<table>
<thead>
<tr>
<th>UNCLASSIFIED</th>
<th>GRC-CR-1-925</th>
<th>AFOSR-TR-80-0234</th>
<th>N</th>
</tr>
</thead>
</table>

**ADAPTIVE SEARCH TECHNIQUES APPLIED TO SOFTWARE TESTING.** (U)
FEB 80  

J P BENSON, D M ANDREWS

F49624-79-C-0115
Adaptive Search Techniques Applied to Software Testing

Final Report

February 1980

General Research Corporation
A Subsidiary of Flow General Inc.
P.O. Box 6770, Santa Barbara, California 93111

Sponsored by
Air Force Office of Scientific Research
Bolling Air Force Base
Washington, D.C.
Under Contract F49620-79-C-0115

Approved for public release! distribution unlimited.
In addition to approval by the Project Leader and Department Head, General Research Corporation reports are subject to independent review by a staff member not connected with the project. This report was reviewed by R. L. Stone.

Sponsored by

Air Force Office of Scientific Research
Under Contract F49620-79-C-0115
An experiment was performed in which executable assertions were used in conjunction with search techniques in order to test a computer program automatically. The program chosen for the experiment computes a position on an orbit from the description of the orbit and the desired point. Errors were inserted in the program randomly using an error generation method based on published data defining common error types. Assertions were written for the program and it was tested using two different techniques. The first divided up the range of the input variables and selected test cases from within...

The second used a search algorithm from optimization theory. This entailed using the assertions to define an error function and then maximizing its value. The program was then tested by varying only a limited number of the input variables and a second time by varying all of them. The results indicate that this search testing technique was as effective as the grid testing technique in locating errors and was more efficient. In addition, the search testing technique located critical input values which helped in writing correct assertions.
Abstract

An experiment was performed in which executable assertions were used in conjunction with search techniques in order to test a computer program automatically. The program chosen for the experiment computes a position on an orbit from the description of the orbit and the desired point.

Errors were inserted in the program randomly using an error generation method based on published data defining common error types. Assertions were written for the program and it was tested using two different techniques. The first divided up the range of the input variables and selected test cases from within the subranges. In this way a "grid" of test values was constructed over the program's input space.

The second used a search algorithm from optimization theory. This entailed using the assertions to define an error function and then maximizing its value. The program was then tested by varying only a limited number of the input variables and a second time by varying all of them. The results indicate that this search testing technique was as effective as the grid testing technique in locating errors and was more efficient. In addition, the search testing technique located critical input values which helped in writing correct assertions.
## CONTENTS

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>1</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENT</td>
<td>vii</td>
</tr>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
</tr>
<tr>
<td>1.1 Problems with Testing</td>
<td>1-1</td>
</tr>
<tr>
<td>1.2 A Proposed Solution</td>
<td>1-2</td>
</tr>
<tr>
<td>1.3 Combining Assertions and Search Algorithms</td>
<td>1-5</td>
</tr>
<tr>
<td>1.4 Overview of the Experiment</td>
<td>1-6</td>
</tr>
<tr>
<td>1.5 The Program</td>
<td>1-6</td>
</tr>
<tr>
<td>1.6 The Search Routine</td>
<td>1-7</td>
</tr>
<tr>
<td>1.7 The Test Driver</td>
<td>1-7</td>
</tr>
<tr>
<td>1.8 The Assertions</td>
<td>1-7</td>
</tr>
<tr>
<td>1.9 Selecting Errors</td>
<td>1-8</td>
</tr>
<tr>
<td>1.10 Testing Techniques</td>
<td>1-8</td>
</tr>
<tr>
<td>1.11 Results</td>
<td>1-9</td>
</tr>
<tr>
<td>2</td>
<td>THE TEST PROGRAM</td>
</tr>
<tr>
<td>2.1 The Search Routine</td>
<td>2-1</td>
</tr>
<tr>
<td>2.2 Test Driver</td>
<td>2-4</td>
</tr>
<tr>
<td>3</td>
<td>THE ASSERTIONS</td>
</tr>
<tr>
<td>4</td>
<td>THE ERRORS</td>
</tr>
<tr>
<td>4.1 The Error Seeding Method</td>
<td>3-1</td>
</tr>
<tr>
<td>5</td>
<td>THE EXPERIMENT</td>
</tr>
<tr>
<td>5.1 Grid Tests</td>
<td>4-6</td>
</tr>
<tr>
<td>5.2 Search Varying Three Parameters</td>
<td>5-1</td>
</tr>
<tr>
<td>5.3 Search Varying All Parameters</td>
<td>5-3</td>
</tr>
<tr>
<td>6</td>
<td>RESULTS OF THE EXPERIMENT</td>
</tr>
<tr>
<td>6.1 Error Detection Using Assertions</td>
<td>6-1</td>
</tr>
<tr>
<td>6.2 Effectiveness of the Search Techniques</td>
<td>6-3</td>
</tr>
<tr>
<td>6.3 Efficiency of the Search Method</td>
<td>6-7</td>
</tr>
<tr>
<td>6.4 Special Cases</td>
<td>6-10</td>
</tr>
</tbody>
</table>
## CONTENTS (cont.)

