idea of simulating the user interface to an interactive system by means of an automated interactive test system. This system includes a specialized programming language.

CHAPTER BIBLIOGRAPHY

1. Baker, F. T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Volume 11, Number 1, 1972.

2. Elmendorf, W. R., "Disciplined Software Testing," Debugging Techniques In Large Systems, edited by Randall Rustin, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1971, 137-140.

3. Elspas, Bernard, Karl N. Levitt, Richard J. Waldinger, and Abraham Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys, Volume 4, Number 2, June, 1972, 97-147.

4. Good, Donald I., Ralph L. London, and W. W. Bledsoe, "An Interactive Program Verification System," Proceedings International Conference on Reliable Software, June 1975, 482-492.

5. Hantler, Sidney L., James C. King, "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys, Volume 8, Number 3, September, 1976.

6. Huang, J. C.,"An Approach to Program Testing," ACM Computing Surveys, Volume 7, Number 3, September, 1973, 113-128.

7. IEEE/ARINC Standard ATLAS Test Language, The Institute of Electrical and Electronic Engineers, Inc., IEEE std 416-1976, New York, New York, 1976.

8. Improved DETOL Programming for the 5500/5510 Automatic Test System (ATS), Report Number ER-8964, AAI Corporation, Cockeysville, Maryland, March, 1977.

9. King, James C., "A New Approach to Program Testing," Proceedings International Conference on Reliable Software, June, 1975, 228-233.

10. Miller, E. F. and R. A. Melton, "Automated Generation of Testcase Datasets," Proceedings International Conference on Reliable Software, June, 1975, 51-58.

11. _____, Methodology for Comprehensive Software Testing, NTIS AD/A013 111, June, 1975.

12. Mills, Harlen, "Top Down Programming in Large Systems," Debugging Techniques In Large Systems, edited by Randall Rustin, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1971, 41-56.

13. Ramamoorthy, C. V. and S. F. Ho, "On the Automated Generation of Program Test Data," Proceedings Second International Conference on Software Engineering, October, 1976, 636.

14. _____, "Testing Large Software With Automated Software Evaluation System," Proceedings International Conference on Reliable Software, June, 1975, 382-394.

15. Ring, Steven J., "A Distributed Intelligence Automatic Test System for PATRIOT Electronic Assemblies," for publication in IEEE Transactions on Aerospace and Electronic Systems, 1977.

16. Sammet, Jean E., "Roster of Programming Languages for 1974-75," Communications of the ACM, Volume 19, Number 12, December, 1976, 655-669.

17. Supnik, Robert M., "Debugging Under Simulation," Debugging Techniques In Large Systems, edited by Randall Rustin, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1971, 117-136.

18. Vander Noot, T. J., "System Testing.... A Taboo Subject," DATAMATION, November 15, 1971, 60-64.

CHAPTER IV

A SOLUTION: AN AUTOMATED TESTING CAPABILITY FOR INTERACTIVE SYSTEMS

As discussed in Chapter III, the current state of system programming technology does not offer much in the way of sophisticated tools for testing software. Several new ideas are in the developmental stages such as program proving and automated test case data generation, however these are not currently practical. What is needed is some method of automating the testing, in particular the functional testing, of interactive systems.

The practical fact of the matter is that functional testing determines if the product meets the design specifications. This is the ultimate purpose of any system. In addition, if the test procedure is repeated for each release of the system then regression can be prevented. The net result is a reliable system which meets the specifications and will continue to do so as the product is improved.

Automatic or automated testing is an idea which is relatively common in testing of hardware systems, as discussed in the previous chapter. Although the application of these techniques to software testing seems quite reasonable, the current literature would indicate that it is not widely used.

Although Scherr indicates something of this type may have been done for the OS/360 time sharing system, few details are given (2). Even though test drivers and stimulators are common practices, the development of a programming language and system specifically for testing interactive systems appears to be an area which has not been adequately developed.

Automating the functional test procedures for an interactive system is not a cure all for interactive system development. The expression "the chain is only as strong as its weakest link" applies to system development. Functional testing is only a link in the chain made up of system analysis, system design, programming practices and many others. If the specifications are ambiguous or the programming technqiues are sloppy, testing may prove of little value. Thus functional testing is only as good as:

- the clarity, completeness and detailedness of the specifications
- the adherance to sound programming practices during development.

If these qualities are present then automated functional testing can be meaningful.

If an automated test method is to be developed for an interactive system, what features are desirable? The following features are considered important:

1. It should require fewer man hours to perform the tests than manual testing.

Manual testing of interactive systems can require a great deal of time. The number of possible combinations and sequences of events for an interactive system can be so large that manual testing may require many man days. If this time can be reduced significantly by an automated test system then substantial dollar savings can be realized as well as freeing manpower to perform other functions.

Another advantage is that the shorter the period required to perform the tests, the more likely they are to be run on a regular basis. It can often be observed that subsequent releases of software systems receive less and less testing. One major reason is that the time required to perform the testing is not allocated. The net result can be regression. The shorter the time required for testing, the more likely it is to be done, the less likely regression will occur and the more reliable the final product.

2. It should be flexible.

Very few software systems are static. No sooner is the initial release complete than work begins on modifications and new features. Any method of automated testing must be easily modified and must allow additions of new tests easily. If this is not true, those features in the initial release

are the only ones which are ever tested,because adding new tests for new features takes too long or is too difficult. In addition, manpower must be devoted to updating the tests which could be used in more productive areas. Thus flexibility helps to insure that new features will be tested and that only minimum manpower will be required to maintain the tests.

3. The tests and test sequences should be repeatable.

In general, software problems which can be recreated by following a certain sequence of steps are more easily solved than those which cannot. How often is the programmer confronted with a memory dump of a failure and no one knows the sequence of events which led to the failure? These problems often prove difficult to solve. However, if the failure occurred as a result of running a specific sequence of tests which can be easily reproduced, not only are the chances of solving the problem improved but the time required to do so is likely to be shortened. This means that if a random sequence of events or inputs are to be used, this same random sequence must be reproducible in case a failure occurs.

4. The tests should be modular and separately executable.

If a problem is discovered with part of the system as a result of performing the automated test procedure, it is very

desirable to rerun only the test that failed rather than the entire test sequence in order to recreate the problem or to determine that a successful correction has been implemented. The ability to execute a particular test or a particular subset of the tests is a feature which can save considerable time when isolating, correcting and retesting a problem.

5. The tests should be easy to execute.

If the knowledge and experience required to execute the test procedure is considerable then valuable manpower resources must be devoted to this task. This too can be a contributing factor to less thorough testing of subsequent releases because the personnel capable of running the tests are on a critical project, have moved to a different department or have quit. If, on the other hand, the tests can be performed by less experienced personnel, there are more options available as to who can execute them, it will require less manpower to execute them and as a result they are more likely to be done. In addition, the testing can serve as a training tool for new personnel.

