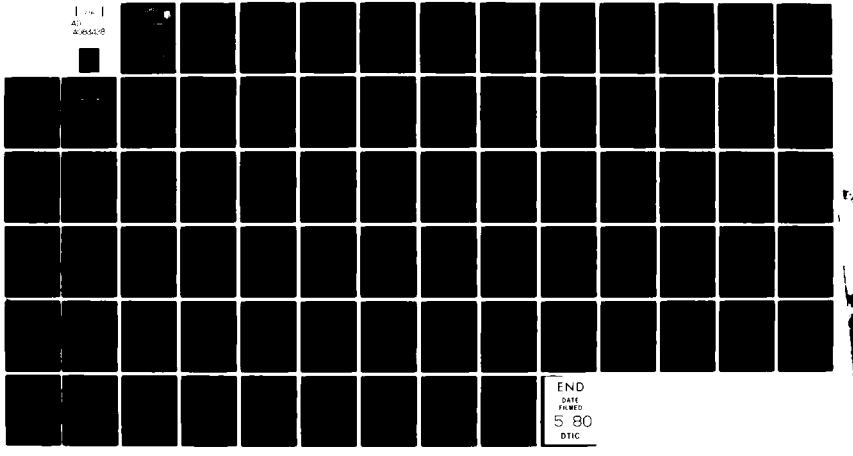
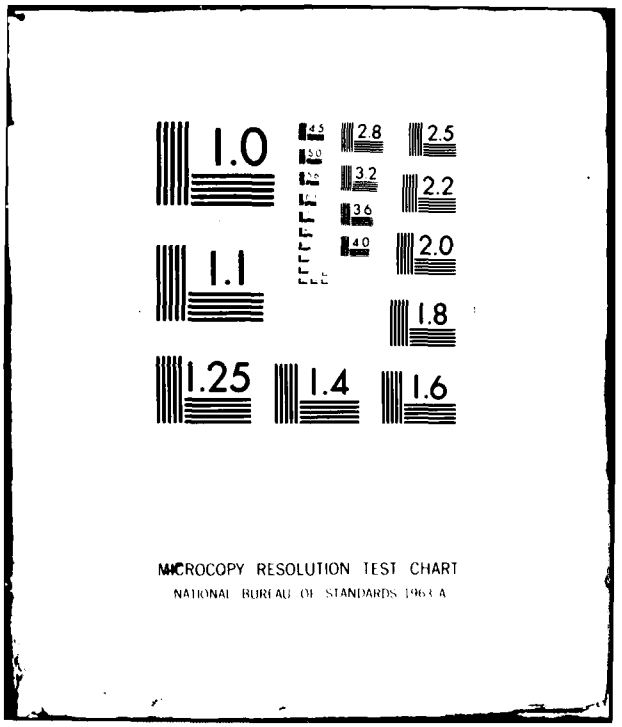
AD-A083 428 LOGICON INC LEXINGTON MA F/6 9/2
AUTOMATED TEST CASE GENERATOR.(U)
FEB 80 D M LEACH, S KUNDU F30602-78-C-0231
UNCLASSIFIED RADC-TR-80-31 NL



END
DATE FILMED
5 80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

AUTOMATED TEST CASE GENERATION

Robert M. Leech
Dr. Zbigniew Stachurski

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

[illegible]

[illegible]

[illegible]

Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the mailing list, or if the addressee is no longer employed by your organization, please notify [illegible] Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(19) REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (18) RADC-TR-80-31	2. GOVT ACCESSION NO. / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) AUTOMATED TEST CASE GENERATOR	5. TYPE OF REPORT & PERIOD COVERED (9) Final Technical Report Aug 78 - Aug 79
	6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) (10) Daniel M. Leach, Sukhamay Kundu	8. CONTRACT OR GRANT NUMBER(s) (15) F30602-78-C-0231
9. PERFORMING ORGANIZATION NAME AND ADDRESS Logicon, Inc. 18 Hartwell Avenue Lexington MA 02173	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (16) 63728F 25310201 (17) 02
11. CONTROLLING OFFICE NAME AND ADDRESS (11) Rome Air Development Center (ISIE) Griffiss AFB NY 13441	12. REPORT DATE Feb 80
	13. NUMBER OF PAGES 69
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same (12) 76	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Frank S. LaMonica (ISIE)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Testing
Computer Software
Test Case
Test Data

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report summarized the results of a study to advance the state-of-the-art in the area of automated software testing. Specific goals were to develop a specification for a system which provides for the automated selection of test cases and the automated generation of test data. The report describes the requirement for such a system and presents a design approach which will fulfill the requirement. Included is a sample software program which is manually carried through the various steps of the

→ next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409110

cont →

proposed system. The report closes with a survey of the state-of-the-art of software testing.

ABSTRACT

This report explains the need for an Automated Test Case Generator and presents a design approach which would fulfill this need. An example is manually carried through the various steps which will be performed by the proposed Automated Test Case Generator. The report closes with a survey of the state of the art of program testing.

ACKNOWLEDGEMENTS

Special thanks goes to Dr. Sukhamay Kundu who was the principal investigator and provided the theoretical foundation for the project.

This document was reviewed by S. Kundu, S.G. Farnum, S.C. Vorenberg, L.A. Johnson, and P.A. Chiavacci. Special thanks goes to them. Special thanks also for the patience and skills in technical typing and proofing goes to Ms. Marcia Brehm, Ms. Deborah Queen, Ms. Dale Thompson, Ms. Sandra Foster, and Ms. Judy Bunker.

TABLE OF CONTENTS

			Page
SECTION	1.	INTRODUCTION	1-1
	1.1	Purpose of Project	1-1
	1.2	Purpose of ATCG	1-2
	1.3	Producing Flowcharts	1-6
	1.4	A New Algorithm for Test Case Selection	1-6
	1.5	Test Data Generation	1-10
	1.6	Feasibility Analysis	1-11
SECTION	2.	MANUAL ANALYSIS OF AN EXAMPLE	2-1
	2.1	Process User Commands	2-1
	2.2	Analyze Structure	2-3
	2.2.1	Segment Program	2-3
	2.2.2	Form DD-paths	2-3
	2.2.3	Produce Flowcharts	2-5
	2.2.4	Graph Program Using DD-paths	2-5
	2.3	Analyze Data Flow	2-5
	2.4	Perform Symbolic Analysis	2-9
	2.5	Generate Constraints	2-10
	2.5.1	Traverse Last Path	2-11
	2.5.2	Select Path	2-12
	2.5.2.1	Select Segment	2-12
	2.5.2.2	Select Reaching Set	2-13
	2.5.2.3	Traverse Selected Path	2-14
	2.5.3	Negate Previous Constraints	2-15
	2.5.3.1	Select Target Constraints	2-15
	2.5.3.2	Generate Data Constraints	2-15
	2.5.4	Analyze Boundary Conditions	2-16
	2.6	Generate Data	2-17
	2.7	Subsequent Steps	2-18

TABLE OF CONTENTS

			Page
SECTION	3.	SURVEY OF THE STATE OF THE ART IN PROGRAM TESTING	3-1
	3.1	Prelude to Testing	3-1
	3.1.1	Hierarchical Design	3-1
	3.1.2	Structured Programming	3-1
	3.1.3	Acceptance Criteria	3-2
	3.1.4	Test Beds	3-3
	3.2	Data Flow Analysis	3-3
	3.2.1	Type I Anomolies	3-3
	3.2.2	Type II Anomolies	3-3
	3.2.3	DAVE	3-4
	3.3	Structural Analysis	3-4
	3.3.1	Restructuring Programs	3-4
	3.3.2	Code Instrumentation	3-4
	3.3.3	Flowcharts	3-5
	3.3.4	Program Graphs	3-5
	3.4	Symbolic Analysis	3-5
	3.4.1	Program Proofs	3-6
	3.4.2	DPEN	3-6
	3.4.3	Symbolic Execution	3-6
	3.4.3.1	Binding	3-6
	3.4.3.2	The Array Variable Problem	3-7
	3.4.3.3	The Loop Variable Problem	3-8
	3.4.3.4	Dealing with Symbolic Ambiguities	3-8
	3.4.3.5	Consistency Checking	3-9
	3.5	Test Case Selection	3-10
	3.5.1	New Data from a Path	3-10
	3.5.2	Path from New Data	3-10
	3.5.3	Analyzing Boundary Conditions	3-11
	3.6	Closing Remarks	3-12
APPENDIX A		GLOSSARY	A-1
APPENDIX B		BIBLIOGRAPHY	B-1
APPENDIX C		AMPIC	C-1

LIST OF FIGURES

Figure		Page
1	Data Flow for the Automated Test Case Generator	1-4
2	Major Processing Steps in the Automated Test Case Generator	1-5
3	A Detailed Flowchart and Its Block Structure	1-7
4	A High Level Form of the Flowchart with Expansion Level 3	1-8
5	A High Level Form of the Flowchart with Expansion Level 2	1-9
6	Program SAMPLE	2-2
7	Segment Listings	2-4
8	DD-path Listings	2-6
9	Flowcharts	2-7
10	Program Graphs	2-8

EVALUATION

The purpose of this contractual effort was to advance the state-of-the-art in the area of automated software testing. A critical problem addressed by this effort was the specification of an automated system which provides the capability to generate test cases and test data that cause the execution of particular paths in a software program. This capability is of importance during the testing phase of the software development life cycle because the current manual generation of test cases and test data is difficult, and expensive in terms of both personnel and computer costs. This effort was responsive to the objective of the RADC Technology Plan, TPO R5A, "Software Cost Reduction."

Frank S. La Monica

FRANK S. LA MONICA
Project Engineer

1. INTRODUCTION

1.1 Purpose of the Project

Software system certification is a critical information processing problem facing the Air Force. Because software is a major element in military systems, large amounts of time and money are currently spent by the Air Force in an effort to correct, maintain, and certify software systems. In spite of this investment, many times the results of testing are unsatisfactory. Systems thought to be correct (and purchased as such by the Air Force) may suddenly produce incorrect results, no results, or behave erratically because special conditions in the data or in the operating environment were not provided for in the program logic and were not encountered in testing. Under a previous RADC procurement, an Automated Verification System (JAVS) for the JOVIAL J3 language was developed. That system provides the developer/tester with the capabilities to accurately measure the effectiveness of a particular test and to determine whether the set of test data has thoroughly exercised the software. A critical problem which remains to be solved is the automated generation of test cases and test data that will cause the execution of particular paths in the code which were not previously tested. Another important problem area is that of generating test data which correspond to special operating conditions of the software for which there may not exist easily identifiable paths in the source code. Current manual generation of test cases and test data is difficult and expensive in terms of both personnel and computer costs.

In the course of this project we have produced both Functional Descriptions and System/Subsystem Specifications for Automated Test Case Generators (ATCG)[1] to operate in two different (though similar) environments.

---

[1] ATCG is used throughout this document to refer both to the tool (Automated Test Case Generator) and the process used by the tool (Automatic Test Case Generation).

The first version of the system which was specified is intended for generation of test cases and test data for programs written in a general algebraic language. This version presumes that an execution monitor for the language exists in the environment in which ATCG is to be run. The second version of the system is intended for the generation of test cases and test data for programs written in the JOVIAL programming language and integrates with the JOVIAL Automated Verification System (JAVS).

1.2 Purpose of ATCG

The purpose of ATCG is to automate the selection of test cases and the generation of test data. The test data so generated will test those software features which can be related to the program structure as well as other software features which do not have a direct relationship to the program structure but are inherent in the specific tasks performed by the software.