<table>
<thead>
<tr>
<th>SECTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 DISCUSSION</td>
<td>7-1</td>
</tr>
</tbody>
</table>

BIBLIOGRAPHY

APPENDIX A - PROGRAM LISTINGS | A-1
APPENDIX B - CHRONOLOGICAL LIST OF PAPERS SUBMITTED | B-1
APPENDIX C - PERSONNEL ASSOCIATED WITH THE PROJECT | C-1
# FIGURES

<table>
<thead>
<tr>
<th>NO.</th>
<th>Title</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Cost as a Function of Building Material</td>
<td>1-4</td>
</tr>
<tr>
<td>1.2</td>
<td>Examples of Assertions</td>
<td>1-5</td>
</tr>
<tr>
<td>2.1</td>
<td>Calculation of Dependent Orbital Parameters</td>
<td>2-2</td>
</tr>
<tr>
<td>2.2</td>
<td>Calculating Radius and Altitude</td>
<td>2-5</td>
</tr>
<tr>
<td>2.3</td>
<td>Coordinates of the Vertices of a Complex</td>
<td>2-6</td>
</tr>
<tr>
<td>2.4</td>
<td>Complex Transformations</td>
<td>2-8</td>
</tr>
<tr>
<td>3.1</td>
<td>An Example of Range Assertions</td>
<td>3-3</td>
</tr>
<tr>
<td>3.2</td>
<td>An Example of Relationship Assertions</td>
<td>3-3</td>
</tr>
<tr>
<td>3.3</td>
<td>An Example of History Assertions</td>
<td>3-3</td>
</tr>
<tr>
<td>4.1</td>
<td>An Error Packet</td>
<td>4-8</td>
</tr>
<tr>
<td>5.1</td>
<td>Search Program Output Template</td>
<td>5-5</td>
</tr>
<tr>
<td>5.2</td>
<td>Complex Initialization</td>
<td>5-6</td>
</tr>
<tr>
<td>5.3</td>
<td>Intermediate Stage of Search Testing</td>
<td>5-7</td>
</tr>
<tr>
<td>5.4</td>
<td>Later Stage of Search Testing</td>
<td>5-8</td>
</tr>
<tr>
<td>5.5</td>
<td>Summary of Search Testing for Error 13</td>
<td>5-9</td>
</tr>
<tr>
<td>6.1</td>
<td>Error 28</td>
<td>6-4</td>
</tr>
<tr>
<td>6.2</td>
<td>Arctangent Function</td>
<td>6-5</td>
</tr>
<tr>
<td>6.3</td>
<td>Value of Divisor at ( \pi )</td>
<td>6-5</td>
</tr>
<tr>
<td>6.4</td>
<td>Error 47</td>
<td>6-5</td>
</tr>
<tr>
<td>6.5</td>
<td>Error 14</td>
<td>6-6</td>
</tr>
<tr>
<td>6.6</td>
<td>Error 74</td>
<td>6-8</td>
</tr>
<tr>
<td>6.7</td>
<td>First Incorrect Assertion</td>
<td>6-11</td>
</tr>
<tr>
<td>6.8</td>
<td>Increasing and Decreasing Radii</td>
<td>6-12</td>
</tr>
<tr>
<td>6.9</td>
<td>Code to Locate Point on Radius</td>
<td>6-13</td>
</tr>
<tr>
<td>6.10</td>
<td>Checking the Value of the MODE Parameter</td>
<td>6-14</td>
</tr>
<tr>
<td>6.11</td>
<td>Checking the Angle from Perigree</td>
<td>6-15</td>
</tr>
<tr>
<td>NO.</td>
<td>TABLES</td>
<td>PAGE</td>
</tr>
<tr>
<td>-----</td>
<td>-----------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.1</td>
<td>Results from Grid Tests</td>
<td>1-10</td>
</tr>
<tr>
<td>1.2</td>
<td>Effectiveness of Second Testing Techniques</td>
<td>1-10</td>
</tr>
<tr>
<td>1.3</td>
<td>Detection of Assertion Violations by Search Method</td>
<td>1-12</td>
</tr>
<tr>
<td>2.1</td>
<td>Orbital Element Vector Parameters</td>
<td>2-1</td>
</tr>
<tr>
<td>2.2</td>
<td>State Vector Parameters</td>
<td>2-3</td>
</tr>
<tr>
<td>2.3</td>
<td>Mode and Value Parameters</td>
<td>2-4</td>
</tr>
<tr>
<td>4.1</td>
<td>Error Types Used in Experiment</td>
<td>4-2</td>
</tr>
<tr>
<td>4.2</td>
<td>Types of Errors Used in the Experiment</td>
<td>4-5</td>
</tr>
<tr>
<td>4.3</td>
<td>Relationships Between Software Properties and Error Types</td>
<td>4-7</td>
</tr>
<tr>
<td>4.4</td>
<td>Errors Used in the Experiment</td>
<td>4-9</td>
</tr>
<tr>
<td>5.1</td>
<td>Standard Orbital Parameters</td>
<td>5-2</td>
</tr>
<tr>
<td>5.2</td>
<td>Possible Search Actions</td>
<td>5-5</td>
</tr>
<tr>
<td>6.1</td>
<td>Errors Not Detected by Assertions</td>
<td>6-2</td>
</tr>
<tr>
<td>6.2</td>
<td>Assertion Violations Detected by Each Testing Method</td>
<td>6-8</td>
</tr>
<tr>
<td>6.3</td>
<td>First Assertion Violations Detected by All-Variable Search</td>
<td>6-9</td>
</tr>
</tbody>
</table>
Acknowledgement

This research project was sponsored by the Air Force Office of Scientific Research, Contract Number F49620-79-C-0115. Lt. Col. George W. McKemie was the program manager.

Participating in the project were J. P. Benson, principal investigator, D. M. Andrews, N. B. Brooks, R. N. Meeson, and D. W. Cooper.
INTRODUCTION

Although Dijkstra's famous comment on testing, that it will never show the absence of bugs, only their presence, is undoubtedly true, testing is still the method most used for showing the correctness of software. If testing is to be used, ways must be found to make it more efficient and effective.

A paper by Alberts¹ presents data indicating that testing and validation efforts account for approximately 50% of the cost of developing a software system, where development includes the typical phases of conceptual design, requirements analysis, development, and operational use. This cost includes those associated with locating the errors, correcting the errors (which may include redesign), and checking that the corrections have removed the cause of the error. The testing process is a very labor-intensive activity, as is any aspect of software development. If methods could be found to automate the testing process, the cost of developing software could be reduced.