6. The tests themselves should be easy to generate.

The greater the time or effort required to develop the tests, the greater the chances that short cuts may be taken, that tests will not be comprehensive or that new tests will not be added. However, if the system allows tests to be

generated easily, the above problems can be avoided. For example, if a thousand random input values of certain characteristics can be generated using only a few directives to a test language then addition of a new test is more likely than if all thousand values had to be specified individually.

If an automated test system incorporates the above ideas, there is a greater likelihood that it will be used, that it will be maintained, that it will be functionally effective and that it will be cost effective.

This chapter states the specifications for an automated testing capability for an interactive system. This specification has been separated into three components: a programming language syntax specification, a specification for a compiler and linkage editor for the language and a specification for the run time system which will execute the code generated by the compiler.

The major component of the automated test system is a programming language specifically designed for that purpose. The language is called ATLIS, standing for Automated Test Language for Interactive Systems. It can be characterized as a high level, block structured language with instructions tailored for interactive system testing. A compiler is required for compilation of the ATLIS language statements. The compiler will accept ATLIS source language statements as input and produce interpretable code as output. The reasons for producing interpretable code are:

- provide machine independence for ATLIS compiler.
- allow multiple, concurrent execution of more than one device using the same or different programs.
- allow paging of code so that large programs can be executed in a small memory space.

In order to allow parts of a program to be compiled separately, the program can be divided into procedures. Each procedure can be compiled separately to produce object code output. A linkage editor is then required to combine the separate object modules, resolve external references and produce an interpretable load module.

The execution time portion of the system interprets the code generated by the ATLIS compiler and performs the appropriate action. This portion of the system has the following features:

- provides multiple, concurrent testing of several interactive devices.
- if deisred, provides an audit trail of functions performed.
- provides indication of errors and allows the test supervisor to select course of action.
- allows a test, a group of tests or all tests to be executed through operator specifications.

Figure 1 gives an overview of the entire system.

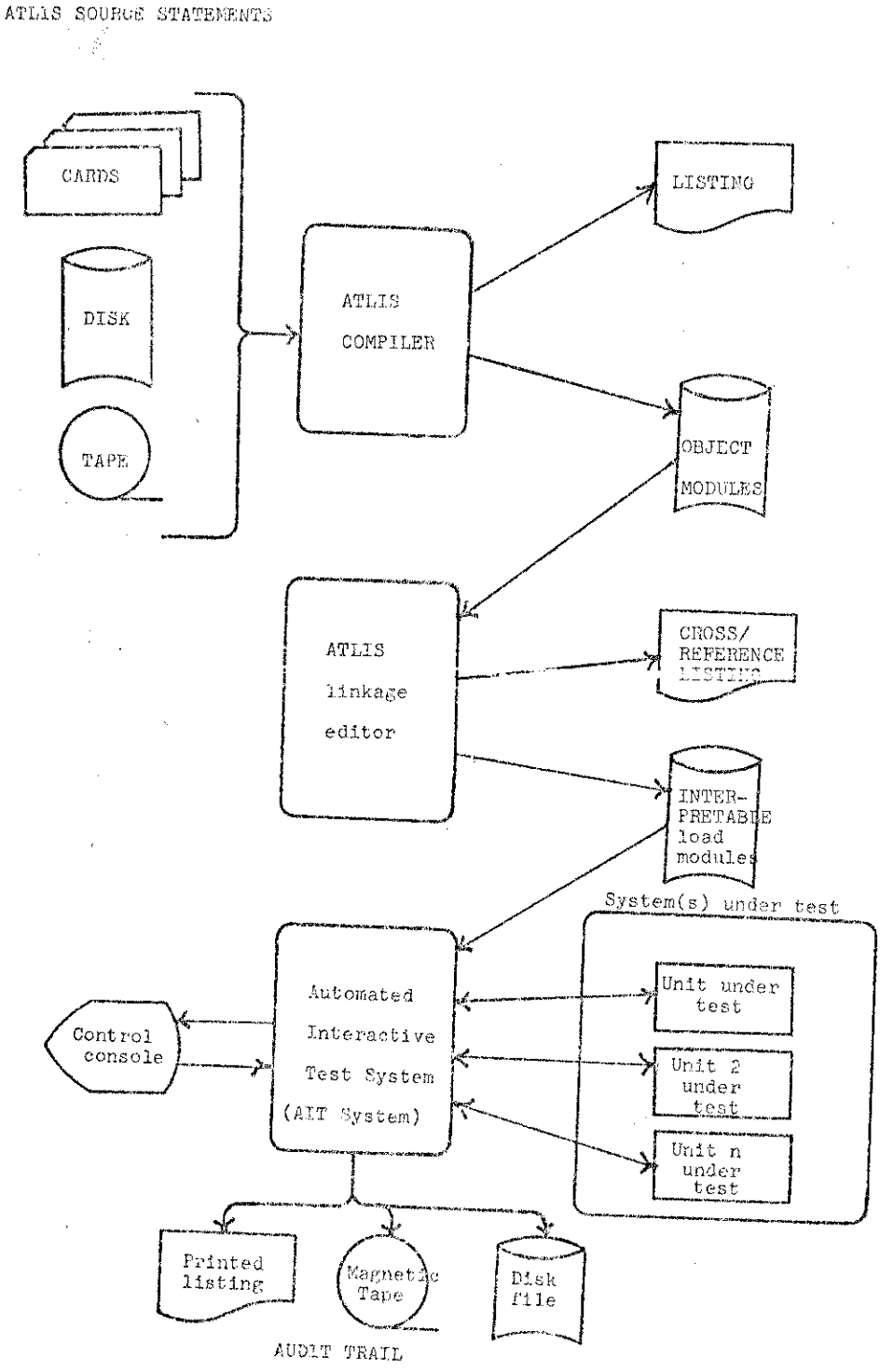

Fig. 1--An overview of the automated interactive test system.

The Programming Language ATLIS

The programming language ATLIS is designed for use as a test language for interactive systems. However, the concepts and characteristics reflect current ideas in the field of program language development. ATLIS is a high level language meaning it is machine (computer) independent as far as its implementation is concerned. It is a structured language, and program execution is based on the successive execution of blocks of code or alternate blocks of code. A block of code can be a single statement or a group of statements contained within a DO, BEGIN or CASE block. ATLIS does not contain a GOTO statement as part of its syntax.

Floating point arithmetic is not provided for in ATLIS. Only character and integer variables are allowed. All variables must be explicitly defined prior to the first executable statement.

Procedures

An ATLIS program is made up of one or more procedures. Each program must have one and only one main procedure and can have multiple procedures which are not main procedures. The main procedure receives control when a program starts execution. It can issue the CALL statement to transfer control to other procedures. Any procedure can call any other procedure except the main procedure, and the RETURN statement returns control from the called procedure to the

to the calling procedure or the operating system in the case of the main procedure.

Variables

Variables defined within the main procedure can be referenced by any procedure. Variables defined in a procedure other than the main procedure can be referenced only by that procedure. Two types of variables are supported by ATLIS, character and integer variables. One-dimensional arrays are supported for both numeric or character variables.