ATCG will choose one test case and one corresponding test data set each time the program is submitted to it. Each such submission is termed a "test case generation session." During the first session, a data base will be built for the program being tested which will be used during succeeding sessions. ATCG will have three different methods of test case selection and two modes of system operation.

The two modes of system operation will be batch and interactive. The three methods used to generate new test cases will be path selection, negation of previous constraints, and boundary condition analysis.[1]
Refer to glossary for definitions of terms.

---

[1] Examinations of errors in operational software have shown a large proportion to be the result of incorrect handling of data at extreme values for the field or variable.

In path selection, ATCG will select a path from the beginning of the program up to a selected area which has either been specified by the user or has been chosen by the system. In the case where the system selects the area, it will do this so as to maximize collateral testing (collateral testing is the act of testing other untested areas of the program in addition to the target area). Test data which will cause program execution to follow the selected path will then be generated. For further details on path selection, see Section 2.5.2.

In negating previous constraints, ATCG will generate new test data directly on the basis of the path constraints of the previously completed test cases in such a way as to force the new data to define a new path. The path will be determined by executing the program using the new data as input. Since the path will be used in subsequent test data generation, it will be necessary for an execution monitor to provide information on the path followed. For further details on negating previous constraints, see Section 1.4 and Section 2.5.3. Also see reference number 30.

The boundary condition analysis will involve generation of test data directly from a previous test case that will be closely related to the starting test case. Each test case generated by ATCG may be subjected to a boundary condition analysis to provide added confidence in the program's correct operation. For further explanation, see Section 2.5.4 and also reference number 30.

For retesting a program which has been modified following its previous testing by ATCG, the retesting analysis in ATCG will make maximum use of the previous test data sets to generate initial test cases for the modified code. In particular, some of the new test data will correspond to modifications of the original test data whereby values for each new additional variable (if any) in the modified program will be added to the list of previous input values.

Figures 1 and 2 show the data flow and the major processing steps in ATCG.

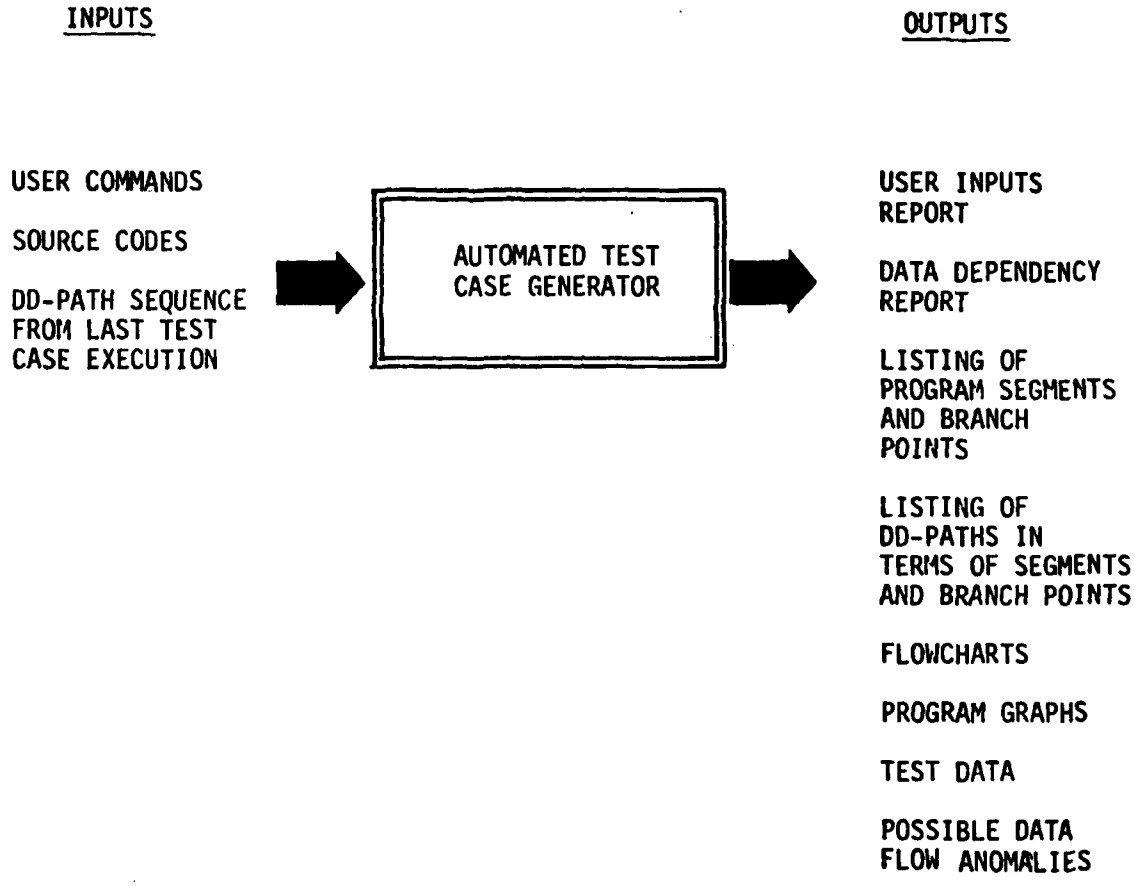


Figure 1. Data Flow for the Automated Test Case Generator
(see glossary for definitions of terms)

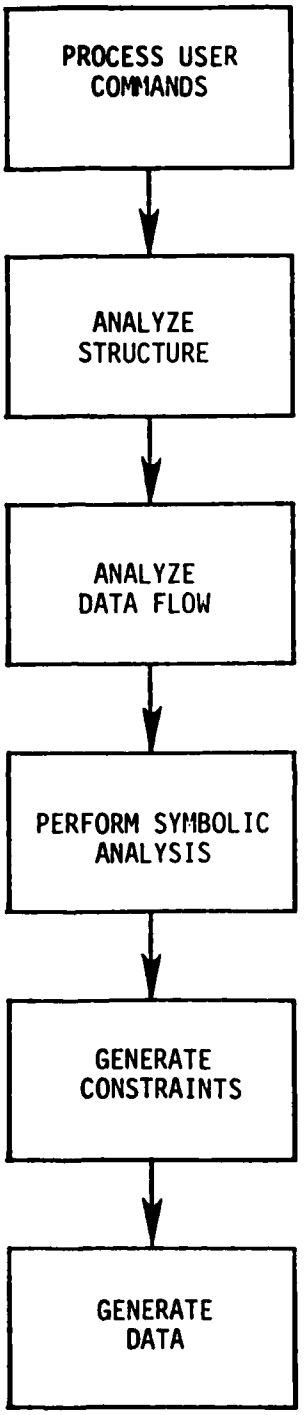


Figure 2. Major Processing Steps in the Automated Test Case Generator

1.3 Producing Flowcharts

ATCG will produce a graphic flowchart for each user specified module. The flowchart will identify each linear segment and branch point with its identifying number and show the control transfer relationship. For each node in the flowchart which corresponds to a segment, the starting and the ending source line numbers of that segment will be indicated. Similar information will be provided for each branch node.

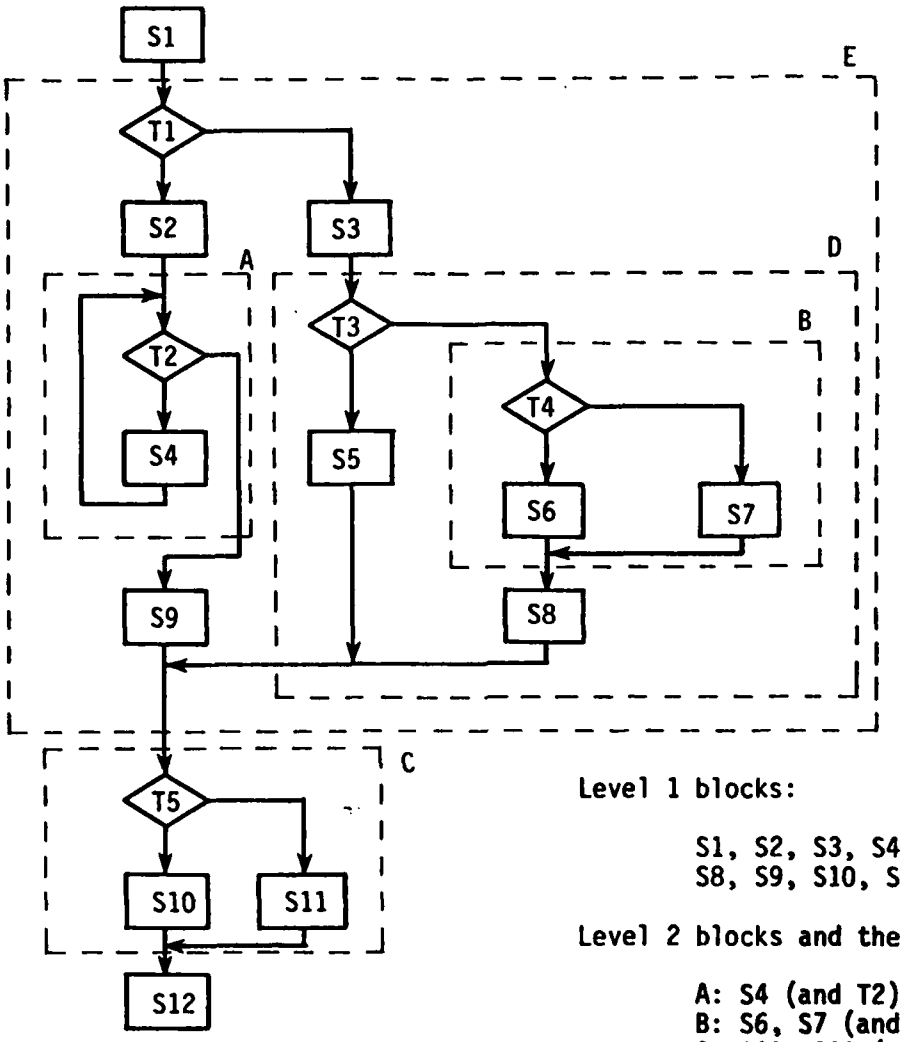
Options will be provided to generate flowcharts which represent the program structure in terms of higher level blocks, including the option of specifying the deepest level of a block in the desired flowchart or the level of expansion. Those blocks which are of lower level and are contained in a larger block of the specified level will not appear explicitly in the generated flowchart. In particular, if the program has the maximum block level M and L equals the level of expansion, then the resulting flowchart shall have maximum level M-L. Figures 3 through 5 illustrate the options of high level flowcharts.

1.4 A New Algorithm for Test Case Selection

A new algorithm (negating previous constraints) for test case selection was developed by Dr. Kundu. This will be incorporated in ATCG as one of the means for test case selection.

There are two basic steps in this method. Assume, to begin with, that the first test case is known a priori, this test case may be constructed by simply assigning arbitrary values to each of the input variables of the program. Each successive test case is obtained by applying the two following steps in the given order.

> Step 1 (Analysis of the last test case). Execute the program with input equal to the most recently generated test case, and determine its execution path. Then perform a partial symbolic execution of that path to determine an approximation to the path-condition of that test case.