1.1 PROBLEMS WITH TESTING

Two of the many problems involved in testing software are (1) how to develop test cases which identify errors and (2) how to check the results from these test cases. Before software testing can be automated and its cost reduced, these two problems must be solved.

Many methods have been proposed for identifying test cases which will show that a program performs correctly or indicate the errors which are present in the program. For examples of these methods see Howden²


and Gannon. Basically, the problem is one of complexity. For most programs, the number of different combinations of input values is practically infinite. Therefore, using exhaustive testing to show that a program works correctly is an impossible task.

Given the fact that programs cannot be tested by trying all test cases, what are the alternatives? Boundary value testing, path testing, and symbolic execution have been some of the suggested solutions. The key problem is finding test cases which detect the errors present in the software. At present, there are no methods for deriving test cases with this property although many studies of the types of errors commonly found in software have been undertaken.

The second problem has to do with checking whether a test has been successful. Even if there were a method for selecting test cases which was able to identify specific errors in a program, the process of evaluating whether or not the program ran successfully is a manual one. The output from the program must be compared with the expected results. For large programs composed of many functions this is a very time-consuming task.

1.2 A PROPOSED SOLUTION

From the above discussion, it is evident that automating the testing of computer programs requires finding methods for developing

---

effective test cases as well as methods for efficiently evaluating the results of using them. A method for solving these problems has been developed that combines the use of search algorithms from operations research with executable assertions from software verification research.

Finding the maximum or minimum value of a function of several variables each subject to some set of constraints is a common problem in operations research. Minimizing the cost of constructing a building given the choice of using brick, wood, and adobe materials in different proportions typifies problems of this sort. Many methods have been developed for solving such problems, for example, see Denn.¹ One of the simplest is to define the parameter of interest (e.g., cost) as a function of the possible alternatives (e.g., brick, wood, adobe). The problem then is to find a minimum value of the function defined by the values of the alternatives (variables). Figure 1.1 illustrates this for two variables, brick and wood. The cost function defines a surface, with "hills" (maxima) and "valleys" (minima).

The goal is to find a point on this surface which is a minimum (in the example of building cost). This point corresponds to a particular set of values of the alternatives or variables. Finding such a minimum value requires that this surface be searched. There are many methods for traversing the surface according to some search heuristic (for example, in the direction of the gradient) until a solution is found.

The problem of evaluating the results limits the application of these techniques to the testing of computer programs. That is, in operations research, we are usually trying to maximize or minimize the

value of one variable, whereas in software testing we are usually trying to compare the value of many output variables with their expected values.

The solution to this problem has been found in "executable assertions," a technique developed for proving software correct and for checking it while it is running. Assertions are comments added to a program which specify how the program is to behave. They may specify a range of values for a variable, the relation the values of two or more variables have to each other or compare the state of a present computation to that of a past computation. Figure 1.2 shows an example of two assertions that specifies the range of values that the variable VALUE can assume.

To make an assertion "executable," we merely translate it into machine language. Then while the program is running, the assertion can
be evaluated. As in the case of a logical function, the assertion has a value of true or false. If the value of an assertion becomes false at any point in the execution of a program, then this can be reported as any other error message.

1.3 COMBINING ASSERTIONS AND SEARCH ALGORITHMS

Assertions give us a method for evaluating whether a program has run correctly without looking at all of its output. If the assertions are written correctly and they completely specify the algorithm, then the correctness of the program can be determined while the program is running. This is not to say that writing assertions to accomplish this is easy; a comprehensive and complete set of assertions for a program is difficult to develop. But if it can be done, then the problem of examining the output of a program to determine whether it executed a test case correctly has been solved.

Since using assertions means that we no longer have to examine the output of a program, the automated testing of computer programs becomes possible—provided we can automate the selection of test cases. If we can transform the output from the assertions into a function, we can utilize the search techniques from operations research to locate errors.

The basic idea is this: The function we define is the number of assertions that become false during the execution of a particular test case. The independent variables are the values of the input variables of the program. The search technique will be used to find the values of
the input variables for which the maximum number of assertions are violated. The function relating the number of assertions violated to the values of the input variables is called the "error function," and the surface that it describes is called the "error space."

If the search algorithm is to perform correctly, the error function must (1) not define a flat (uniform) surface and (2) not be discontinuous (have spikes) at any points. A previous experiment, investigated the error function for a scheduling program. It was found that the error function for this program was neither uniform nor discontinuous. In a second experiment, described below, we have attempted to show that this is also true for another program "seeded" with several types of errors. We have also attempted to determine the efficiency of the search technique in locating these errors relative to other types of testing methods.

1.4 OVERVIEW OF THE EXPERIMENT

The experiment was to select a program, add assertions to it, and seed it with errors from a list of typical software errors. The location of the errors was determined randomly. Each of the errors was inserted in the program one at a time and the program was then tested by systematically choosing combinations of values for the input parameters. This testing was done automatically by a program which varied the input parameters over the required values. After this, the program was tested by the search routine, first by allowing the search algorithm to vary the same variables that were varied in the first tests, and then allowing it to vary all of the input variables.

1.5 THE PROGRAM

The program selected takes an orbit described by six independent parameters (longitude of the ascending node, inclination of the orbit

---

plane, angle of the perigee, eccentricity, time at perigee, and semi-
minor axis) and converts this description into a state vector representa-
tion of a point on the orbit (time, position, velocity, and accelera-
tion). The point is determined by the values of two other parameters.
The range of values of one of these parameters is dependent upon the
other. In all, there are ten input parameters, seven of which are
independent of the others.

1.6 THE SEARCH ROUTINE

The search routine chosen for the experiment was one developed by
Box1 called complex search. This algorithm constructs a hypertriangle,
or complex, of the values of the function from several tests and then
rotates, shrinks, expands, and projects the complex in order to locate a
value which is larger (in the case of finding the maximum) than the
worst point currently in the complex. The worst point is then replaced
by the new point and the process continued until no further progress can
be made.