Identifiers

An identifier is a string of characters used to identify a variable or control structures such as a BEGIN/END block. Identifiers may be from one to sixteen characters, the first of which must be alphabetic. The remaining characters may be alphabetic, numeric or the underscore character. Identifiers must begin in column one and must be separated from the rest of a statement by at least one blank. Embedded blanks are not permitted. For Example:

```
ALPHA_SET   DCL  26A,'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
SET_STATUS BEGIN
                .
                .
                .
                .
           END  SET_STATUS
```

Constants

Constants may be numeric or character constants. Numeric constants may be positive or negative. Character constants must begin and end with a quote. Whenever a quote is to be used in a literal, it should be replaced with two quotes.

Example: -9972 +143

'ENTER NAME'

'THIS IS A QUOTE '' MARK'

Spaces, Comments and Continuations

Spaces may not be embedded in identifiers (including subscripts), operators or numeric constants. A blank must follow all identifiers, precede and follow all operators.

Comments must have an "*" in column one and must be entered as a separate card (i.e., input record).

A statement may be continued on the next input record by placing an "X" in column 71 of the record to be continued. Comments may not be continued.

Arithmetic, Relational and Concatenation Operators

The following are the allowed arithmetic operators:

addition	+
subtraction	-
division	/
multiplication	*

The following are the relational operators which are supported:

equal	.EQ.
not equal	.NE.
less than	.LT.
less than or equal	.LE.
greater than	.GT.
greater than or equal	.GE.

Two character strings may be concatenated to form one string using the concatenation operation:

For example:

```
SET NEW_STRING = STRING_ONE || STRING_TWO
```

ATLIS Instructions

The following are the ATLIS instructions in alphabetic order. Key words are shown as all capital letters. If the key word is optional, it and its associated parameters are enclosed in brackets []. Parameters which are required but are variable are shown as lower case and underlined. Those not required are shown as lower case and not underlined.

BEGIN statement.--The BEGIN statement is used in combination with the END statement to enclose a group of instructions as a block. It is generally used with the IF statement or CASE statement.

```
identifier BEGIN
```

The following example shows the BEGIN statement used with the IF statement:

```
        IF (I .EQ. 1) THEN
TRUE    BEGIN
        SET   IVALUE = INVALUE+3
        SET   ICASE = 1
        END   TRUE
        ELSE
FALSE   BEGIN
        SET   IVALUE = 0
        SET   ICASE = 1
        END   FALSE
```

CALL statement.--The CALL statement allows the current procedure to transfer control to a different procedure. When the called procedure completes its function, control returns to the instruction following the CALL statement.

CALL procedure name(parameter string)

where: procedure name = the identifier of the procedure to be called

(parameter string) = up to ten variables or constants separated by commas

The following are several examples of valid CALL statements, some with and some without parameters:

```
CALL  INITIATE
```

```
CALL  PROCESS(1,37,VALUE,ICASE)
CALL  FINISH(RET_CODE)
```

CASE statement.--The CASE statement determines which of the blocks of code which follows it are to be executed based on the value in the specified variable. The ELSE statement must be used in conjunction with the CASE statement and serves to terminate the CASE statement. The block following the CASE is executed if the index equals the number of blocks between the CASE and ELSE statement.

```
CASE  opl
```

where: opl = numeric variable which contains case index

example:

```
        CASE IVALUE
CASE_1  BEGIN
          .
          .
          .
        END   CASE_1
CASE_2  DO  WHILE (I .EQ. 0)
            .
            .
            .
        END  CASE_2
*    FOLLOWING IS CASE 3
        SET  I = I +1
        ELSE
ELSE_CASE  BEGIN
             .
             .
             .
```

END ELSE_CASE

Although the ELSE statement is required, it can be specified as ELSE NULL in which case no action is taken.

CLOSE statement.--The CLOSE statement is used to close a previously opened data file.

CLOSE op1,op2

where: op1 = name of the FILE statement corresponding to the file to be closed

op2 = numeric variable in which completion code is to be stored. Must be numeric variable

Example: CLOSE INPUT_FILE

CLOSE INPUT_FILE,RET_CODE

Data Declaration (DCL) statement.--The data declaration statements describes all variables used in a procedure. All variables used must be declared, and all declarations must precede any executable statements.

identifier DCL length type(subscript),initial value(s)

where: length = the number of positions in a numeric variable or the number of characters in a character variable (default = 1)

type = N for numeric

C for character
subscript = if this variable is to be an array, this value specifies the dimension
initial value(s) = initial value(s) separated by commas. Character values must be enclosed in quotes

Examples:

```
CHAR          DCL   C
CHAR_1        DCL   C,'A'
CHAR_11       DCL   2C,'AA'
CHAR_ABC      DCL   3C,'ABC'
CHAR_A_B_C    DCL   C(3),'A','B','C'
BLANKS        DCL   6C,'      '
STRING        DCL   26C,'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
THREE_STR     DCL   3C(3),ABC,DEF,GHI
V33           DCL   2N,33
VALUE_0       DCL   8N,0
NOT_INIT      DCL   6N
NUM_ARRAY     DCL   1N(3),0,1,2
```

DO statement.--The DO statement allows a block of code, terminated by an END statement, to be executed repeatedly until a specified condition is true. There are two forms of the DO statement.

identifier DO WHILE (op1 opr op2)

where: op1 = variable or constant

opr = .EQ. .NE. .LT. .LE. .GT. .GE.

op2 = variable or constant

or

identifier DO op1 = op2 TO op3 [BY op4]

where: op1 = numeric variable

op2, op3, op4 = numeric variable or numeric constant. op4 is valid only if BY is specified.

Examples:

```
        SET  J = 1
LOOP    DO WHILE (J .NE. 6)
        CALL RETRY
        SET  J = J + 1
        END  LOOP
LOOP    DO  J = 1 TO 10 BY 2
        CALL RETRY
        SET I = I + J
        END LOOP
```

ELSE statement.--The ELSE statement is valid only with the CASE and IF statements. See the description of these instructions for an explanation of the ELSE instruction.

END statement.--The END statement is used to terminate the control instructions BEGIN, DO and PROCEDURE. Its format is:

END identifier

where: identifier = the identifier on the BEGIN, DO or PROCEDURE instruction

See the BEGIN, DO and PROCEDURE instructions for more details and examples.

EXIT statement.--This instruction allows program control to be transferred to the end of any block it is a part of. A block is defined by a BEGIN, DO or PROCEDURE statement.

EXIT,op1

where: op1 = identifier of the block to be exited

Example:

```
TEST  MAIN_PROCEDURE
        .
        .
        .
LOOP  DO  J = 1 TO 100
        .
        .
        .
IF (K .EQ. 7) THEN
EXIT,LOOP
        .
        .
END  LOOP
        .
        .
IF (STRING .EQ. 'STOP') THEN
EXIT,TEST
        .
        .
        .
END  TEST
```

FILE statement.--The FILE statement is used to declare what files the PROCEDURE will use. Two types of files may be referenced, data files or the system console. The file specification can be for either input or output.

identifier FILE DATA

or

identifier FILE CONSOLE

Example: DATA_IN FILE DATA

DATA_OUT FILE DATA

MSG_OUT FILE CONSOLE

CMD_IN FILE CONSOLE

IF statement.--The IF statement allows the selection of two alternate paths of execution based on a test. The second path may be null. The IF statement may be used in conjunction with the ELSE instruction. The ELSE instruction identifies the alternate path of execution but is not required or can be specified as ELSE NULL.