Level 1 blocks:

S1, S2, S3, S4, S5, S6, S7
S8, S9, S10, S11, S12;

Level 2 blocks and their components:

A: S4 (and T2);
B: S6, S7 (and T4);
C: S10, S11 (and T5);

Level 3 blocks and their components:

D: S5, B, S8 (and T3);

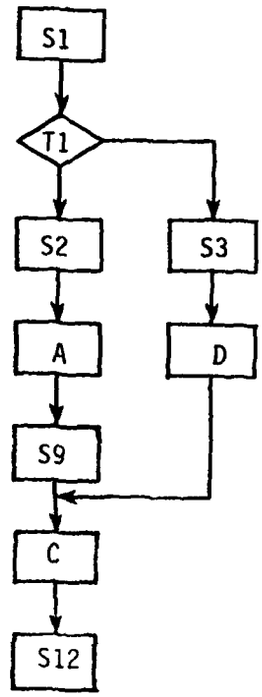
Level 4 blocks and their components:

E: S2, A, S9, S3, D (and T1);

Level 5 blocks and their components:

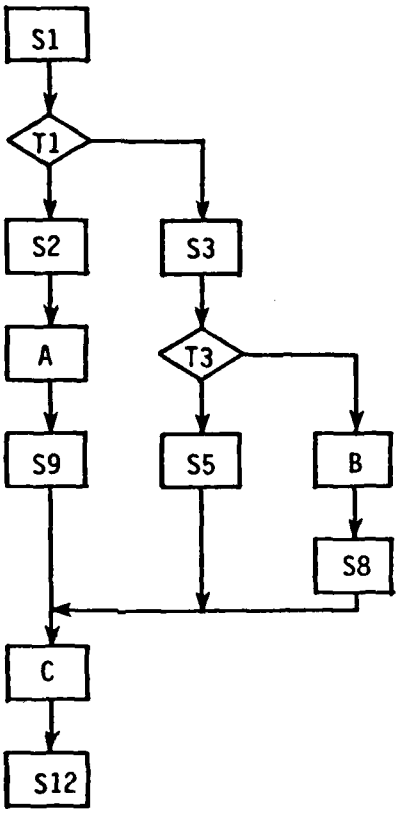
F: S1, E, C, S12;

Figure 3. A Detailed Flowchart and Its Block Structure



(All blocks of level > 3 are expanded.)

Figure 4. A High Level Form of the Flowchart with Expansion Level 3



(All blocks of level > 2 are expanded. This form immediately shows the blocks of level 2, namely, blocks A, B, and C.)

Figure 5. A High Level Form of the Flowchart with Expansion Level 2

Step 2 (Selection of the next test case). Determine the test data for the next test case such that it violates at least one constraint in each of the path-conditions generated so far.

The path is determined from the test data. A specific path is not used for finding input data that corresponds to that path. Also, previously, we were selecting both the test paths and the test data, one within each of the respective path domains. But now only the test data are being selected. This gives us an extra degree of freedom. Each test case generated by this method thus represents a distinct input class (that is, corresponds to a distinct path) because of the requirements in Step 2.

For further explanation of this method, it is recommended that the reader refer to reference number 30.

1.5 Test Data Generation

A machine is restricted, in the final analysis, to making simplistic comparisons (>, <, =, >=, <=, <>) of bit strings of fixed, finite length (such as the length of one word of primary memory). Since bit strings may be regarded as binary integers, it is possible to reduce much of the problem of general inequality solving to that of solving linear inequalities over the set of integers. Thus a linear programming method can be used in many cases for data generation. In particular, this method can easily be applied to the generation of logical and character data. For example, the expression (AA or (NOT BB)) is replaced by ((AA EQ 1) or (BB EQ 0)). Character variables are treated as integer variables.

For those cases where this model cannot be applied, there is no general solution and this is a topic of on-going research. However, much progress has been made in recent years with non-linear programming models. A further discussion of these models is provided in reference number 42. Software packages are available for performing this analysis, and some of these will be incorporated in ATCG.

1.6 Feasibility Analysis

The system as specified in the System/Subsystem Specification of ATCG is primarily intended as a batch system, though it could be run online. Being run online, the system could provide the user with an opportunity to supply additional information. It would do this if the data generator runs into trouble instead of simply requesting the constraint generator to try another test case which meets the user's criteria.

It is estimated that ATCG will contain 60,000 to 80,000 lines of higher-order language code. This estimate is based on the size of existing programs which perform similar functions to those proposed for ATCG (e.g., the program segmentation module in ATCG will be about the same size as the program segmentation module in AMPIC). For a breakdown of the estimates, see the Function Description of ATCG. ATCG can be developed so that memory resident components contain not more than the object code equivalent of 10,000 to 20,000 lines of higher order language (HOL) code. If FORTRAN is assumed, with four words per line of code, then the memory requirements are estimated 80K words [1]. Memory requirements for intermediate files (excluding the primary data base) will be of the order of 200K words.

It is further estimated that the data base will be on the order of ten times the size of the program being tested and that intermediate files used by ATCG will also be on the order of ten times the size of the program. These estimates are based on reference number 23. The minimal memory buffer requirements are 2000 words. The primary data base must be stored on a random access storage median such as a disk.

---

[1] For the purposes of this report, the letter "K" represents the numeral 1024. A word is defined as 4 bytes. A byte is defined as 8 bits. These definitions are assumptions for sizing purposes and would possibly have to be translated for some computer choices.

Expected execution time will be approximately 20-30 minutes CPU time for a typical job (approximately 500 lines of HOL code, 50 variables, 50 inequalities) on a computer with a 1 MIPS (millions of instructions per second) capability.

2. MANUAL ANALYSIS OF AN EXAMPLE

In this section, we will manually carry out the proposed analysis on the program shown in Figure 6.

2.1 Process User Commands

While the specific formats of the user interface language (in the JAVS specific version, the user commands to ATCG would be processed by JAVS) have not been specified, the information which ATCG would need is specified the System/Substyem Specification in Section 4.2.1.A. A possible set of user commands for program SAMPLE might be the following:

```
INIT:   SAMPLE
FLOWCHART:   MAIN, SUB
GRAPH:   MAIN, SUB
LISTSEGMENTS:   SUB
MUSTTEST:   SUB, DD3
```

The first line specifies that this is the first submission of SAMPLE to ATCG and therefore the first test case generation session. Thus it is necessary for ATCG to initialize a data base for SAMPLE.

The second and third lines specify that the user desires both flowcharts and program graphs of MAIN and SUB.

The fourth line specifies that the user desires a segment listing of SUB.

The last line specifies that the user desires that the test case generated test DD-path 3 in SUB. This implies that the user desires that path selection be used as the means of test case selection.

```
1   START$
2   "PROGRAM SAMPLE"
3   ITEM FL F$
4   ITEM KK I 36 S$
5   FILE READ V(NORM) V(EOF)$
6   IN(1,READ)$
7   FILE PRINT V(OK)$
8   OUT(1,PRINT)$
9   I022(0,READ,0,FL,1)$
10  IF FL GT 7.$
11    GOTO LAST$
12  SUB(FL = KK)
13  I023(3,PRINT,0,KK,1)$

14  LAST.
15  IN(2,READ)$
16  OUT(2,PRINT)$

17  PROC SUB(INPUT1 = OUTPUT1)$
18  ITEM INPUT1 F$
19  ITEM OUTPUT1 I 36.S$
20  BEGIN "SUB"
21  IF INPUT1 LS 3.$
22    OUTPUT1 = 1$
23  IF INPUT1 GQ 3.$
24    OUTPUT1 = 0$
25  RETURN$
26  END "SUB"

27  TERM$
```

Figure 6: Program SAMPLE

The Process User Commands function will extract this information from the user commands and encode it into a user control vector. The user control vector will then direct ATCG's subsequent functions.

2.2 Analyze Structure

This function shall analyze the structure of the source program, dividing it into linear segments and branch points. It shall then organize these into DD-paths. It shall also generate flowcharts and program graphs at the user's direction. This function shall be performed only during the first test case generation session, after program changes, or at the user's direction. A detailed description of how each of these functions of structural analysis would be applied to SAMPLE is given below.

2.2.1 Segment Program

This subfunction shall partition the source code into linear segments and branch statements. Each segment and branch point shall be assigned a unique identifying number within each module. In order to reference any given segment or branch point, it will be necessary to give the module name, whether it is a segment or a branch point (segments are designated S elements and branch points are designated T elements), and the number of that element. This information is then stored in the linear segments and branch point table. It may also be printed (this is called a segment listing) for any modules which the user desires. The segment listing which will be printed for MAIN and SUB are shown in Figure 7.

2.2.2 Form DD-paths

This subfunction shall identify each DD-path in the source program, starting with the results of program segmentation. The test itself will be referred to as T followed by the test number. The two possible resolutions of that test, true and false, will be refered to as 'R' and 'P' followed by the

SEGMENT LISTING OF MODULE MAIN IN SAMPLE

```
S1:   IN (1, READ)
T1:   IF FL GT 7.  DO S2
S2:   GOTO S4
S3:   SUB (FL = KK)
S4:   IN (2, READ)
      OUT (2, PRINT)
```

SEGMENT LISTING OF MODULE SUB IN SAMPLE

```
S1:   PROC SUB (INPUT1 = OUTPUT1)
      BEGIN
T1:   IF INPUT1 LS 3.  DO S2
S2:   OUTPUT1 = 1
T2:   IF INPUT1 GQ 3.  DO S3
S3:   OUTPUT1 = 0
S4:   RETURN
      END
```

Figure 7: Segment Listings

test number. Thus 'R1' would indicate that test number one for this module was traversed and that the true branch of this test was followed. 'P1' would indicate that the false branch of that test was followed. The reason for this is that DD-paths are defined as being overlapping--one DD-path will contain the test itself as its last element, while the two following DD-paths will contain the resolutions of that test as their first elements.

A table (the DD-paths table) giving the sequence of identifying numbers of the segments which comprise each DD-Path shall be generated. It may also be printed (referred to as DD-path listings) for any modules which the user desires. The DD-path listings which would be produced for MAIN and SUB are shown in Figure 8.

2.2.3 Produce Flowcharts

This function shall produce a flowchart for each user specified module The flowchart shall identify each linear segment and branch point with its identifying number as determined in the Segment Program function and show the control transfer relationship. The flowcharts that will be produced for MAIN and SUB are shown in Figure 9.

2.2.4 Graph Program Using DD-paths

This subfunction shall generate a representation of the control transfer relationships among the DD-paths for each user specified module. Each DD-path shall be represented as an arc with a label indicating the DD-path number. The control transfer points shall be shown as nodes. The program graphs that will be produced for MAIN and SUB are shown in Figure 10.

2.3 Analyze Data Flow

The analysis of the flow of data in the program shall consist of two parts. The first part shall analyze the source program text to determine whether

DD-PATH LISTING OF MODULE MAIN IN SAMPLE

DD1: S1. T1

DD2: R1. S2. S4

DD3: P1 .S3. S4

DD-PATH LISTING OF MODULE SUB IN SAMPLE

DD1: S1. T1

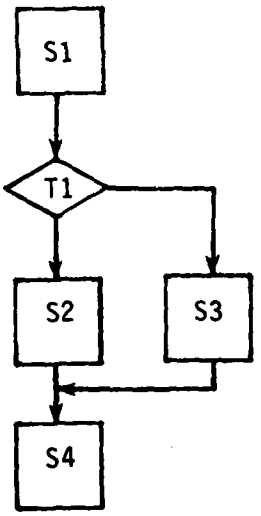
DD2: R1. S2, T2

DD3: P1. T2

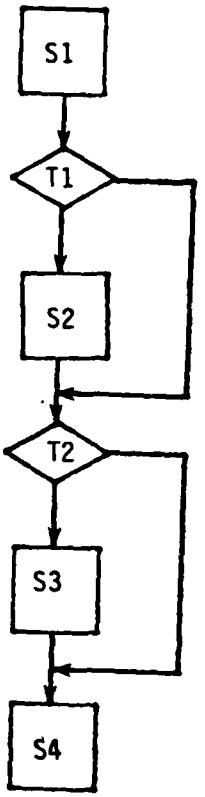
DD4: R2. S3. S4

DD5: P2. S4

Figure 8: DD-path Listings

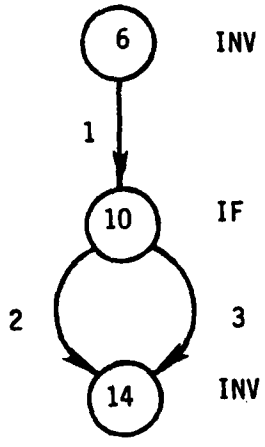


FLOWCHART OF
MAIN
IN SAMPLE

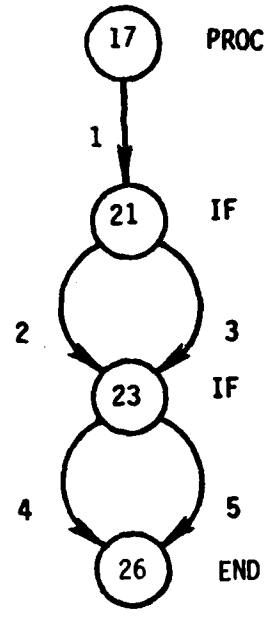


FLOWCHART OF
SUB
IN SAMPLE

Figure 9. Flowcharts



PROGRAM GRAPH
OF MAIN
IN SAMPLE



PROGRAM GRAPH
OF SUB
IN SAMPLE

Figure 10. Program Graphs

or not either of the following data flow rules have been violated:

(a) A reference to a variable should be preceded by an assignment to that variable without an intervening undefinition (see glossary).