1.7 THE TEST DRIVER

Several programs were also written in order to support the testing
and make it as automatic as possible: (1) A test driver, which handled
the selection of the testing method to be used and read in an initial
test case was written, (2) a set of subroutines which implemented the
constraints among the input variables used in generating new values for
the search routine, and (3) a set of routines to count the number of
assertions violated in each test and print the results.

1.8 THE ASSERTIONS

Assertions added to the program were of three types: (1) those
that described ranges of variable values, (2) those that described the
relationship between values of variables, and (3) those which kept track
of the history of the computation. Two routines were also written which
included assertions to check the values of the input variables and the

---

1 M. J. Box, "A New Method of Constrained Optimization and Comparison
correctness of the results. These routines were invoked at the begin-
ing of the test program and at the end of the test program.

1.9 SELECTING ERRORS

Certain categories of errors were selected from a list of common software errors. Errors of these types were inserted into the test program by randomly selecting sites (statements in the program) where the particular type of error could occur.

1.10 TESTING TECHNIQUES

The program was then tested by inserting one error at a time. First, the program was tested by taking combinations of values from three input variables. The permissible input range of each of the variables was divided up into equal subranges so that a reasonable number of test cases could be performed. Test values for each variable were selected by choosing the end-points of each subrange. The program was then tested using the selected values for the three input variables. First, the values of two of the three variables were fixed at a value selected from their range of test values. Then, a test was run for each of the test values of the third variable. The value of the third variable was then fixed, and the first variable was varied over its set of test values. After this, the values of the first and third variable were fixed and the second variable was varied. The testing continued until all combinations of the test values for the three variables had been used. In this way a "grid" over the input space was obtained. The values of the variables which caused assertions to be violated and the number of assertions violated were recorded.

A majority of the errors (15 out of 24) were not detected by the original assertions for a number of reasons. Two of the errors were not detected since they occurred only if another error had occurred previously during program execution. For other errors, it was found to be very difficult to write assertions that would detect them. Finally,
eight of the errors were not detected simply because the program did not contain enough assertions. In order to investigate the performance of the search algorithm, new assertions were added to the program and the grid tests were run again. Errors which were not detected in this second set of tests were removed from the list of errors used in the experiment.

Next, the errors were again inserted one at a time and the search routine was allowed to vary only the variables which were varied in the grid tests. The number of assertions violated and the input values which caused the violations were recorded.

Finally, the errors were again used one at a time; but this time the search routine was allowed to vary any of the seven independent variables in order to locate a maximum. Again, the assertions violated and the input values which caused the violations were recorded.

1.11 RESULTS

The results from the grid tests demonstrated the effectiveness of the assertions in detecting the errors. Table 1.1 shows the results of these tests. Of the original 24 errors, nine (thirty-eight percent) were detected by the original assertions, and eight (thirty-three percent) were detected by the assertions that were added. (The seven errors, twenty-nine percent, which could not be detected by assertions, were not tested).

The relative effectiveness of the search testing methods versus the grid testing method is summarized in Table 1.2. (In this table, and those following, the "error number" column refers to a unique number assigned to each error by the error generation method discussed in Sec. 4.) In one case, the grid technique caused an assertion violation which
TABLE 1.1
RESULTS FROM GRID TESTS

<table>
<thead>
<tr>
<th>Errors Detected by Original Assertions</th>
<th>Number</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Errors Detected by Added Assertions</td>
<td>8</td>
<td>33</td>
</tr>
<tr>
<td>Errors Not Detected by Assertions</td>
<td>7</td>
<td>29</td>
</tr>
<tr>
<td>Total</td>
<td>24</td>
<td>100</td>
</tr>
</tbody>
</table>

TABLE 1.2
EFFECTIVENESS OF SECOND TESTING TECHNIQUES

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Grid</th>
<th>3-Variable Search</th>
<th>All-Variable Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>74</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
neither search technique caused. In another case, the search technique using all variables was not able to cause an assertion violation that was caused by the grid technique and the search varying three variables. On the other hand, the search technique using all variables was able to cause an assertion violation which neither the grid technique nor the search using three variables was able to cause. Finally, in one case the search technique using three variables caused an assertion violation that the grid technique did not cause while the search using all variables caused another assertion violation in addition to the one discovered by the search using three variables. In all other tests, each of the methods caused the same assertions to be violated.

The efficiency of the search technique was not measured directly, but an estimate of the behavior of the all-variable search technique in relation to the grid technique can be given. Except for error 52, which required 683 tests, the grid technique required 317 tests. In the case of the search method which varied all input variables, Table 1.3 shows, for each error, the number of the test in which the first assertion violation was detected. In all, fifteen of the seventeen detectable errors were detected by the seventh test in the search.
<table>
<thead>
<tr>
<th>Error Number</th>
<th>Test Number of First Assertion Violation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>28</td>
<td>*</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
</tr>
<tr>
<td>37</td>
<td>5</td>
</tr>
<tr>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>47</td>
<td>57</td>
</tr>
<tr>
<td>48</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>3</td>
</tr>
<tr>
<td>54</td>
<td>3</td>
</tr>
<tr>
<td>56</td>
<td>5</td>
</tr>
<tr>
<td>57</td>
<td>7</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>67</td>
<td>5</td>
</tr>
<tr>
<td>74</td>
<td>2</td>
</tr>
</tbody>
</table>

*No assertion violations detected.*
THE TEST PROGRAM

The program selected for testing is one of a number of subroutines in a program library called TRAID. This set of programs is used to compute solutions to orbital mechanics problems. The particular program, ORBP, was written in 1968 and has been used extensively since that time. It has undergone several revisions. The function of the program is to take as input an orbit described by a set of eight parameters or orbital elements (only six of which are independent), and produce from this set a state vector representation of a point on the orbit. The state vector includes the time, position, velocity and acceleration in three dimensions. The particular point on the orbit is specified by a parameter (MODE), which, in conjunction with another parameter (VALUE), allows the state vector describing the point to be computed. (For a simple discussion of the methods for describing orbits see Macko.) The orbital element vector is shown in Table 2.1 along