IF (op1 opc op2) THEN

where: op1 = a variable or constant

opc = .EQ. .NE. LT. .LE. .GT. .GE.

op2 = a variable or constant

Examples: IF (X .EQ. 2) THEN

SET Y = 1

ELSE

SET Y = 2

```
*     EXAMPLE 2
            IF  (ANSWER .EQ. 'YES') THEN
YES         BEGIN
            SET  Y = 2
            CALL  RETRY(Y)
            END  YES
            ELSE
NO          BEGIN
            SET  X = 5
            CALL FINISH
            END  NO
*     EXAMPLE 3
            IF (TEST .EQ. VALUE) THEN
            SET  TEST = 1
            ELSE  NULL
                or
            IF (TEST .EQ. VALUE) THEN
            SET  TEST = 1
```

PROCEDURE statements.--There are two procedure statements. The MAIN_PROCEDURE statement can be used only once in a program, and it receives control when the program is started. There may be multiple non-main procedures. The main procedure may reference any non-main procedure, and the non-main procedures may reference any other non-main procedure.

identifier MAIN_PROCEDURE

identifier PROCEDURE(parameter string)

where: parameter string = a list of variable names separated by commas

Variables may be passed from one procedure to another via the parameter string. Each procedure is terminated with an end statement.

Examples:
```
TEST_PGM   MAIN_PROCEDURE
             .
             .
             .
           END  TEST_PGM
FILE_IO    PROCEDURE(VALUE,X)
             .
             .
             .
           END  FILE_IO
```

OPEN statement.--The OPEN statement is used to notify the system that input or output is anticipated by the program for a particular file.

OPEN op1,op2,op3

where: op1 = identifier for a FILE statement

op2 = IN or OUT

op3 = numeric variable in which the return code for the OPEN is to be saved

Examples:
```
OPEN  DATA_IN,IN
OPEN  DATA_OUT,OUT,RETURN_CODE
```

READ statement.--The READ statement is used to read a record or message from a data file or the system console. The data or message is read into the variable until the

length of the variable is satisfied or the record is completed.

READ op1,op2,op3,op4

where: op1 = identifier of a FILE statement

op2 = name of variable where input is to be placed

op3 = length of actual input

op4 = procedure name to receive control if an end of file encountered

Examples: READ DATA_IN,BUFFER,LEN,END_OF_IP

READ MSG_IN,BUFFER

READ DATA_FIL,BUFFER,,END_OF_FILE

RECEIVE statement.--The RECEIVE statement results in data being accepted from an interactive device. The data which is received is compared to the data specified in the RECEIVE statement. If they compare, the next statement is executed. If they do not compare, the system issues messages to the system console giving the last data sent to the device, the data received and the data expected. The operator has the option to accept the data in which case the program will begin execution at the next statement, or the operator can abort the program. Concatenation operations are valid.

RECEIVE op1

where: op1 = a variable or a constant

Examples: RECEIVE 'ID = LX,PSW = X7'

```
RECEIVE  BUFFER
RECEIVE  'TERMINATE'
RECEIVE  FIRST_NAM || LAST_NAME
RECEIVE  'CITY=' || CITY_NAME
```

RETURN statement.--The RETURN statement allows a procedure to return control to the procedure which called it. If the RETURN statement is executed in the main procedure, control is returned to the operating system.

```
RETURN
```

SEND statement.--The SEND statement results in data being transferred to an interactive device. Concatenation operations are allowed. If error conditions are being tested and an error indication is expected then the ERROR parameter should be set to YES.

```
SEND  op1,[ERROR=op2]
where: op1 = variable or constant
       op2 = NO if no error expected (default).
           = YES if error expected
             op2 is valid only if ERROR= is
             specified
Examples:  SEND  'ENTER LAST NAME'
           SEND  'CITY= ' || CITY_NAM
           SEND  'CITY= ' || CITY || 'STATE='|| STATE
           SEND  'A',ERROR=YES
```

SET statement.--The SET statement allows the value of a variable to be changed. All variables used in the statement must be of the same type (i.e., character or numeric). Multiple arithmetic or concatenation operations may be performed, however, arithmetic functions are valid only for numeric variables and constants. Concatenation functions are valid only for character variables and constants.

```
SET  op1 =  op2 opr op3

where:  op1 = variable to be altered
        op2 = variable or constant
        opr = +-/* if numeric variables involved
            = || if character variables involved
        op3 = variable or constant

Examples:  SET  ADR =  CITY || STATE
           SET  NAME =  'JOHN' || 'D' || 'DOE'
           SET  COUNT =  0
           SET  COUNT =  X + Y - 1
```

STOP statement.--The STOP statement can be issued at any point in the program and results in immediate termination of the program.

```
STOP
```

WRITE statement.--The WRITE statement is used to write a record or message to a data file or to the system console. The data written may be contained in a variable, or it may be a character constant.

WRITE op1,op2

where: op1 = identifier of a FILE statement

op2 = name of variable to be outputted or character constant

Examples: WRITE DATA_OUT,MESSAGE

WRITE MSG_OUT,'TEST COMPLETE'

Standard Functions Provided in ATLIS

Five functions are provided for in ATLIS: LENGTH, MOD, RAND, SELECT, and SUBSTR. The LENGTH, MOD and RAND functions are valid any place a numeric variable is valid. The SELECT and SUBSTR functions are valid any place a character variable is valid.

LENGTH function.--The LENGTH statement provides the number of characters currently in a character variable and treats the length as a numeric variable.

LENGTH(op1)

where: op1 = identifier for a character variable

Examples: SET A = LENGTH (ADDR_FIELD)

IF (LENGTH(ADDR_FIELD) .GT. 0) THEN

MOD function.-- The MOD function provides the remainder of the first operand divided by the second operand.

MOD(op1,op2)

where: op1 and op2 = numeric variable or numeric constant

Examples: SET X = 6 + MOD(11,5)

SET VALUE = SUM / MOD(X,10)

RAND function.--The RAND function provides a numeric value between 0 and 9. The probability of the value being any of the values is equal.

RAND(op1)

where: op1 = numeric variable which serves as a seed

Examples: DO I = 1 TO RAND(X) * 10

CASE RAND(X)

SELECT function.--The SELECT function provides a character string of a specified length which is selected randomly from a character variable. If the character variable is length one then the new string is made up of the same character.

SELECT(op1,op2)

where: op1 = numeric variable or numeric constant which specifies the length of character string to be generated

op2 = character variable or character constant from which the new string is to be randomly selected

Examples: SET STR = SELECT(6,ALPHABET)

SET NEW_STR = SELECT(RAND(X),NUMERIC)

SEND SELECT(10,ALPHA_NUMERIC)

SUBSTR function.--The substring function selects a part of another character variable, starting with a specified position and with the specified length.