(b) A definition of a variable should be followed by a reference to that variable prior to a redefinition or an undefinition.

This analysis shall be performed first for the program segments. Then the results shall be combined together to complete the analysis for the whole program. A report indicating any violation of the rules (a) and (b) above shall be generated. In the case of our program SAMPLE, no such report would be produced. For further explananation see reference number 37.

The second part shall generate a table for each DD-path giving, for each variable, the list of variables which may affect the value of that variable This analysis shall be performed first for the program segments, and then combined for each DD-path separately. The analysis shall identify, for each DD-path, the variables that can potentially create "ambiguity" during symbolic execution. Thus across DD-path 1 in MAIN, no variable depends on any other variable. The same is true in MAIN across DD-path 2. Across DD-path 3 in MAIN, KK depends on FL. For further explanation, see reference number 29.

2.4 Perform Symbolic Analysis

This function shall prepare tables for each DD-path describing in algebraic terms the effect of traversing that path. This function is performed first for the program segments and then the results are combined for each DD-path separately. For DD-path 1 in MAIN, the table will contain an entry noting that FL had been input along that path. For DD-path 2 in MAIN, the table will contain an entry noting that there had been no symbolic changes across

that path. For DD-path 3 in MAIN, the table will contain an entry noting that KK had been assigned a functional (SUB) value of FL. For DD-path 2 in SUB, the table will contain an entry noting that OUTPUT1 had been assigned the value 1 across that path. The other entries in the tables will be similar to these.

This function shall only be performed during the first test session or after program changes. Potential symbolic execution ambiguities shall be identified and flagged at this time. In the case of program SAMPLE, nothing is flagged because SAMPLE contains no potential symbolic ambiguities.

2.5 Generate Constraints

This function shall generate constraints which shall have to be satisfied by the data for the test case being generated. If this is not the first test case generation session, this function shall update the data base prior to selecting new constraints. Such updating shall be accomplished by symbolically traversing the path followed during execution of the last test case that was generated. Following that, new constraints will be generated using one of the following user selected methods:

(1) Path Selection: Constraints will be generated for data that will cause execution of a specific path which crosses a selected portion of code. In this, the user has the option of specifing all or part of the path to be followed or allowing the system to select it. See Section. 2.5.2.

(2) Negation of Previous Constraints: Constraints shall be generated for data which will cause execution of a new (as of yet, unknown) path. See Section 2.5.3.

(3) Special Value Constraints: Constraints shall be generated for data near boundry conditions for selected variables across a previously tested path. See Section 2.5.4.

A detailed descripton of what each of these subfunctions will mean for SAMPLE is given below.

2.5.1. Traverse Last Path

This function shall perform symbolic execution of the path followed by the program execution of the last test case generated by ATCG. As a result, this function will derive the full set of constraints which were true of the last test case. This function will alsc derive a list of tested segments and branch points. This function shall not be performed during the first test case generation.

Suppose that the DD-path sequence followed by our last test case was MAIN-DD1, MAIN-DD3, SUB-DD1, SUB-DD2, SUB-DD5, MAIN-DD3. This corresponds to the following segment and branch point sequence:

MAIN-S1. MAIN-P1. MAIN-S3.
SUB-S1. SUB-R1. SUB-S2. SUB-P2. SUB-S4.
MAIN-S3. MAIN-S4.

Thus our full set of constraints for this test case will be the following:

((NOT (FL GT 7.)) AND
(FL LS 3.) AND
(NOT (FL GQ 3.)))

This will be added to the list of previous constraints on previous test cases in the constraints table. If there were no previous constraints, the constraints table will be created at this point and this will become its first entry.

Our list of tested segments and branch points will be:

MAIN-S1
MAIN-P1
MAIN-S3
MAIN-S4

SUB-S1
SUB-R1
SUB-S2
SUB-P2
SUB-S4

Each of these will be checked off in the hit table if this is not the second test case session. If it is the second test case session, the hit table will first be created and then they will be checked off.

2.5.2 Select Path

This function shall generate constraints for data that shall cause execution of a specific program path. This path will cross the area of the program which has been designated as the section of code to be tested next. A detailed description of what each of the subfunctions will mean for SAMPLE is given below.

2.5.2.1 Select Segment: This function shall choose the next target segment for testing. This will be done on the basis of one of two criteria:

(1) user has specified the segment
(2) otherwise the system will choose the most deeply nested untested segment

If the user has specified the segment, then the identifying symbols of that segment are the output of this function.

For a more interesting example, let us say that the user has specified that the system will select the target segment. Let us also say that

the test case discussed as the last test in the previous section is the only test case run so far. This function will then derive a list of untested program segments and branch points from the systems list of tested segments and branch points. The list will consist of the following:

MAIN-R1
MAIN-S2
SUB-P1
SUB-R2
SUB-S3

Of these, all of the ones in SUB are more deeply nested than MAIN since SUB is a called subroutine and MAIN is not. When examining the untested segments and branch points in SUB, S3 will be chosen because both of the other candidates are at level 0 while S3 is at level 1. Had this not eliminated all but one segment, then the later of the two in the text would have been chosen. Thus, in this case, SUB-S3 is our selected segment.

2.5.2.2 Select Reaching Set: This function shall choose a DD-path sequence starting at the head of the program and ending at the selected segment, including any portion of the path which the user has chosen to specify after the first segment in the target area. The system shall begin by choosing a calling sequence beginning at the main routine and ending at the routine containing the selected text. The system shall then choose paths from the beginning of each module in the calling sequence to call to the next module in the sequence (in the lowest module, of course, we choose a path from the beginning of the program to the first segment in the target area). If parallel processing is available, these paths may be chosen in a simultaneous manner initially. In choosing all paths, the system shall choose the path along the program graph working forward from the beginning of the routine and backward from the selected call or target area. This will significantly reduce the search space in the tree. Backtracking shall be done along these paths breaking links (and forming new ones) from the middle of the path towards its extremities. When the system is presented with a

choice between an untried branch and an already tested branch, it shall initially select the untried branch. When so instructed by the user, the system shall either select loop branches to be executed a minimum number of times or as close to a user-specified maximum number times as is possible and still maintain a consistent set of constraints. As the path traverser encounters inconsistencies, it shall call on this function to backtrack to the inconsistency and patch the path if possible.

In our case, we have been passed SUB-S3 by the Select Segment function. There is only one way to reach the call to SUB in the MAIN routine, namely Main-DD1 followed by MAIN-DD3. Thus we select that path through MAIN the first time through.

In SUB, we select SUB-S1. Then, since SUB-R1 has been tried, we select SUB-P1. After that, since SUB-P2 has been tried, we select SUB-R2. Next is SUB-S3. Thus our complete selected path is:

MAIN-S1. MAIN-P1. MAIN-S3.
SUB-S1. SUB-P1. SUB-R2. SUB-S3.

2.5.2.3. Traverse Selected Path: This function shall symbolically execute the selected path using the forward substitution method. The forward substitution method will be used because it allows early detection of infeasible paths with contradicting input constraints; further, it has advantages over backward substitution in handling arrays. For a discussion of this point, see Sections III and IV.(ref. no. 40). The constraints traversed shall be added to the pool one at a time. As each constraint is added, it shall be checked to see if it conflicts with any of the existing constraints. If it is not consistent, then the path traverser shall request that the reaching set selector form a new path from where the inconsistency was encountered to the target area.

In our case, the path we have been passed is consistent and the set of constraints that we generate is the following:

((NOT (FL GT 7.)) AND
(NOT (FL LS 3.)) AND
(FL GQ 3.))

2.5.3 Negate Previous Constaints

A detailed description of what each of the subfunctions of Negate Previous Constraints means for SAMPLE is given below. For further explanation, see reference number 30.

2.5.3.1 Select Target Constraints: This function shall select one constraint from the constraint set of each of the previous test cases. If no previous test cases are available, the target constraint set is empty. It that case the first test data generation step shall assign randomly selected values to the input variables.

Let us say that the test case described in Section 2.5.1 is the only test case in the data base. Since there is only one set of constraints in the data base, it is sufficient for us to violate only one constraint. We choose the first constraint of the test case available. Had there been n test cases, we would have selected the first constraint from the first case, the first constraint from the second that was different from the constraint we had already selected which was not inconsistent with the constraints selected so far, and so on until we had n constraints (one from each test case). For further explaination of this method, see reference number 30. In this case, our output will be the single constraint (NOT (FL GT 7.)).

2.5.3.2 Generate Data Constraints: This function shall combine the logical negation of the selected target constraints to form the constraints which will have to be satisfied by the new test data in order for it to cause the program execution to follow a path (not yet known) which is different

from all previous test paths. For further explanation of this method see reference number 30. .

In this case, we will negate (NOT (FL GT 7.)) giving us (FL GT 7.) as our output.

2.5.4 Analyze Boundary Conditions

This function shall assist the user in doing a detailed comparative study of the program's input-output behavior on a set of closely related or "neighboring" test cases. This analysis shall be performed as a user-selected option on each test data with respect to one or more variables that the user has selected for this purpose and with respect to critical areas of the source which the user has also identified for this purpose. ATCG shall then examine the branching expressions which involve the selected variable in the critical area and select a new value for the variable near one of the boundaries. All other input values for that set of test data so generated shall be the same as those for the test case in question.

Let us say that the test case described in Section 2.5.1 has been selected for this analysis. Furthermore, let us say that the selected variable is FL (in this case, any other selected variable would result in an error message since FL is the only one involved in branching constraints) and that the critical area selected is MAIN-DD2. Thus the boundary condition which we wish to be near is (FL LS 7.)

We do not wish to violate this condition, but merely come close to doing so. Thus the constraint which we wish to generate is FL EQ (7. - E) where E is a postive number and small with respect to 7. E will be calculated as a percentage of 7. The percentage which will be used here will be decided on at the time of implementation (perhaps the user may be allowed to specify it). Let us say here that that percentage is .1%. Thus the constraint which we generate is FL EQ 6.993.

The conditions which will be output will be all the constraints up to the critical area with the new condition substituted for the old one. (These will be checked for consistency before being output.) For further explanation, see reference number 30.

In this case, we will output the single condition FL EQ 6.993.