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Longitude of the ascending node</td>
<td>0 to 2π</td>
</tr>
<tr>
<td>2. Inclination of the orbit plane</td>
<td>0 to π</td>
</tr>
<tr>
<td>3. Angle of the perigee</td>
<td>0 to 2π</td>
</tr>
<tr>
<td>4. Semi-latus rectum</td>
<td>dependent</td>
</tr>
<tr>
<td>5. Eccentricity (E)</td>
<td>0.1 to 0.9</td>
</tr>
<tr>
<td>6. Time at perigee</td>
<td>0 to period</td>
</tr>
<tr>
<td>7. Period divided by 2π</td>
<td>dependent</td>
</tr>
<tr>
<td>8. Semi-major axis (A)</td>
<td>6,375,180 to 35,861,000 meters</td>
</tr>
</tbody>
</table>


with the ranges of each independent parameter as used to test the program. The letters E and A are used to indicate the eccentricity and semi-major axis respectively.

An orbit is described by the following eight parameters: (1) longitude of the ascending node, (2) inclination of the orbit plane, (3) argument (angle) of the perigee, (4) semi-latus rectum, (5) eccentricity, (6) time at perigee, (7) period divided by two pi, and (8) the semi-major axis. Of these eight parameters, the semi-latus rectum and the period are dependent upon the others; they are included in the vector only to simplify the calculations. The way in which these parameters are calculated from the others is shown in Fig. 2.1.

The output state vector is shown in Table 2.2. It includes the time at the point on the orbit, and the position, velocity and acceleration in three dimensions. These last parameters are given in a coordinate system relative to the center of the earth.

\[
\text{Semi-latus rectum} = A \times (1 - E^2)
\]

where

\begin{align*}
A & = \text{semi-major axis} \\
E & = \text{eccentricity}
\end{align*}

\[
\text{Period} = (A \times A/GCON) \times 2\pi
\]

where

\begin{align*}
GCON & = \text{gravitational constant} = 3.9857 \times 10^{14}
\end{align*}

Figure 2.1. Calculation of Dependent Orbital Parameters
TABLE 2.2
STATE VECTOR PARAMETERS

Parameter
1. Time
2. X-coordinate
3. Y-coordinate
4. Z-coordinate
5. X-velocity
6. Y-velocity
7. Z-velocity
8. X-acceleration
9. Y-acceleration
10. Z-acceleration

Together, the parameters MODE and VALUE specify a point on the orbit. The possible values of the mode parameter and the corresponding ranges of the value parameter are shown in Table 2.3. The mode parameter directs ORBP to perform one of six possible computations to locate a point on the orbit specified by the orbital element vector. The MODE parameter indicates how the VALUE parameter is to be interpreted. That is, the value parameter is only a number, the MODE parameter indicates what that number stands for. For example, MODE could indicate the time at the desired point, and therefore VALUE could assume any value between 0 and the period of the orbit. The six possible modes are: (1) angle of the point from the perigee point, (2) radius in the increasing direction (i.e., toward apogee), (3) radius in the decreasing direction (toward perigee), (4) time, (5) altitude in the increasing direction, and (6) altitude in the decreasing direction.

According to the values of MODE and VALUE, ORBP calculates a state vector using the orbital element vector. The calculations for altitude and radius are performed using the same code. This is done by adding
TABLE 2.3
MODE AND VALUE PARAMETERS

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning of Value</th>
<th>Range of Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Angle from perigee</td>
<td>0 to 2π</td>
</tr>
<tr>
<td>1</td>
<td>Increasing radius</td>
<td>( R_{\text{min}} ) to ( R_{\text{max}} )</td>
</tr>
<tr>
<td>2</td>
<td>Decreasing radius</td>
<td>( R_{\text{min}} ) to ( R_{\text{max}} )</td>
</tr>
<tr>
<td>3</td>
<td>Time</td>
<td>0 to Period</td>
</tr>
<tr>
<td>4</td>
<td>Increasing altitude</td>
<td>( \text{Alt}<em>{\text{min}} ) to ( \text{Alt}</em>{\text{max}} )</td>
</tr>
<tr>
<td>5</td>
<td>Decreasing altitude</td>
<td>( \text{Alt}<em>{\text{min}} ) to ( \text{Alt}</em>{\text{max}} )</td>
</tr>
</tbody>
</table>

the radius of the earth to \( \text{VALUE} \) if it corresponds to an altitude (MODE equal 4 or 5). (See Fig. 2.2.) The point on the orbit is found by computing the angle between the point and the perigee and the radius from the focus of the orbit (the center of the earth) to the point. The only loop in the program occurs when MODE indicates that \( \text{VALUE} \) is to be interpreted as time. In this case, an iterative algorithm is used to calculate the angle of the point from the perigee.

2.1 THE SEARCH ROUTINE

The search routine selected for the experiment was one invented by Box,\(^1\) called complex search. It is a method for solving for the maximum or minimum of a nonlinear function. The independent values of the function may be limited by nonlinear inequality constraints. The independent values of the function along with the function value define a space. The set of values of the function define a hyperplane in the

\(^1\)Box, op. cit.
Radius = A * (1 - E)

where

A = semi-latus rectum
E = eccentricity

Altitude = radius - RBODY

where

RBODY = radius of earth = 6,375,180 meters

Figure 2.2. Calculating Radius and Altitude

This hyperplane can then be expanded or contracted to find an extremum of the function. The hyperplane is called a "complex."