SUBSTR(op1,op2,op3)

where: op1 = numeric variable or constant specifying the starting position in op3 to select string

op2 = numeric variable or constant specifying the length of the string to be selected

op3 = character variable from which new string is to be selected

Examples: SET CHR_STR = SUBSTR(1,10,NAME)

SEND SUBSTR(RAND(X),4,ALPHA)

The ATLIS Compiler and Linkage Editor

The ATLIS compiler is to be written in a high level, machine independent language such as PL/I. The compiler can be a cross compiler, that is, it executes on a machine different than the one on which the generated code is used, or it can be a resident compiler on the machine on which the code is used. The compiler produces "pseudo code," that is, code which in itself is not executable but must be interpretively executed by the AIT System (Automated Interactive Test System). Since the ATLIS procedures may be compiled separately, references to other procedures (CALL statements) and data references to the main procedure by other procedures

are marked as external references which are resolved at linkage edit time.

Because the generated code is executed interpretively and because it is desirable to execute multiple programs concurrently, main memory requirements might be a problem if the system is implemented on a mini-computer. For that reason, the compiler produces "paged" pseudo code. This means that each procedure is divided into pieces, or pages, which have a maximum size. The smaller the size selected for the page, the more pages which are required and therefore the more page fetches from disk. The larger the page size the more memory that is required. Each program in execution has one page area,and it contains the page currently being executed.

Since the pseudo code is paged then the compiler determines where the page breaks occur and provides a pseudo instruction to so indicate. The compiler also satisfies addressing references between pages for any branches generated by the control statements. All address references within a procedure dealing with the transfer of control are expressed internally as relative offsets within pages, and the page numbers are relative to a procedure.

The data for a procedure presents a somewhat different problem. If data was treated similar to the pseudo code then as each page or procedure was executed or called, the previous data would be destroyed. Therefore the data must

be kept separately. The data for the main procedure always remains in memory since it can be referenced by any procedure. The data for all other procedures is present from the time it receives control until the time a RETURN statement is encountered or until the END statement for the procedure is encountered. This means that if there are nested procedure calls, the data for each procedure which has not executed a return is in memory. For an example, see Figure 2.

The compiler therefore separates the data from executable code when producing the object module so that at execution time the AIT System can retrieve the data and the pseudo code separately. This is discussed in more detail when the AIT System is discussed.

The function of the ATLIS linkage editor is to take the object modules produced by the ATLIS Compiler and combine them to form an executable load module. This module is then executed by the AIT System. This involves assigning relative procedure numbers to each procedure, the main procedure being procedure number one, and supplying the relative offset within the main procedure data space for references to data items in the main procedure by other procedures.

The input to the linkage editor is a group of object modules, and its output is an executable load module if no errors were encountered. In addition, the linkage editor provides printed output which contains a list of each procedure and which procedures call it. Also, a cross reference of

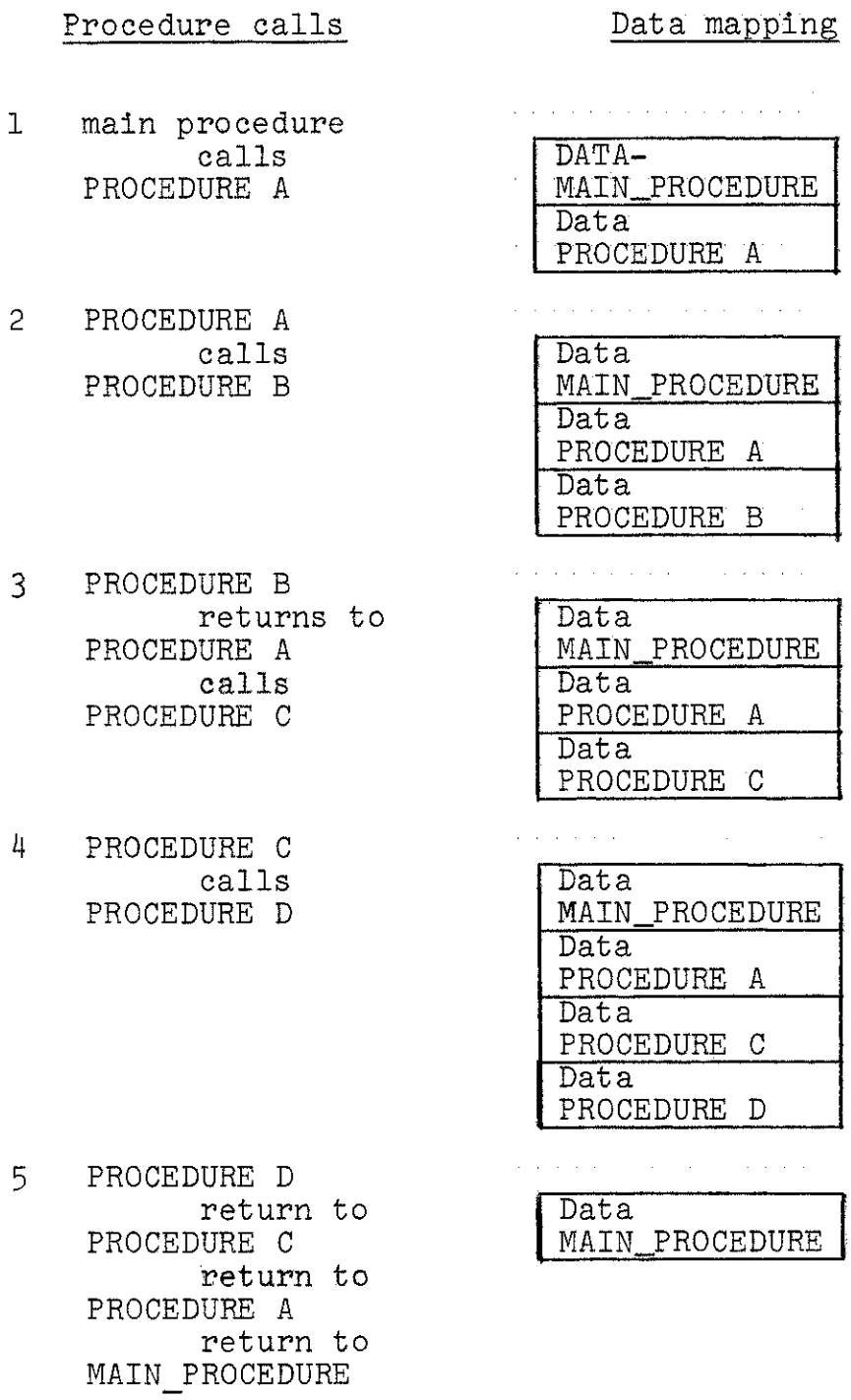

Fig. 2--An example of the data area mapping for several levels of procedure calls.

data variables in the main procedure and which procedures reference each item is provided. All references are indicated by relative procedure number and statement number within the procedure. The listing also indicates unresolved references.