2.6 Generate Data

This function shall obtain a solution to the data constraints, by employing linear/integer programming methods, numerical methods for constrained and unconstrained optimizations, and other mathematical tools. It shall be capable of generating real, integer, boolean, and character data.

Let us examine the constraints generated by each of the previous modes of test case generation.

First recall that our constraints from Select Path were:

```
((NOT (FL GT 7.))  AND
(NOT (FL LS 3.))  AND
(FL GQ 3.))
```

We begin by combining the constraints on each variable into sets. In this case, this is already done. Next we simplify these constraints giving us:

```
((FL LQ 7.)  AND
(FL GQ 3.)  AND
(FL GQ 3.))
```

We now note that the third constraint is redundant and remove it. This gives us:

((FL LQ 7.) AND (FL GQ 3.))

or algebraically,

7. $\geq$ FL $\geq$ 3. or
4. $\geq$ FL - 3. $\geq$0.

In this case, we can use a standard random number generator to generate a number between 0. and 1. We then multiply that by 4. We then add 3. to that and we have a value of FL which satisfies the constraints.

Next, consider the constraints which we generated in negating previous constraints. Recall that our output from that step was (FL GT 7.). Here again we may use a random number generator to generate a value of FL which satisfies the constraints.

Lastly, consider the constraints which we generated in boundary condition analysis. Recall that our output was (FL EQ 6.993) Thus, the selection of 6.993 as FL is done trivially.

2.7 Subsequent Steps

This completes the manual analysis of the example program, SAMPLE. In the practical application of the ATCG system, the next step will be computer execution of the program, SAMPLE, using as input the data generated in accordance with Section 2.6. Computer execution using this data will accomplish the objective specified by the user. Execution of the program will be carried out under the control of an execution monitor, such as JAVS. The execution monitor will record in a file the order and number of times which DD-paths are crossed. This file will provide input for the Generate Constraints function (see Section 2.5) when the data base is updated during the next test case generation session.

3. SURVEY OF THE STATE OF THE ART OF PROGRAM TESTING

3.1 Prelude to Testing

Thoughts about program testing should not be put off until the program is completed--they should begin with the program in the design phase and should be kept in mind throughout implementation. The ideas presented in this section will help to clarify classes of inputs to a program--and will allow the user to select modules which will need retesting after changes.

3.1.1 Hierachical Design

Hierarchical or top down design is the process of designing the top level (or main features) of a task first and defining later how those subtasks will be accomplished in terms of other smaller tasks.

This design of program structure is strongly recommended as it helps clarify classes of inputs to the program. Tools currently exist to assist in this type of design.

One such tool is Logicflow. This system was developed by Logicon under contract number F04704-76-C-0001 for SAMSO/MNNC. The Logicflow language helps a designer organize his ideas in a hierarchical fashion. It allows him to automatically generate flowcharts at any desired level of detail from text input. It also eliminates the need of redrawing flowcharts by hand every time a minor change is made, thus allowing him to always have up to date information about the design.

3.1.2. Structured Programming

Structured programming is the analog in the implementation phase to hierarchical design in the design phase. It keeps the most important--or primary tasks--of a program in the top level modules (procedures or

routines) and the smaller tasks that comprise those tasks at a much lower level. In this respect, it is not very different from the management of well organized human systems.

This technique helps the programmer to grasp what he is doing at any given level of detail. Tree structured module dependency is also invaluable when deciding which routines should be retested after changes have been made to a system.

3.1.3. Acceptance Criteria

One should decide between the many forms of acceptance criteria before any actual testing begins. The major choices are the following:

1) Correct execution of an arbitrary number of statements (e.g., 85 per cent).
2) Correct execution of every statement in the program at least once.
3) Correct execution of an arbitrary number of control branches (or decision-to-decision paths) (e.g., 90 per cent).
4) Correct execution of every control branch in the program or system at least once, but not in every possible combination.
5) Correct execution of every control branch in the program or system in every possible combination at least once.

While (5) is obviously the strongest of the acceptance criteria, it is not in general possible for even small programs. The best that can generally be hoped for is (4)--though for an extremely large system, this might not even be feasible.

3.1.4 Test Beds

Before the analysis covered in Section 3.2 can begin, it may be necessary to divide a large system into smaller subsystems. A test bed for a subsystem consists of all the modules to be tested at this level plus "stubs" (dummy modules) for any routines which are referenced by this system but not contained in it. It also includes input data--either live or generated--for the subsystem.

3.2 Data Flow Analysis

Data flows analysis is a way of examining how information flows through a program. It searches particularly for two kinds of mistakes. These are normally termed "type I" and "type II" data flow anomalies.

3.2.1 Type I Anomalies

Type I anomalies are constructions such that a reference to a variable is not preceeded at some point in the module by a definition of that variable. This is a fatal error on many systems. Even on systems which allow the user to specify a preset value of all storage locations, this is a bad programming practice as it can lead to unpredictable effects if the software is moved to another system.

3.2.2 Type II Anomalies

Type II anomalies are constructions such that a definition of a variable is not followed by a reference prior to another definition of that variable or prior to passing outside the scope of that variable. While this in itself is not a fatal error, it is normally an indication of another error--such as a mistyped variable name or a variable which is no longer used by the program but which has not been removed.

3.2.3 DAVE

DAVE is an automated data flow analysis tool. It was developed at the University of Colorado for use on ANSII standard FORTRAN IV programs. It performs an exhaustive search for data flow anomalies in a static fashion (without execution of the program). The time that is takes to do this is linearly proportional to the product of the number of edges in the program graph and the number of variables in the program.

3.3 Structural Analysis

Structural analysis can provide us with flowcharts, program graphs, and other information about the system.

3.3.1 Restructuring Programs

No matter how careful a programmer might (or might not) be, there are usually some violations of structured programming rules in his code. Since structural analysis is most easily performed on well structured code, it would be useful if the code could be converted to a structured form before attempting such an analysis. Fortunately, tools to do this are already in existence. They are called "Restructuring Programs"

One such tool is contained within AMPIC. AMPIC was developed by Logicon as an in house testing and debugging tool and is primarily applied to FORTRAN and LITTON ASSEMBLY language programs currently, but could be expanded to handle JOVIAL code. For further details on AMPIC, see Appendix C.

3.3.2 Code Instrumentation

Instrumentation is the process of placing software counters or probes in each control branch in the program to perform such monitoring as path execution and tracing, timing analysis, etc. This allows us to see if a

given path was executed after a program run. (This is a facility currently supported by JAVS).

3.3.3 Flowcharts

While it is true that Logicflow aids the designer in drawing flowcharts of a system before its implementation, there is never any guarantee that these were followed during the implementation. And even if they were followed, they are undoubtably no longer an accurate representation of the existing system. Fortunately, there are systems for producing flow charts from existing code. In this way, we may see what the control flow structure at any given level really is.

One such system for producing flowcharts is contained in AMPIC. Given source code for programs written in several LITTON ASSEMBLY languages or FORTRAN, it can produce a flowchart of that source text. The techniques used in AMPIC could also be applied to JOVIAL. For further details on AMPIC, see Appendix C.

3.3.4 Program Graphs

A program graph is a graphic representation of the control structure of a program or module. A program graph differs from a flow chart in that a program graph contains only structural information and carries no information about the statements themselves. From this, one may deduce a minimal covering set for the program.

A minimal covering set for a program is the smallest set of paths (from the beginning of a program to one of its exit points) which collectively covers all the edges (or branches) of the program graph.

3.4 Symbolic Analysis

Symbolic analysis is the process of analysing how the symbols (variables) in a program are related to one another in algebraic fashion.

3.4.1 Program Proofs

Program proving is a validation technique which considers a program to be a mathematical theorem and, on the basis of certain assumptions about the state of the input variables, attempts to prove that theorem, with techniques from algebra. This form of analysis is most often carried out by hand. But even for modest sized programs, this can become extrememly complex.

3.4.2 DPEN

DPEN is an algorithm used to perform a simple--but powerful--form of symbolic analysis. Given the source text of a program, it will produce a list of all the output variables of the program and the input variables that each variable depends on (either as a result of an assignment statement or the path condition). As an option, the user may specify that a restricted form of the DPEN function, RDPEN, will be used instead of DPEN.

This restricted form of DPEN produces a list of the output variables and the input variables that they depend on as a result of assignment statements. See reference number 29 for a more detailed description.

3.4.3 Symbolic Execution

Symbolic execution might be thought of as an automated aid to program proofs. With help from the user to make decisions at branch points (two types of branching where user interaction is required will be discussed in a moment), the system accumulates algebraic expressions for all variables and path conditions in terms of the (symbolic) values of the input variables. The following paragraphs describe issues involved in symbolic execution.

3.4.3.1 Binding: An issue of interest to us with regard to symbolic evaluation is the binding of symbolic variables to their values. This deals with storage of symbolic variables. There are two basic approaches:

shallow binding and deep binding. Shallow binding means that only the most recent symbolic values for variables are maintained. This scheme is used in ATTEST (a test case/test data generator reported by Clarke for FORTRAN programs). Deep binding means that all symbolic values for variables are kept along with information describing the area(s) of the program in which those values are valid. This is the scheme used in DISSECT (a symbolic execution system developed by Howden for FORTRAN programs). The advantage of shallow binding is that it consumes less memory and is faster if only one program path is to be considered at a time. The advantage of deep binding is that previous program states are more easily restored if backtracking becomes necessary or if more than one potential path is being considered at a time. Both SRI SELECT (a test case/test data generator and symbolic execution system due to Boyer et al for programs written in a subset of LISP) and EFFIGY (a symbolic execution system reported by King for programs written in a subset of PL/I) use hybrid schemes which maintain a certain amount of deep binding information. That is, after the variable-value pair becomes "old" (several nodes back in the execution tree) it is eliminated.

3.4.3.2 The Array Variable: The array variable problem occurs for programs with subscripted variables whose subscripts are themselves variables. Consider a program with the following statements in it:

```
      .
      .
I = 5
J = 6
IF (N.GT.6)  I = 6
      .
      .
      .
X(I) = 3
X(J) = N/5
      .
      .
      .
```

If we were to do a purely symbolic execution of the above program, we would be unable to determine whether X(I) was 3 or N/5 since this is dependent on the input value of N.

3.4.3.3 The Loop Variable Problem: The loop variable problem occurs when loop constructions are encountered in which the number of interations are input variables. Consider the following test-first loop construction:

```
      .
      .
      .
DO WHILE (I<N),
      .
      .
      .
I = I + 1,
      .
      .
      .
```

How do we decide when I is no longer less than the symbolic value of N? If we continue following both branches of a test as we do in pure symbolic execution, the execution tree would become infinite.

3.4.3.4 Dealing with Symbolic Ambiguities: The array variable problem and the loop variable problem are collectively termed symbolic ambiguities. Systems differ with respect to their treatment of symbolic ambiguities. In ATTEST, if a variable is assigned the value of an array reference which has an ambigously defined index, then the variable is marked as having an undefined value. EFFIGY used an exhaustive case analysis. In this approach, a path which contains a reference to an ambiguously defined array reference is split into a collection of paths. Each path in the collection is associated with a possible choice of the array element. The SELECT system contains a sophisticated implementation

of this idea. In DISSECT, if a variable is assigned an ambiguous array reference, it is flagged as such. If the value of the variable is referenced during the output phase of the symbolic evaluation, then the deep binding value stack is used to construct a representation of the different possible values.

In FACES, (a test case/test data generation and symbolic execution system developed by Ramamoorthy), the user is allowed to select which variables he would like to be symbolic. All the others will have test data generated for them. If one of the variables designated as a symbolic (sometimes called "primary") variable turns out to be a loop parameter or a subscript, after that point in that program, it will be considered to be a non-symbolic variable and will have a value assigned to it accordingly. At the end of this type of an execution, the user will then have a list of the non-symbolic variables (and the values generated for those variables). In addition he will have a list of the symbolic variables in terms of the non-symbolic variables and the other symbolic variables.