The technique is as follows. Choose a set of values of the independent variables at random (subject to constraints) and determine the value of the function from these values. The independent values and the function value define a point on the complex. Define other points in the same way until there is one more point than the number of independent variables in the function. Then replace the point with the worst function value with a new point. The new point is found by constructing the line formed by the rejected point and the centroid of the remaining points. A set of coefficients is then calculated to determine the exact location of the new point. These coefficients determine the degree of reflection, expansion, shrinkage, contraction, and rotation to be applied in forming the new set of points. New points are selected for the complex using the above technique until a solution is found. This technique is somewhat immune to irregularities (hills and valleys) in the surface being searched.
The search program was adapted from an implementation by Cooper which was used during the Adaptive Testing Experiment. In general, the function may have many independent variables. In order for the search routine to function correctly, there must be one more point in the complex than there are independent variables in the function. For example, if the function has two independent variables, then the complex would have three vertices. That is, it would be a triangle. The coordinates of each vertex would be the values of the independent variables and the value of the function. For a function of two variables x and y, each complex point would have the coordinates \(x_1, y_1, f(x_1, y_1)\) as shown in Fig. 2.3.

\[f(x, y)\]

\[f(x_1, y_1)\]

\[x, y\]

\[x_1, y_1, f(x_1, y_1)\]

Figure 2.3. Coordinates of the Vertices of a Complex

---

The constraints were computed by a subroutine written especially for the test program. They included ranges of variables and relationships between the variables. Examples of the latter constraints include the relationship between the semi-major axis and the semi-latus rectum of the orbit (shown in Fig. 2.1) and the valid range of the VALUE parameter for different values of the MODE parameter (shown in Table 2.3).

An input parameter selects which independent variables are to be varied by the search algorithm. This was used in the experiment to vary only three of the independent variables in one test and all of the independent variables in the other test.

The termination condition for the search routine is determined by another input parameter. This parameter is a maximum function value which when found, reinitializes the search. When a set of input values has been found which causes this number of assertion violations, the maximum function value is increased by one and the search is begun again for this new value. If the new maximum value is not found, then the algorithm continues searching until one-hundred tests of the test program have been run.

After constructing the complex, the search routine finds the worst point (minimum function value over all points in the complex) and tries to replace it with a point with a larger function value (assuming the maximum of the function is being sought). It does this by applying the operations of reflection, expansion, centroid substitution, contraction, shrinkage, and rotation to the complex in that order. In order to illustrate each of these operations, Fig. 2.4 shows the effect of each of these operations on a triangle. In "reflection" the new point is found by reflecting the old point through the centroid of the complex.

Note that the "test number" column in Table 1.3 refers to the number of tests or runs of the test program (ORBP), not the search program. One run of the search program corresponds to at least 100 runs of the test program.
Figure 2.4. Complex Transformations
That is, the new point and the old point lie on a line through the centroid. The new point and the old point are each the same distance from the centroid. In "expansion," the distance of the new point from the centroid is greater than the distance from the old point to the centroid. In centroid substitution, the worst point is replaced by the centroid of the complex. "Contraction" reduces the distance from the new point to the centroid to be less than the distance from the old point to the centroid. "Shrinkage," instead of reflecting through the centroid of the complex, uses the point defined by the largest function value as the reflection point. Finally, "rotation" rotates the complex about the centroid in order to locate a new point. The cycle of operations continues until a maximum value is found or one-hundred tests have been run.

2.2 TEST DRIVER

A test driver was written to interface the search routine with the test program and initialize the test. The test driver determines which testing technique will be used: grid, search varying three variables, or search varying all variables. It initializes the values of all variables needed to conduct the test and reads in the basic set of orbital parameters which are common to all tests. It reads the values of the variables to be varied and their ranges and, for the grid test, divides the ranges up into intervals and selects a set of values for each variable corresponding to this division. It also calculates the dependent orbital parameters (semi-latus rectum and period divided by $2\pi$) and runs the grid tests. The search routine itself runs the search tests.

Other routines detect when assertions are violated, count the number of assertions violated in each of the tests, print a table of the assertions violated by test and record and print other information. The test program runs with other routines from the TRAID library which it uses to perform certain computations and input, output and formatting operations.
Assertions are statements added to a program to describe the intent of the program, the relationships which must hold between the variables in the program, the rules by which the variables can be accessed, and other information about the program which cannot be expressed in the programming language. In short, assertions are a way in which to state a program's specifications. They are useful in program verification, in consistency checking, and for reporting unexpected behavior while the program is being tested.

When assertions are translated into executable code by a compiler or preprocessor, they are called "executable assertions." Executable assertions placed at the beginning of the program are called "initial assertions," those placed at the end of the program are called "final assertions," and those placed within the program are called "intermediate assertions." Initial assertions describe the conditions that must be satisfied when the program is entered. These conditions can be the values of certain variables, their ranges, or the relationship between the value of one variable and the value of another (for example that X is greater than Y). Final assertions describe the result that the program is to compute—the range of values of the results and the relationships that must hold between any of the resultant variables. Intermediate assertions are used to describe the values that variables can assume and the relationship between these values and the values of other program variables at intermediate points in the program. They may also be used to specify the computational steps that a program must perform in response to the value of a particular logical expression.

Almost any condition or specification can be expressed using executable assertions. An executable assertion is a logical expression which, if evaluated to false, signals the violation of a specification. When the program is executed, the logical expression in an assertion is evaluated when the assertion is reached. If it is false, an error
message is printed, the assertion that was violated is recorded and a recovery routine (if specified in the assertion) is executed.