The Automated Interactive Test System (AIT System)

The function of the Automated Interactive Test System, to be referred to as the AIT System, is to interpretively execute the load modules produced by the ATLIS compiler and linkage editor. The AIT System would be best implemented on a 16 or 32 bit general purpose mini-computer with at least 65K of memory. The system should also support card input, print output, magnetic tape (800 or 1600 BPI) and mass storage device or devices of at least 3 million 8 bit bytes of storage.

The system can be implemented as a system within itself or as a subsystem under an operating system. If implemented as a system itself then it must include device drivers to control each of the devices described above. The more desirable choice, and the one to be assumed for the purposes of this discussion, is for the AIT System to execute as a subsystem under an operating system. The operating system would be responsible for device management, memory management and job management if it is a multijob operating system.

It is recommended that the AIT System be a multitask system which is capable of supporting multiple interactive devices, each as a separate task. These tasks are able to be started and stopped independent of each other. In addition, there is a main task that coordinates the starting and stopping of the other tasks and provides for the operator interface through the system console.

The AIT System is coded in the assembly language of the computer it is being implemented on or in a high level systems programming language if one is supported. Depending on memory constraints and operating system capabilities, it may be necessary to overlay parts of the AIT System but only those functions which are not of a time critical nature. Candidates for being overlayed include the system console interface, system initialization and task initialization.

The system supports at least two types of disk file organization. These organizations are logical sequential files and partitioned files. A logical sequential file is a file whose records may only be accessed in sequence from start to finish. A partitioned file is effectively one or more logical sequential files, called members, which are grouped together under one name. Each member of the partitioned file has its own unique member name and can be accessed separately, or the entire partitioned file can be treated as one large sequential file.

The partitioned file organization is necessary for the object modules produced by the ATLIS compiler. Each procedure of a program has its own unique name, and the object code produced by the ATLIS compiler for that procedure is placed in a member with the same name. The name of the partitioned file is the same as the name of the ATLIS program. Thus, just as a program is made up of a group of procedures, the partitioned file for a program is made up of a group of members, each of which represents one procedure. The output of the linkage editor is also maintained in a partitioned file. As a procedure is referenced, the pseudo code and data for that procedure is loaded from the appropriate member.

The logical sequential file organization is needed for ATLIS source files, the AIT System log file and other data files.

The AIT System is controlled through the system console interface. The operator at the console specifies which interactive devices are to be tested, assigns each device a relative priority, specifies what tests are to be executed and starts the appropriate group of tests. Each program name is made up of three parts, the group name followed by the sub group name followed by the sequence number. The format of the program name is:

(group name/subgroup name.group sequence number).

Since a program is maintained as a file, the program name is also the file name. The purpose of having three parts to a

program name is to allow programs to be executed by group name, subgroup name or by sequence number. For example, assume there were the programs with the following names:

(INPUT/TEST1.1)

(INPUT/TEST1.2)

(INPUT/TEST1.4)

(INPUT/TEST2)

(INPUT/TEST3)

(INPUT/TEST6)

(OUTPUT/TEST1)

(OUTPUT/TEST2.1)

(OUTPUT/TEST2.2)

(OUTPUT/TEST2.3)

(OUTPUT/TEST3)

(OUTPUT/TEST4)

When a group of tests are executed, they are executed in alphanumeric order (i.e., A-Z, 0-9). The control operator can specify execution of a specific program such as (INPUT/TEST1.4). The operator can also specify execution of a subgroup of tests such as (INPUT/TEST1) in which case (INPUT/TEST1.1), (INPUT/TEST1.2) and (INPUT/TEST1.4) are executed sequentially. The operator can also execute a group of tests such as (OUTPUT) in which all programs with the group name of (OUTPUT) are executed sequentially. If all programs are to be executed, the operator can specify "ALL" in which case all programs are executed sequentially based on alphanumeric order of their group name.

Grouping tests in this manner is an adaptation of the idea of organizing tests into a testing structure presented by Cicu (1, pp. 44-50). The advantage of this capability is that the operator can execute all tests if confidence testing of the entire system is necessary, or specific programs can be executed if problems exist or are suspected with a certain area. In addition, a repeat count can be specified, allowing a program or group of programs to be executed repeatedly.

This flexibility is provided for each interactive device to be tested, so each device can be tested concurrently with the same program or group of programs.

When the control operator indicates that execution is to begin, the AIT System obtains from the operating system sufficient memory to contain a page of ATLIS pseudo code and the data for the main procedure of the program. The first page of pseudo code for the main procedure is loaded into memory from the program file and interpretive execution begins. When a page boundary is reached, the next page is read in from the program file.

If another procedure is called, its code is located in the partitioned file, and the first page of its pseudo code is read into memory. In addition, sufficient memory is requested from the system to contain the data area of the new procedure. If sufficient memory is not available, the AIT System continues to request the needed memory for a period of time. If still unsuccessful, the control operator is notified

via the system console that insufficient memory is available to continue. The control operator then tells the system to wait, continue to retry or terminate the program.

When a procedure has completed execution and is ready to return to the calling procedure, the data space for the procedure is released back to the system, the page of pseudo code in the calling procedure is reloaded and execution resumes at that point. When a program completes execution, the AIT System checks if there are additional programs to be executed, and if so, the next one in the group is located and the process begins again.

Part of the AIT System is the ATLIS interpreter which acts upon the pseudo code produced by the ATLIS compiler. After a predetermined number of pseudo instructions have been executed for a program testing a certain device, the interpreter determines if there are other tasks (programs) awaiting execution. If there are, they begin execution based on priority and aging count. Execution of most of the instructions is straight forward. The exceptions to this are the CALL statement, which was described previously, SEND statement and RECEIVE statement.

The SEND and RECEIVE statements are the heart of the AIT System. When a SEND instruction is encountered, the data which forms the argument for the SEND instruction is moved into a holding buffer and also written to the log file. The data is then transmitted a character at a time. Assuming

that the device operates in echo-plex mode (i.e., the receiving device echos the characters back to the sending device), the AIT System compares the characters sent to be sure that echoed characters match those transmitted. Any discrepancy is noted in the log. Should an error indication be received from the receiving device, it is so noted in the log. An error indication can result from data being altered erroneously during transmission or as a result of intentionally sending bad data to test error detection logic. In order to determine which case has occurred, the interpreter, upon receiving an error indication, retransmits data from the holding buffer beginning with the character following the last character echoed by the receiving device. This is repeated a set number of times. If the error condition still exists then the data must be in error. The interpreter then checks if the SEND statement specified ERROR=YES, meaning an error is expected. If it did, the next instruction is executed. If the SEND statement had ERROR=NO then an unexpected error has occurred, and a message is issued to the control console to alert the operator. This message gives the program name, the procedure name, the statement number, what was sent and what was echoed. The console operator can override the error and go on, change the data being sent or cancel the program. If all data sent is echoed correctly, the interpreter checks to be sure that the SEND statement did not specify ERROR=YES. If it did then an error was expected,

but it was not received. A message is issued to the system console to notify the operator.