In AMPIC, the user interactively defines certain variables as being symbolic and supplies values for the other variables. AMPIC also allows the user to make decisions in the program at ambiguous decision points. It further allows the user to make certain assertions about the values of the symbolic variables (e.g., "ASSERT N<S", where N is symbolic). AMPIC is described further in Appendix C.

3.4.3.5 Consistency Checking: All of the symbolic evaluation systems which have been constructed use a similar approach to building path-conditions (see glossary). Each time a branch is traversed, the condition associated with that branch is evaluated and added to a list of branch conditions. Most systems contain facilities for checking the consistency of or solving systems of predicates. ATTEST and SRI SELECT contain linear programming packages which can be used to solve linear systems of constraints. DISSECT contains facilities for checking the consistency of a system but does not contain facilities for

generating a solution.

3.5 Test Case Selection

There are three basic ways of selecting test cases. They all assume that the first test case exists a priori; this may be constructed by simply assigning arbitrary values to each of the input variables of the program. The first selects new data from a path, the second selects a path from new data, the third selects new data which are close to the boundary conditions of the previous test case.

3.5.1 New Data from a Path

In this method, we first select a path which contains some statements or branches which have not yet been executed. The most deeply nested untested code is a reasonable choice for this since it typically provides the most collateral testing. We then generate data to satisfy this path. (This is generally done either with an inequality solver or through manual analysis by trial and error.)

One of the problems with this selection is that such data might or might not really exist depending on whether or not a randomly selected path is executable.

3.5.2 Path from New Data

There are two basic steps in this method. Each successive test case is obtained by applying the two basic steps in the given order.

> Step 1 (Analysis of the last test case). Execute the program with input equal to the most recently generated test case, and determine its execution path. Then perform a (partial) symbolic execution of that path to determine (an approximation to) the path-condition of that test case.

Step 2 (Selection of the next test case). Determine the test data for the next test case such that it violates at least one constraint in each of the path-conditions generated so far.

We see that the roles of the test path and the corresponding test data have been simply reversed in this method. Here, the path is determined from the test data in order to guide the next test case away from the previous test cases; the path is not used for finding input data that corresponds to that path. Also, previously, we were selecting both the test paths and the test data, one within each of the respective path-domains. But now only the test data are being selected. This gives us an extra degree of freedom. Each test case generated by this method thus represents a distinct input class (that is, corresponds to a distinct path) because of the requirements in Step 2.

For further explanation of this method, it is recommended that the reader refer to reference number 30.

3.5.3 Analyzing Boundary Conditions

In this method, we first select a previous test case which we want to examine more closely. We then choose a critical area of the program during that test case. Next we identify which variables and branching conditions are to be studied more closely. Then, we modify the input data which influence those variables so that when this point in the execution path is reached again, the selected variables will have values which are very close to the boundary defined by the selected branching condition. (e.g., If X>7 were the branching condition, 7 would be the boundary defined for X by that branching condition.)

3.6 Closing Remarks

Hierarchical design and structured programming help insure that the classes of inputs to a program will be well defined--but even more importantly, they insure that we will know which modules do and which don't need retesting as a result of changes or corrections to the source code. Data flow analysis insures that the symbolic execution of a program will be meaningful--otherwise, we might be generating values for variables that should not exist (or might be misspelled and therefore misinterpeted). Symbolic execution then provides us with both a functional form of the program (output variables expressed algebraically in terms of the input variables) and the path conditions that we need to generate the next test case. Linear and nonlinear programming methods then allow us to generate data for these constraints. These together constitute the state of the art of program testing.

APPENDIX A
GLOSSARY OF TERMS

Algebraic Language

An algebraic language is an algorithmic language whose statements are structured to resemble the structure of algebraic expression, e.g., ALGOL, FORTRAN.

Block Level

The concept of block level is used to describe level of detail in software description. The most detailed flow chart or software description is defined as level one. Level two is derived by collapsing into a single block a pair of linear code segments and the common preceeding branch point. Level three is derived by applying the above procedure to the level two description. Higher levels are similarly derived. A more formal statement of block level is as follows: A program block is of level n ($\geq$ 1) if it is minimal with respect to the properties that (1) all of its subblocks are of level less than n, and (2) at least one subblock is of level (n-1). A program block of level 1 is the same as a linear segment, and all blocks of a given level are disjoint.

Boundary Condition Analysis

Boundary condition analysis of a piece of software is the study of how that software behaves at or near the limits of its constraints. In ATCG, this would entail choosing a new set of test data from an old (user specified) test case which would come within a user specified epsilon of violating a constraint in a user specified critical area of the software in the context of that test case.

Branch Point

A branch point is a statement which transfers control to another linear segment in the program which need not be the next sequential linear segment.

Debugging

Debugging is the process of isolating the source of an error and finding a solution to the problem.

Decision-to-Decision Path (DD-path)

A decision-to-decision path is sometimes called a program edge or simply an edge, also called a DD-path. It is a sequence of linear segments in the program which may be executed as a result of the evaluation of a predicate (conditional branch) in the program, but prior to a second predicate evaluation.

Edge

See Decision-to-Decision Path.

Execution Monitor

An execution monitor is a program that instruments (inserts source code statements) into a program being tested to enable counting of the number of times each segment is executed by the particular test.

Flowchart

A flowchart is a directed graph representation of the program which shows the linear program segments, the branch statements, and the control flow among these elements.

Incompatible DD-path Sequence

An incompatible DD-path sequence is a sequence of DD-paths which are contiguous (i.e., the terminal point of one edge is the starting point of the next DD-path in the sequence) but not a part of any execution path in the program. An example of this is where the initial DD-path in the

sequence requires, say, X < 0 and the terminal DD-path requires an incompatible condition, say, X > 2, without X being modified in-between or by the initial DD-path. Each incompatible DD-path sequence is loop-free in the sense that the sequence contains no repetitions.

Linear Segment

A linear segment or simply a segment is a sequence of statements in the program such that in every execution of the program either all of them are executed (in the given order) or none of them is executed.

Negating Previous Constraints

Negating previous constraints is the process of selecting a new set of test data in such a way as to violate at least one constraint from each set of constraints which were true of all previous data sets.

Path-Conditon

The path-condition is the set of constraints on the input variables that must be satisfied in order for the program execution to follow the specific path.

Path Selection

Path selection in ATCG is the process of choosing a path from the beginning of the software to the end of a critical area (chosen on the basis of user specified constraints) and then generating the constraints which would have to be satisfied by a set of input data in order to force the execution of that path.

Program Edge

See Decision-to-Decision path.

Program Graph

A program graph is a simplified directed graph form of the flowchart which has a node only corresponding to each DD-path in the flowchart. The control transfers between DD-path (via branch points) are shown as arcs between the corresponding nodes.

Reaching Set

A reaching set is a sequence of DD-paths which originates at the beginning of the program and ends at a specified segment of the program A reaching set may be described alternatively as a sequence of linear segments and branch points.

Segment

See Linear Segment.

Target Area

The target area is the DD-path or sequence of DD-paths which should be the ending sequence of the reaching set generated for a given test case.

Target Segment

The target segment is the linear segment selected by ATCG as the next segment which should logically be tested next. The DD-path which contains this segment will be the target area.

Test Case

A test case is synonymous with test path.

Test Data

A set of test data is a vector of input values.

Test Path

A test path is a sequence of linear program segments which is traversed as the result of a single test execution.

Testing

Testing is the process of certifying that the program meets its requirements. For each requirement specification, one or more sets of inputs is applied to the program and executed. The actions performed by the program are then compared with the required actions.

Undefinition of a Variable

The undefinition of a variable occurs at a point in the source code that is beyond the scope of that variable.

APPENDIX B

BIBLIOGRAPHY

1. Abrahams, P. and Clarke, L.A., "Compile-Time Analysis of Data List-Format List Correspondences," University of Massachusetts Tech. Rep. #78-11, Computer and Information Science Dept.

2. Benson, J.P. et al, "JAVS: JOVIAL Automated Verification System - System Design and Implementation Manual", General Research Corporation, Rept. No. CR-1-782, June 1978.

3. Boehm, B.W., McClean, R.K., and Urfrig, D.B., "Some Experience with Automated Aids to the Design of Large Scale Reliable Software," IEEE Trans. Software Eng., Vol. SE-1, pp. 125-133, 1975.

4. Boyer, R.S., Elspas, B., and Levitt, K.N., "SELECT-A Formal System for Testing and Debugging Programs by Symbolic Execution," in Proc. Int. Conf. Reliable Software, Los Angeles, CA, pp. 234-244, 1975.

5. Brooks, N.B. et al, "JAVS Methodology Report," RADC-TR-77-126, Vol. III, April 1977 , AD# A041 048.

6. Clarke, L.A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Trans. Software Eng., Vol. SE-2, pp. 215-222, 1976.

7. Clarke, L.A., "Automatic Test Data Selection Techniques," Infotech State of the Art Conf. on Software Testing, September 1978, London.

8. Clarke, L.A., "Testing: Achievements and Frustrations," IEEE Computer Soc. 2nd International Computer Software and Applications Conference, Nov. 1978, Chicago.

9. DeMillo, R.A., Lipton, R.J., and Sayward, F.G., "Hints on Test Data Selections: Help for the Practical Programmer," Computer, Vol. II, No. 4, pp. 34-41, 1978.

10. Elspas, B. et al, "An Assessment of Techniques for Proving Program Correctness," Computing Surveys, Vol. 4, 1972.

11. Fairley, R.E., "An Experimental Program Testing Facility," IEEE Trans. Software Eng., Vol. SE-1, pp. 350-357, 1975.

12. Galvan, R.J., Tucker, A.G., and Fujii, R.U., "Flow-Language Study - Final Report," Logicon Rept. No. CDB-76084, 1976.

13. Gannon, C. et al, "JAVS Reference Manual," General Research Corporation. Rept. No. CR-1-722/1, Vol. 2, June 1978.

14. Gannon, C. et al, "JAVS Users Guide," General Research Corporation, Rept. No. CR-1-722/1, Vol. 1, June 1978.

15. Goodenough, J.B., and Gerhart, S.L., "Toward a Theory of Test Data Selection," IEEE Trans. Software Eng., Vol. SE-1, pp. 156-173, 1975.

16. Hollin, T.G. and Hansen, R.C., "Toward A Better Method of Testing Software," IEEE Computer Soc. 2nd International Computer Software and Applications Conference, Nov. 1978, Chicago, Ill.

17. Hetzel, W.C., "Program Test Methods," Publ. by Prentice-Hall, New York, 1972.

18. Hoffman, R.H., "User Information for the Interactive Automated Test Data Generator," NASA, Johnson Space Center, JSC Internal Note 74-FM-88, 1976.

19. Howden, W.E., "Functional Program Testing," IEEE Computer Soc. 2nd International Computer Software and Applications Conference, Nov. 1978, Chicago.

20. Howden, W.E., "Methodology for the Generation of Program Test Data," IEEE Trans. Comp., Vol. C-24, pp. 555-560, 1975.

21. Howden, W.E., "Reliability of the Path Analysis Testing Strategy," IEEE Trans. Software Eng., Vol. SE-2 pp. 208-214, 1976.

22. Howden, W.E., "Symbolic Testing and the DISSECT Symbolic Evaluation System," IEEE Trans. Software Eng., Vol. SE-3, pp. 266-278, 1977.