The assertions written for the test program, ORBP, describe three kinds of specifications: (1) the ranges of variables, (2) the relationships among variables, and (3) the history of the computation. For example, the assertions shown in Fig. 3.1 define the range of the parameter VALUE when it is interpreted as the angle between a point and the perigee. An example of the second type of assertion is shown in Fig. 3.2. Here VALUE is interpreted as the radius of the orbit. Therefore, its value must have a particular relation to the value of the semi-major axis, \( A \), and the eccentricity of the orbit, \( E \). The final type of assertion is used to keep track of the iterative computation of the angle from perigee when VALUE is interpreted as the time at which a point on the orbit is reached. The computation proceeds in two different ways depending on whether the number of iterations is even or odd. The code segment which performs this computation is shown in Fig. 3.3. The computation is limited in the number of iterations it is to perform. This is verified by adding the variable MTRY to the code to count the number of iterations and an assertion to test its value. This also helps identify errors which cause the computation to be performed out of sequence.

The assertions for the test program were organized in the following way. Initial assertions were gathered together in a logical function INPCHK which was invoked by the initial assertion

\[
\text{INITIAL ( INPCHK(MODE, VALU, ORBEL, STATE) )}
\]

which is the first assertion in the test program. This assertion shows that assertions can contain calls to logical functions, that is functions whose value evaluates to true or false. INPCHK contains assertions which check the ranges of the input variables to ORBP, verify the
ASSERT (VALUE .GE. 0.0)

ASSERT (VALUE .LE. TWOPI)

Figure 3.1. An Example of Range Assertions

ASSERT (VALUE .GE. (A * (1.0 - E)))

ASSERT (VALUE .LE. (A * (1.0 + E)))

Figure 3.2. An Example of Relationship Assertions

T = VALUE
EA1 = FM
NTRY = -1
41 CONTINUE
MTRY = NTRY
MTRY = NTRY + 1
IF (NTRY .EQ. 20) GO TO 250
EA = FM + E * SIN (EA1)
IF (ABS (EA1-EA) .LE. EMISS) GO TO 42
IF (MOD (NTRY,2) .EQ. 1) 45, 46
45 CONTINUE
EA1 = EA2 - (EA1-EA2)**2/(EA+EA2-2.*EA1)
ASSERT ( MTRY .LT. NTRY )
GO TO 41
46 EA2 = EA1
EA1 = EA
ASSERT (MTRY .LT. NTRY )
GO TO 41

Figure 3.3. An Example of History Assertions
relationships that must hold among these variables and verifies that the orbit defined by the orbital element vector is an ellipse.

The output assertions for ORBP were written in the same way. A logical function OUTCHK was written which was invoked by the assertion

FINAL ( OUTCHK(MODE, VALU, ORBEL, STATE) )

just before ORBP was exited. The function OUTCHK checked the output of the test program by comparing the representation of the orbit in terms of the state vector which was calculated, to the representation of the orbit as input to ORBP in the orbital element vector. It does this by recalculating the orbital element vector from the state representation of the point on the orbit. The code and assertions for OUTCHK are shown in Appendix A.

Other assertions were added directly to the test program to check the ranges of variables, the relationships between their values and the order of the computation. These assertions were derived from documentation provided with the program and from equations from the theory of orbital mechanics. The listings of these three programs are included in Appendix A.

The assertions for ORBP were not all written at one time. In fact, the combination of existing assertions and the search algorithm made the creation of new assertions an iterative process. As more was learned about the behavior of the program through the testing process, better, more precise assertions could be written about it.

Assertions were first written from information gained by reading the program and its documentation and by studying the equations of orbital mechanics. However, the first set of grid tests identified a number of errors which could not be detected using assertions and a
number of errors which were not detected by the assertions already in
the code. Therefore, the results from these tests were used to write
more precise assertions which could detect these errors. No new
assertions were added to the code after the first set of grid tests
although a number of assertions were changed. This is discussed more in
the results section below.
THE ERRORS

Errors were generated for the test program using a procedure developed by Brooks. A complete description of the method can be found in Cannon, Brooks and Meeson. The method uses error types and frequencies from a previous study to randomly select a set of errors to be "seeded" in the program. The error types from Project 5 of this study were used in the experiment. These error types or categories are shown in Table 4.1.

Not all of the categories were chosen for use in the experiment. Operation errors, other errors, documentation errors, and problem report rejection errors were not included because they did not include errors which were detectable while running the program. The experiment was specifically concerned with detecting run-time errors. Data input errors and data output errors were not included because the test program does not include any input or output statements of any consequence other than error messages. Data definition errors (which have to do with subscript referencing) were not included since explicit, constant subscripts were used to access arrays in the test program. Finally, data base errors were not included since the test program does not access a defined data base.

The remaining categories (computational errors, logic errors, data handling errors, and interface errors) were used to generate errors for ORBP. Table 4.2 shows (1) the percent of errors found in each category by the original study, (2) the percent of errors in each category when only these categories are considered, (3) the number of errors and the percent of errors in each category which were used in the study, and (4)