The RECEIVE instruction is treated as follows. Data from the interactive device being tested is received into a holding buffer. As it is received, it is compared to the data given as an argument on the RECEIVE instruction. If any variations are encountered, a message is issued to the control console. The console message indicates the program name, procedure name, statement number, last data sent, data expected and actual data received. The operator has the option to accept the data, override the data or abort the program. The same type of message is issued if the interpreter stops receiving data prior to completing input. Once the correct number of characters has been received, the next instruction is executed.

The basic premise of this type of testing is that the response of the system under test should always be anticipated based on what is sent. The purpose of the testing is to insure that the system responds as expected to predefined sets of input. The SEND and RECEIVE statements are the cornerstone of this testing procedure.

Conclusions

As described in Chapter I, testing of interactive systems can be very time consuming. Therefore, it is desirable to automate this procedure if possible. The techniques and

tools which are discussed in Chapter III offer little in the way of solutions to this problem. On the other hand, the ATLIS language and the AIT System provide a solution to the problem of interactive system testing.

The ATLIS language and AIT System have the necessary features to allow the automation of interactive system testing. Because the testing is done through programs written in the high level language ATLIS, new features of the interactive system can be tested by changing existing programs or by writing new programs. Thus, flexibility is a feature of this solution. New features can be easily tested.

Another advantage to this solution is repeatability. If problems are found with part of the interactive system, the appropriate test program can be executed to reproduce the problems. Since the system offers an execution trace, it is possible to review the trace to determine where the errors occur and what events preceded them.

This system is also modular. Because of the way programs can be named using the format (group name/subgroup name.group sequence number), a group of programs or a single program can be executed. Thus, a specific area can be tested or the entire system can be tested. The test programs are also easy to execute. The control operator must only specify the name of the group, subgroup or program desired. If no errors are encountered, no further operator intervention is required.

Since the test programs are easy to execute, there is less time required to execute them. The tests can be conducted faster by the system than by human testers, because there is no reaction time on the part of the system as there is with human testers. The input can be generated at the speed of the communications link between the system being tested and the AIT System.

The features of the ATLIS language also facilitate the development of test programs. For example, the SELECT and SUBTR functions allow generation of character strings, and when combined with the DO statement, they allow various combinations of character strings to be easily generated as input data.

The features described above enable the ATLIS language and AIT System to aid in the automation of the testing involved in interactive systems. In particular, they aid in the testing associated with system testing, alpha testing, functional testing and regression testing as described in Chapter II.

Although performance testing is not specifically discussed in this chapter, it is quite possible that the AIT System can be expanded to include performance testing. The code produced by the ATLIS Compiler is interpretable and paged, and since the AIT System is designed as a multi-task system, expansion of the system to include performance testing should not be too difficult. If timing constraints

are to be considered then new statements may be required in the ATLIS language such as START_TIMER and STOP_TIMER which provide a time delta for a sequence of tests. This time delta can then be compared to what is acceptable in order to determine if the system meets predefined timing constraints.

The ATLIS language and the AIT System, as described here, provide a tool for automating the testing of interactive systems. Use of such a system should significantly reduce the time required to test interactive systems and should aid in the development of thorough test procedures for these systems.

CHAPTER BIBLIOGRAPHY

1. Cicu, A., "Organizing Tests During Software Evaluation," Proceedings International Conference on Reliable Software, June, 1975, 43-50.

2. Scherr, A. L., "Developing and Testing a Large Programming System, OS/360 Time Sharing Option," Program Test Methods, edited by William C. Hetzel, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1973, 165-180.

APPENDIX I

Backus-Naur Form (BNF) of ATLIS Language

<program>::= <main procedure>
|<main procedure><sub procedures>

<main procedure> ::= <main procedure statement><end statement>
|<main procedure statement>
<procedure body><end statement>

<subprocedures> ::= <procedure>
|<procedure><subprocedures>

<procedure> ::=<procedure statement><end statement>
|<procedure statement>
<procedure body><end statement>

<main procedure statement> ::= <procedure name>MAIN_PROCEDURE

<end statement> ::= END <statement identifier>

<procedure statement> ::= <procedure name> PROCEDURE <argument list>
|<procedure name> PROCEDURE

<procedure name> ::= <statement identifier>

<argument list> ::= (<arguments>)

<arguments> ::= <data identifier>
|<data identifier>,<arguments>

<procedure body> ::= <executable block>
|<data declarations><executable block>

```
<data declarations> ::= <data group>
                      |<file group>
                      |<data group><file group>
<data group> ::= <data statement>
               |<data statement><data group>
<data statement> ::= <data identifier> DCL<data description>
<data description> ::= <type>
                     |<length><type>
                     |<type><sub-init>
                     |<length><type><sub-init>
<sub-init> ::= (<numeric constant>)
             |<numeric constant>),<init-string>
             |,<init-string>
<init-string>::= <constant>
              |<constant>,<init-string>
<length> ::= <numeric constant>
<type> ::=  N | C
<file group> ::= <file statement>
               |<file statement><file group>
<file statement> ::= <file identifier> FILE DATA
                   |<file identifier> FILE CONSOLE
<executable block> ::= <control block>
                     |<control block><executable block>
<control block> ::= <simple statement>
                  |<if block>
                  |<statement identifier><do block>
```

```
                    <end statement>
                   |<case block><end statement>
                   |<begin block><end statement>
<simple statement> ::= <set statement>
                       |<call statement>
                       |<exit statement>
                       |<I/O statement>
                       | RETURN
                       | STOP
<set statement> ::=  SET <variable><expression>
<expression> ::= <arithmetic exp>
                |<concat exp>
<arithmetic exp> ::= <term>
                     |<arithmetic exp><add/sub operator><term>
<term> ::= <signed operand>
          |<term><multiply/divide operand><signed operand>
<signed operand> ::= <add/sub operator><signed operand>
                     |<operand>
<operand> ::= <constant>
             |<variable>
             |<function reference>
             |(<arithmetic exp>)
<variable> ::= <simple variable>
              |<subscripted variable>
<simple variable> ::= <numeric identifier>
<subscripted variable> ::= <numeric identifier>(<arithmetic exp>)
```

```
<function reference> ::= <function name>
                        |<function name>(<arithmetic exp>)
<function name> ::= <identifier>
<numeric identifier> ::= <data identifier>
<concat exp> ::= <concat operand>
                |<concat operand> || <CONCAT exp>
<concat operand> ::= <constant>
                    |<variable>
                    |<function reference>
<call statement> ::=  CALL <procedure name>
                    | CALL <procedure name><argument list>
<exit statement> ::=  EXIT,<statement identifier>
<I/O statement> ::= <send statement>
                   |<receive statement>
                   |<open statement>
                   |<close statement>
                   |<read statement>
                   |<write statement>
<send statement> ::=  SEND <concat exp>
<receive statement> ::=  RECEIVE <concat exp>
<open statement> ::= <file identifier><disp><return>
                    |<file identifier><disp>
<disp> ::=  IN | OUT
<return> ::= <variable>
<close statement> ::= <file identifier>
                     |<file identifier><completion>
```

```
<completion> ::= <variable>
<read statement> ::= <file identifier><variable>
                   |<file identifier><variable>,<variable>
                   |<file identifier><variable>,<variable>,
                    <variable>
                   |<file identifier><variable>,,<variable>
<write statement> ::= <file identifier><variable>
<if block> ::= <if header><executable block>
             |<if header><executable block><else block>
<if header> ::=  IF <condition> THEN
<condition> ::=  (<expression><relational operator><expression>)
<else block> ::=  ELSE <control block>
<do block> ::= <do while><executable block>
             |<do range><executable block>
<do while> ::=  DO WHILE <condition>
<do range> ::=  DO <variable> = <operand> TO
              <operand> BY <operand>
<case block> ::= <case statement><executable block>
               |<case statement><executable block><else block>
<case statement> ::=  CASE <variable>
<begin block> ::= <begin statement><executable block>
<begin statement> ::= <statement identifier> BEGIN
<file identifier> ::= <identifier>
<data identifier> ::= <identifier>
<statement identifier> ::= <identifier>
<identifier> ::= <alpha>
```

```
                |<alpha><alpha-num-string>
<alpha-num-string> ::= <alphanumeric>
                     |<alphanumeric><alpha-num-string>
<alphanumeric> ::= <alpha>
                  |<numeric>
<constant> ::= <numeric constant>
              |<character constant>
<numeric constant> ::= <numeric>
                     |<numeric><numeric constant>
<character constant> ::= '<character>'
                        |'<character><character constant>'
<character> ::= <alpha>
               |<numeric>
               |<special>
<add/sub operator> ::=  + | -
<multiply divide operator> ::=  * | /
<relational operator> ::=  .EQ. | .NE. | .LT.
                         | .LE. | .GT. | .GE.
<numeric> ::=  0|1...|9
<alpha> ::=  A|B|...|Z
<special> ::=  ¢|.|<|...|"
```