23. Howden, W.E., "Symbolic testing--design techniques, costs and effectiveness," U.S. National Bureau of Standards GCR77-89, National Technical Information Service PB268517, Springfield, Virginia 1977.

24. Howden, W.E., "Theoretical and Emperical Studies of Program Testing," Third International Conference on Software Engineering, pp. 305-311, 1978.

25. Huang, J.C., "An Approach to Program Testing," Computing Surveys, Vol. 7, pp. 113-128, 1975.

26. Ikezawa, M.A. and Kayfes, R.E., "A Structural Calculus for Program Analysis and Testing," 9th Annual Asilowar Conference on Circuits, Systems, and Computers, 1975.

27. Ikezawa, M.A., "An Introduction to AMPIC," Logicon Rept. No. CSS-75002, 1975.

28. King, J.C., "Symbolic Execution and Program Testing," Comm. ACM, Vol. 19, pp. 385-394, July 1976.

29. Kundu, S., "A New Program Analysis Tool With Applications to Test Data Generation," 3rd USA-Japan Computer Conference, Oct. 1978, San Francisco, CA.

30. Kundu, S., "SETAR - A New Approach to Test Case Generation, Infotech State of the Art Report, Software Testing Vol. II, pp. 162-186, 1979 Infotech, London.

31. Leach, Daniel M., AUTOMATED TEST CASE GENERATOR, FUNCTIONAL DESCRIPTION, Vols. I and II, 14 September 1979, Unclassified.

32. Leach Daniel M., AUTOMATED TEST CASE GENERATOR, SYSTEM/SUBSYSTEM SPECIFICATION, Vols. I and II, 14 September 1979, Unclassified.

33. Miller, E.F., "The Philosophy of Testing," in Tutorial on Program Techinques, (ed. E.F.Miller, Jr.), COMSAC 77, Chicago, pp. 1-3, 1977.

34. Miller, E.F., "RXVP - An Automated Verification System for FORTRAN," in Proc. Computer Science and Statistics: 8th Annual Symposium on the Interface, Los Angeles, CA. (1975).

35. Miller, E.F., and Melton, R.A., "Automated Generation of Testcase Datasets," in Proc. Int. Conf. Reliable Software, Los Angeles, CA (1975), pp. 51-58.

36. Osterweil, L.J., "The Detection of Unexecutable Program Paths through Static Data Flow Analysis," University of Colorado at Boulder, Technical Report #CU-CS-110-77, May, 1977.

37. Osterweil, L.J., and Fosdick, L.D., "DAVE-A Validation Error Detection and Documentation System for FORTRAN Programs," Software Practice and Experience, Vol. 6, pp. 473-486, 1976.

38. Paige, Michael R., " Program Graphs, an Algebra, and Their Implication for Programming" IEEE Trans. on Software Eng., Sept., 1975.

39. Ramamoorthy, C.V., and Ho, S.F., "Testing Large Software with Automated Software Evaluation Systems," IEEE Trans. Software Eng., Vol. SE-1, pp. 46-58, 1975.

40. Ramamoorthy, C.V., Ho, S.F., and Chen, W.T., "On The Automated Generation of Program Test Data," IEEE Trans. Software Eng., Vol. SE-2, pp. 293-300, 1976.

41. Reifer, D.J. and Trattner, S., "A Glossary of Software Tools and Techniques," IEEE Computer, July 1977.

42. Wright, M.H., "A Survey of Available Software for Nonlinearly Constrained Optimization," Stanford University Technical Report SOL 78-4, January, 1978.

APPENDIX C
AMPIC

C-1 INTRODUCTION

Logicon has developed a software product called AMPIC that establishes a solid foundation for meeting current and future needs in software verification, documentation, maintenance, and development. AMPIC is based on a technique that represents computer programs as "well-structured" programs regardless of how they were actually programmed. The advantages of such a program representation are exploited by AMPIC to provide various program analyses and user-oriented output that are difficult, if not impossible, to obtain otherwise.

The existing AMPIC capability (for a special military computer) includes:

- Automatic structuring and flowcharting of assembler language source programs
- Automatic translation of assembler language code into a higher level ("functional") form
- Semi-automatic path analysis that provides various path-oriented information, including the necessary conditions and the functional result of traversing each path
- Useful data concerning program structure characteristics, such as the relationship of code segments to loops and branches, logically inconsistent paths, and ill-structured segments
- Details of deductions by which the translations are produced

AMPIC is an extremely valuable tool where software development, maintenance, modification, integration, testing, and verification are conducted for complex programs. The inevitable problems that arise in an environment such as this can be attributed, to a large extent, to inadequate visibility into the state of the program. Examples are as follows:

1) Software Development: Documentation does not keep up with the changes made as the result of computer implementation considerations. Consequently, the development programmer does not really know his program when it is time to thoroughly test or integrate it, thus increasing the time and effort necessary to get a program accepted.

2) Software Testing/Verification: Test and verification personnel cannot determine precisely what constitutes a comprehensive test of a program. As a result, some segments of code are not tested at all while other segments are tested redundantly.

3) Software Modification/Maintenance: Modification and maintenance of software are typically given to personnel who are not involved in the original program development. As a result, much time is spent in becoming familiar with the program and many changes are implemented incorrectly.

AMPIC can greatly alleviate these problems. Documentation can be kept current thus enabling development, test, and verification personnel to stay abreast of the program status and to determine whether changes have adversely affected the original program specifications or negated other program functions. With its extensive capabilities for path analysis, AMPIC will greatly aid test and verification personnel in determining such factors as what tests must be conducted and the percentage of code checked out. For maintenance personnel, AMPIC will not only allow them to become familiar with the program more readily, but will also allow them to analyze proposed program changes before the changes are implemented.

The following is a brief introduction to the AMPIC approach. Section C-2 introduces the subject of program structure. It is followed in Section C-3 by an explanation of the concept of a well-formed structure. Functional translations of program code utilizing the simplified structural representation are provided in Section C-4. Section C-5 discusses how the functional translations can be combined into meaningful descriptions of each path through the program.

C-2 PROGRAM STRUCTURE

The term "program structure" refers to those properties of a computer program that determine the starting and ending points of the program and the routes that it is possible to traverse from starting to ending point. The simplest structure exists in a program consisting of a linear sequence of instructions that are to be executed from top to bottom. The total effect of such a sequence of instructions could be rather complex, but in structural terms it can be represented quite simply, as shown in Figure C-1 (a). Here the starting and ending points of the program are represented by circles labeled "IN" and "OUT", respectively, and the sequence of instructions is represented by a single rectangle labeled "S".

Program structures that contain single routes from start to end, regardless of the number of instructions, are trivial for modern computers as such programs do not utilize at all the computer's ability to make decisions. Next, consider programs with two or more routes. An example is a program that examines whether a number is positive or negative and then calculates its square root if positive and does nothing if negative. The structure of such a program could be shown as illustrated in Figure C-1 (b), where the decision based on the sign criterion is representd by the rhombus labeled "T" and the entire square root calculation is represented by the rectangle labeled "S". (It is assumed that the square root calculation is a simple sequence of instructions.) This structure is obviously more complex than the previous one. It contains new structural components represented by the

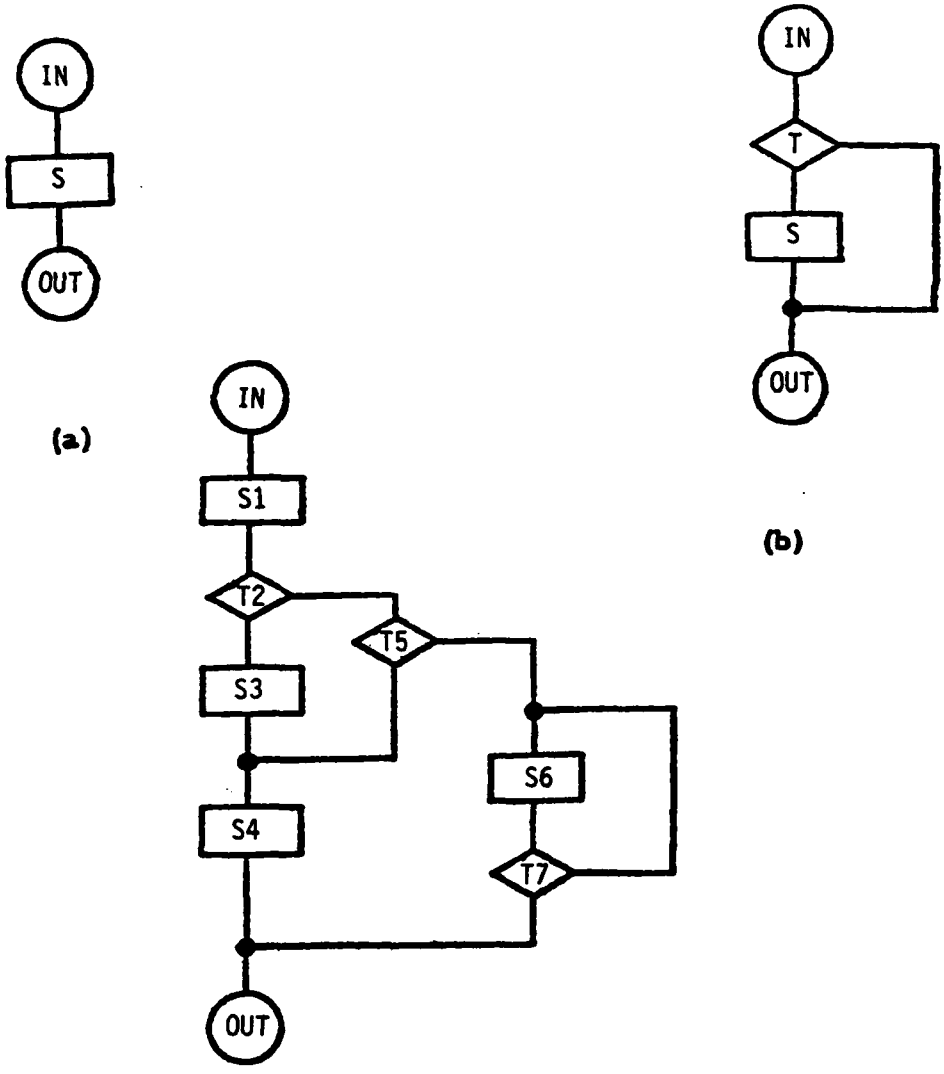


Figure C-1. Examples of Program Structures

rhombus and the dot. The former signifies route divergence and the latter signifies route convergence. They correspond to conditional jump instructions and jump destination location, respectively, in the computer program. Moreover, there are two routes possible through this program.

By allowing the "T"s and "S"s to take on subscripts and by allowing the lines emanating from the right side of rhombuses to go up as well as down, depending on where the convergence point is located, any type of program structure can be represented. Figure C-1 (c) is a random example of a program that has three routes, one of which contains a special kind of sub-route called a "loop" consisting of T7 and S6. Since the convergence point of the right side of T7 is at a point preceding S6, this path will return to T7.

All "flowcharts" of computer programs employ variations of the above scheme to represent the structural properties of computer programs. Instead of "T"s and "S"s, however, a translation of the program instructions are inserted within the rhombuses and rectangles. The nature of these translations and the degree of structural detail presented in a flowchart are indications of whether it is a "low" level or a "high" level flowchart. At the higher flowchart levels, English language statements replace mathematical or computer-oriented statements accompanied by a progressive degeneration of the uniqueness of meaning maintained at the lower levels. On the other hand, the more global meanings of programs are not apparent at the lower levels.

Another important consideration in describing a structural properties of a computer program is the clarity with which each route in the program structure is presented. A typical program structure is flow-charted either by employing lines that cross over each other or by employing a set of connector symbols to avoid crossovers. Neither technique conveys a clear picture of the individual routes nor of their interrelationships.

Both the structural route problem and the higher level translation problem mentioned above are attacked in the approach taken by the AMPIC technique. Each of these is discussed in the following two sections.

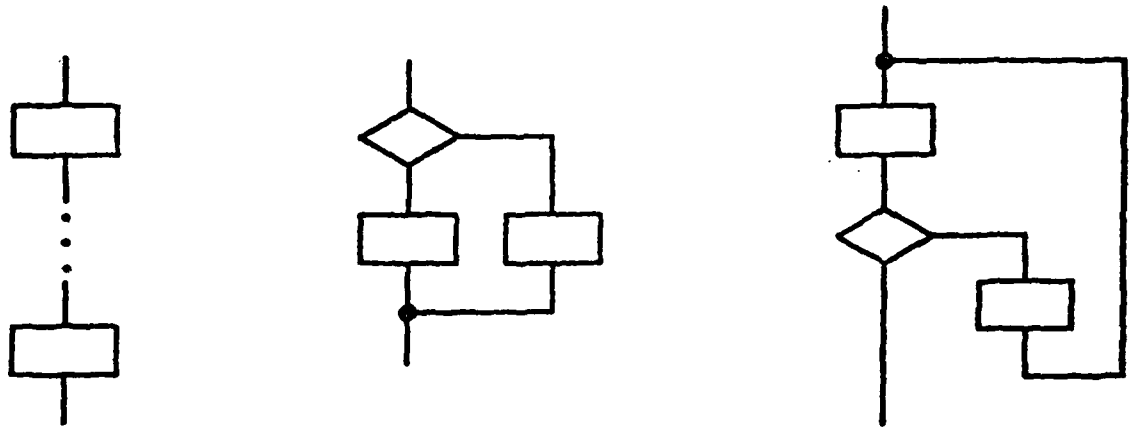
C-3 WELL-FORMED PROGRAM REPRESENTATION

If computer programs were not limited by the physical limitations of the hardware on which they are executed, their structures could be designed so that all the routes through the programs would be easily recognizable directly from their structural form. Unfortunately, execution speed and memory capacity considerations often dictate that programs be optimized for one or both of these. Hence, what is desirable for documentation, verification, maintenance, and analysis are undesirable for the operational configuration.

Suppose, however, that a computer program that has been optimized for operational use could be shown to have an exact equivalent that is optimized for the other activities and, moreover, that such an equivalent program can be produced automatically from the original. If this could be done, even with some limitations in extreme cases, it would no longer be necessary to look at typical flowcharts, except as a reminder of what the program could look like if not represented intelligibly.

AMPIC has been designed to produce such a logically equivalent program structure and to represent that structure in a systematic and convenient form. The graphic portion of the AMPIC output portrays the program structure in terms of the "T" and "S" elements discussed in Section C-2 in such a way that each route through the structure is immediately recognizable. Moreover, the program structure is in the form of a systematic hierarchy of substructures each of which contains only the structural forms shown in Figure C-2. (Readers who are familiar with tree structures may observe that this type of representation preserves the clarity of the tree representation without the tedious and space-consuming overhead inherent in the unfolding of the entire structure into non-converging paths.) Figure C-3 is a reproduction of an actual AMPIC output.

Although a representation such as Figure C-3 provides the structural properties of a computer program, it does not relate the structural elements to the



Notes: 1) Any of the rectangles may be absent in special cases of the above forms.

2) Any of the rectangles can itself be one of the above forms.

Figure C-2. Basic Structural Forms of AMPIC Representation

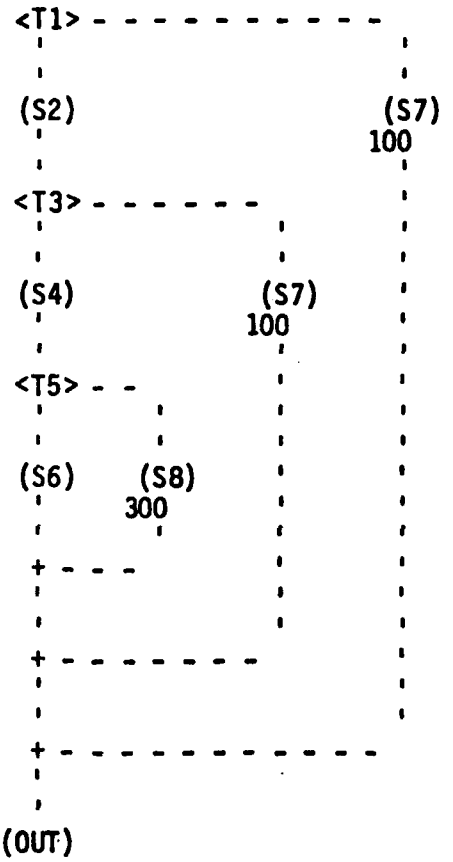


Figure C-3. Well-Structured Program Representation

program parts. Hence, AMPIC provides a mapping of the structural elements (the "T" and "S" elements) to the appropriate program parts in the format shown in Figure C-4. All instructions with no "T" or "S" number belong to the previous "T" or "S" element, except that unconditional jump instructions do not take part in this mapping.

C-4 TRANSLATION OF PROGRAM INSTRUCTIONS

The computer program shown in Figure C-4 is written in FORTRAN. The instructions are executed in sequence from top to bottom until a branch type instruction is encountered. A branch instruction may cause the sequence to continue at an instruction other than the one following the jump instruction.

The translation of the program shown in Figure C-4 is shown in Figure C-5. Translations of "S" elements are logical statements of the total effects of program instruction sequences. The effect of any single instruction in a sequence of instructions is not stated in the translation unless, of course, it is the only instruction in the sequence. (AMPIC provides an instruction-by-instruction translation as an option.) The translation of each "S" element is a set of mathematical relations that hold as a consequence of the corresponding instruction sequence.

Each variable appearing in any "T" or "S" element translation has an unstated superscript, as follows: All variables in "T" type translations have the superscript "OLD". All variables to the right of the equality (or inequality) symbol of "S" element translations have the superscript "OLD". All variables appearing in subscripts (i.e., within the outermost pair of parentheses) of variables to the left of the equality (or inequality) symbol of "S" element translations have the superscript "OLD." All other variables have the superscript "NEW" ("<NU>"). Keeping these superscripts in mind, the translations can be read as would any mathematical or logical statement. In the example of Figure C-5, names containing dots and followed by an empty pair of parentheses are functional symbols, not variables.

```
      SUBROUTINE ROOT
      IMPLICIT INTEGER*2(I,K-N)
      COMMON/ARGS/A,B,C
      COMMON/VALUE/ROOT1,ROOT2,NROOT
      IF(A.EQ.0) GOTO 100
      TEMP1 = 4.*A*C
      TEMP1 = B*B - TEMP1
      IF(TEMP1.LT.0) GOTO 100
      TEMP2 = 2.*A
      ROOT2 = -(B/TEMP2)
      IF(TEMP1.EQ.0) GOTO 300
      TEMP2 = FSQRT(TEMP1)/TEMP2
      ROOT1 = ROOT2 + TEMP2
      ROOT2 = ROOT2 - TEMP2
      NROOT = 2
      RETURN
100   NROOT = 0
      RETURN
300   ROOT1 = ROOT2
      NROOT = 1
      RETURN
      END
```

```
T1    IF(A.EQ.0.) GOTO S7

S2    TEMP1=4.*A*C
      TEMP1=B*B-TEMP1

T3    IF(TEMP1.LT.0.) GOTO S7

S4    TEMP2=2.*A
      ROOT2=-B/TEMP2

T5    IF(TEMP1.EQ.0.) GOTO S8

S6    TEMP2=FSQRT(TEMP1)/TEMP2
      ROOT1=ROOT2+TEMP2
      ROOT2=ROOT2-TEMP2
      NROOT=2
      GOTO OUT

S7    NROOT=0
      GOTO OUT

S8    ROOT1=ROOT2
      NROOT=1
      GOTO OUT
```

Figure C-4. Mapping of Structural Element to Source Code

```
T1:  JUMP       (A = 0.)
     NO JUMP    (A ≠ 0.)

S2:  TEMP1<NU> = B * B - 4. * A * C

T3:  JUMP       (TEMP1 < 0.)
     NO JUMP    (TEMP1 ≥ 0.)

S4:  TEMP2<NU> = 2. * A
     ROOT2<NU> = - B / (2. * A)

T5:  JUMP       (TEMP1 = 0.)
     NO JUMP    (TEMP1 ≠ 0.)

S6:  TEMP2<NU> = .FSQRT(TEMP1) / TEMP2
     ROOT1<NU> = ROOT2 + .FSQRT(TEMP1) / TEMP2
     ROOT2<NU> = ROOT2 - .FSQRT(TEMP1) / TEMP2
     NROOT<NU> = 2

S7:  NROOT<NU> = 0

S8:  ROOT1<NU> = ROOT2
     NROOT<NU> = 1
```

Figure C-5. Translations of Structural Elements

C-5 COMBINING TRANSLATED ELEMENTS

The translations of Figure C-5 together with the structural representations of Figure C-3 are suitable for those who wish to remain at a level just above the program instruction level. Others will want to be at one or two levels higher. AMPIC allows such a level to be chosen by the user.

For example, the user may wish to dispense with translations of individual "T" and "S" elements but may be interested in the complete translations of each route through the program. In Figures C-6 and C-7, two of the possible routes through the example program were processed by AMPIC. AMPIC will use the previously derived information and provide input-output functional expressions for entire paths through the input module.

The examples in Figures C-6 and C-7 show this for two of the four paths in the FORTRAN language module.

The first example (Figure C-6) is the functional expression for the path that goes straight down from T1 to OUT. The line labeled "PATH" identifies this path. "Pn" means "go down" ("no jump") (take the false branch) at Tn. ("Rn" means that the "jump" exit is taken at Tn.) Hence, (P1, S2, P3, S4, P5, S6, OUT) is an ordered set that describes the path.

The "IF" statement describes the three conditions to be satisfied (P1, P3, and P5) in order to traverse this path. The conditions are with respect to the input values of A, B, and C.

The "THEN" statement describes the total input-output functional results of traversing this particular path. (The "<NU>" is now the output with respect to the whole path. Variables without "<NU>" are inputs with respect to the whole path).

The second example (Figure C-7) shows a similar result of traversing another route in the example program.

```
PATH:  (P1,S2,P3,S4,P5,S6,OUT)

IF:    (A ≠ 0.) & (B * B - 4. * A * C ≥ 0.) & (B * B - 4. * A * C ≠ 0.)

THEN:  TEMP1<NU> = B * B - 4. * A * C
       TEMP2<NU> = .FSQRT(B * B - 4. * A * C) / (2. * A)
       ROOT2<NU> = - B / (2. * A) - .FSQRT(B * B - 4. * A * C) / (2. * A)
       ROOT1<NU> = - B / (2. * A) + .FSQRT(B * B - 4. * A * C) / (2. * A)
       NROOT<NU> = 2
```

Figure C-6. Example of Combined Translation (Case 1)

```
PATH:  (P1,S2,P3,S4,R5,S8,OUT)

IF:    (A ≠ 0.) & (B * B - 4. * A * C ≥ 0.) & (B * B - 4. * A * C = 0.)

THEN:  TEMP1<NU> = B * B - 4. * A * C
       TEMP2<NU> = 2. * A
       ROOT2<NU> = - B / (2. * A)
       ROOT1<NU> = - B / (2. * A)
       NROOT<NU> = 1
```

Figure C-7. Example of Combined Translation (Case 2)