APPENDIX II

ATLIS Language Summary

STATEMENTS:

identifier	BEGIN
	CALL procedure name(op1,op2...opn)
	CASE op1
	CLOSE op1,op2
identifier	DCL lengthtype(subscript),initial values
identifier	DO WHILE (op1 opR op2)
identifier	DO op1 = op2 TO op3[BY op4]
	ELSE
	END identifier
	EXIT,op1
identifier	FILE DATA
identifier	FILE CONSOLE
	IF (op1 opr op2) THEN
identifier	MAIN_PROCEDURE
	OPEN op1,op2,op3
identifier	PROCEDURE (op1,op2,...opn)
	READ op1,op2,op3,op4
	RECEIVE op1
	RETURN
	SEND op1,[ERROR=op2]

SET op1 = op2 opr op3

STOP

WRITE op1,op2

FUNCTIONS:

LENGTH(op1)

MOD(op1,op2)

RAND(op1)

SELECT(op1,op2)

SUBSTR(op1,op2,op3)

BIBLIOGRAPHY

Books

Hetzel, William C., editor, Program Test Methods, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1973.

IEEE/ARINC Standard ATLAS Test Language, IEEE std 416-1976, The Institute of Electrical and Electronic Engineers, Inc., 445 Hoes Lane, Piscataway, New Jersey 08854, 1976.

Proceedings International Conference on Reliable Software, SIGPLAN NOTICES (a monthly publication of the ACM special interest group on programming languages), Volume 10, Number 6, ACM SIGPLAN, 1133 Avenue of the Americas, New York, New York 10036, June, 1975.

Proceedings Second International Conference on Software Engineering, IEEE Catalog Number 76CH1125-4 C, IEEE Computer Society, 5855 Naples Plaza, Long Beach, California 90803, October, 1976.

Rustin, Randall, editor, Debugging Techniques in Large Systems, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1971.

Articles

Baker, F. T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Volume 11, Number 1, 1972.

Boehm, Barry W., "Software and Its Impact: A Quantitative Assessment," DATAMATION, May, 1973, 48-59.

Cicu, A., "Organizing Tests During Software Evaluation," Proceedings International Conferences on Reliable Software, June, 1975, 43-50.

Duke, M. O., "Testing In a Complex Systems Environment," IBM Systems Journal, Volume 14, Number 4, 1975, 353-365.

Elmendorf, W. R., "Disciplined Software Testing," Debugging Techniques In Large Systems, edited by Randall Rustin, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1971, 137-140.

Elspas, Bernard, Karl N. Levitt, Richard J. Waldinger and Abraham Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys, Volume 4, Number 2, June, 1972, 97-147.

Good, Donald I., Ralph L. London and W. W. Bledsoe, "An Interactive Program Verification System," Proceedings International Conference on Reliable Software, June, 1975, 482-492.

Gruenberger, F., "Program Testing: The Historical Perspective," Program Test Methods, edited by William C. Hetzel, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1973, 11-14.

Hantler, Sidney L. and James C. King, "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys, Volume 8, Number 3, September, 1976.

Huang, J. C., "An Approach to Program Testing," ACM Computing Surveys, Volume 7, Number 3, September, 1973, 113-128.

King, James C., "A New Approach to Program Testing," Proceedings International Conference on Reliable Software, June, 1975, 228-233.

Miller, E. F. and R. A. Melton, "Automated Generation of Testcase Datasets," Proceedings International Conference on Reliable Software, June, 1975, 51-58.

Mills, Harlen, "Top Down Programming in Large Systems," Debugging Techniques In Large Systems, edited by Randall Rustin, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1971, 41-56.

Ramamoorthy, C. V. and S. F. Ho, "On the Automated Generation of Program Test Data," Proceedings Second International Conference on Software Engineering, October, 1976, 636.

_____, "Testing Large Software With Automated Software Evaluation System," Proceedings International Conference on Reliable Software, June, 1975, 382-394.

Sammet, Jean E., "Roster of Programming Languages for 1974-75," Communications of the ACM, Volume 19, Number 12, December, 1976, 655-669.

Scherr, A. L., "Developing and Testing a Large Programming System, OS/360 Time Sharing Option," Program Test Methods, edited by William C. Hetzel, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1973, 165-180.

Supnik, Robert M., "Debugging Under Simulation," Debugging Techniques In Large Systems, edited by Randal Rustin, Englewood Cliffs, New Jersey, Prentice Hall, Inc., 1971, 117-136.

Vander Noot, T. J., "System Testing .... A Taboo Subject," DATAMATION, November 15, 1971, 60-64.

Reports

Improved DETOL Programming for the 5500/5510 Automatic Test System (ATS), Report Number ER-8964, AAI Corporation, Cockeysville, Maryland, March, 1977.

Public Documents

Miller, E. F., Methodology for Comprehensive Software Testing, National Technical Information Service (NTIS) AD/A013 111, June, 1975.

Montgomery, George Wynn, System Test Methodology, National Technical Information Service (NTIS) AD/A-012 461, June, 1975.

Unpublished Materials

Ring, Steven J., "A Distributed Intelligence Automatic Test System for PATRIOT Electronic Assemblies," for publication in IEEE Transactions on Aerospace and Electronic Systems, Raytheon Company, Hartwell Road, Bedford, Mass. 01730, 1977.