2Thayer et al., op. cit.
<table>
<thead>
<tr>
<th>Project 5 Error Categories</th>
<th>Applicable to Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_000 Computational Errors</td>
<td>✓</td>
</tr>
<tr>
<td>A_100 Incorrect operand in equation</td>
<td>✓</td>
</tr>
<tr>
<td>A_200 Incorrect use of parenthesis</td>
<td>✓</td>
</tr>
<tr>
<td>A_300 Sign convention error</td>
<td>✓</td>
</tr>
<tr>
<td>A_400 Units or data conversion error</td>
<td>✓</td>
</tr>
<tr>
<td>A_500 Computation produces over/under flow</td>
<td>✓</td>
</tr>
<tr>
<td>A_600 Incorrect/inaccurate equation used/wrong sequence</td>
<td>✓</td>
</tr>
<tr>
<td>A_700 Precision loss due to mixed mode</td>
<td>✓</td>
</tr>
<tr>
<td>A_800 Missing computation</td>
<td>✓</td>
</tr>
<tr>
<td>A_900 Rounding or truncation error</td>
<td>✓</td>
</tr>
<tr>
<td>B_000 Logic Errors</td>
<td>✓</td>
</tr>
<tr>
<td>B_100 Incorrect operand in logical expression</td>
<td>✓</td>
</tr>
<tr>
<td>B_200 Logic activities out of sequence</td>
<td>✓</td>
</tr>
<tr>
<td>B_300 Wrong variable being checked</td>
<td>✓</td>
</tr>
<tr>
<td>B_400 Missing logic or condition tests</td>
<td>✓</td>
</tr>
<tr>
<td>B_500 Too many/few statements in loop</td>
<td>✓</td>
</tr>
<tr>
<td>B_600 Loop iterated incorrect number of times (including endless loop)</td>
<td>✓</td>
</tr>
<tr>
<td>B_700 Duplicate logic</td>
<td>✓</td>
</tr>
<tr>
<td>C_000 Data Input Errors</td>
<td>✓</td>
</tr>
<tr>
<td>C_100 Invalid input read from correct data file</td>
<td>✓</td>
</tr>
<tr>
<td>C_200 Input read from incorrect data file</td>
<td>✓</td>
</tr>
<tr>
<td>C_300 Incorrect input format</td>
<td>✓</td>
</tr>
<tr>
<td>C_400 Incorrect format statement referenced</td>
<td>✓</td>
</tr>
<tr>
<td>C_500 End of file encountered prematurely</td>
<td>✓</td>
</tr>
<tr>
<td>C_600 End of file missing</td>
<td>✓</td>
</tr>
<tr>
<td>D_000 Data Handling Errors</td>
<td>✓</td>
</tr>
<tr>
<td>D_050 Data file not rewound before reading</td>
<td>✓</td>
</tr>
<tr>
<td>D_100 Data initialization not done</td>
<td>✓</td>
</tr>
<tr>
<td>D_200 Data initialization done improperly</td>
<td>✓</td>
</tr>
<tr>
<td>D_300 Variable used as a flag or index not set properly</td>
<td>✓</td>
</tr>
<tr>
<td>D_400 Variable referred to by the wrong name</td>
<td>✓</td>
</tr>
<tr>
<td>D_500 Bit manipulation done incorrectly</td>
<td>✓</td>
</tr>
<tr>
<td>D_600 Incorrect variable type</td>
<td>✓</td>
</tr>
<tr>
<td>D_700 Data packing/unpacking error</td>
<td>✓</td>
</tr>
<tr>
<td>D_800 Sort error</td>
<td>✓</td>
</tr>
<tr>
<td>D_900 Subscripting error</td>
<td>✓</td>
</tr>
</tbody>
</table>
### Table 4.1 (cont.)

<table>
<thead>
<tr>
<th>PROJECT 5 ERROR CATEGORIES</th>
<th>Applicable to Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>E_000</strong></td>
<td></td>
</tr>
<tr>
<td><strong>E_100</strong></td>
<td>DATA OUTPUT ERRORS</td>
</tr>
<tr>
<td><strong>E_200</strong></td>
<td>Data written on wrong file</td>
</tr>
<tr>
<td><strong>E_300</strong></td>
<td>Data written according to the wrong format statement</td>
</tr>
<tr>
<td><strong>E_400</strong></td>
<td>Data written in wrong format</td>
</tr>
<tr>
<td><strong>E_500</strong></td>
<td>Data written with wrong carriage control</td>
</tr>
<tr>
<td><strong>E_600</strong></td>
<td>Incomplete or missing output</td>
</tr>
<tr>
<td><strong>E_700</strong></td>
<td>Output field size too small</td>
</tr>
<tr>
<td><strong>E_800</strong></td>
<td>Line count or page eject problem</td>
</tr>
<tr>
<td><strong>E_900</strong></td>
<td>Output garbled or misleading</td>
</tr>
<tr>
<td><strong>F_000</strong></td>
<td>INTERFACE ERRORS</td>
</tr>
<tr>
<td><strong>F_100</strong></td>
<td>Wrong subroutine called</td>
</tr>
<tr>
<td><strong>F_200</strong></td>
<td>Call to subroutine not made or made in wrong place</td>
</tr>
<tr>
<td><strong>F_300</strong></td>
<td>Subroutine arguments not consistent in type, units, order, etc.</td>
</tr>
<tr>
<td><strong>F_400</strong></td>
<td>Subroutine called is nonexistent</td>
</tr>
<tr>
<td><strong>F_500</strong></td>
<td>Software/data base interface error</td>
</tr>
<tr>
<td><strong>F_600</strong></td>
<td>Software user interface error</td>
</tr>
<tr>
<td><strong>F_700</strong></td>
<td>Software/software interface error</td>
</tr>
<tr>
<td><strong>G_000</strong></td>
<td>DATA DEFINITION ERRORS</td>
</tr>
<tr>
<td><strong>G_100</strong></td>
<td>Data not properly defined/dimensioned</td>
</tr>
<tr>
<td><strong>G_200</strong></td>
<td>Data referenced out of bounds</td>
</tr>
<tr>
<td><strong>G_300</strong></td>
<td>Data being referenced at incorrect location</td>
</tr>
<tr>
<td><strong>G_400</strong></td>
<td>Data pointers not incremented properly</td>
</tr>
<tr>
<td><strong>H_000</strong></td>
<td>DATA BASE ERRORS</td>
</tr>
<tr>
<td><strong>H_100</strong></td>
<td>Data not initialized in data base</td>
</tr>
<tr>
<td><strong>H_200</strong></td>
<td>Data initialized to incorrect value</td>
</tr>
<tr>
<td><strong>H_300</strong></td>
<td>Data units are incorrect</td>
</tr>
<tr>
<td><strong>I_000</strong></td>
<td>OPERATION ERRORS</td>
</tr>
<tr>
<td><strong>I_100</strong></td>
<td>Operating system error (vendor supplied)</td>
</tr>
<tr>
<td><strong>I_200</strong></td>
<td>Hardware error</td>
</tr>
<tr>
<td><strong>I_300</strong></td>
<td>Operator error</td>
</tr>
<tr>
<td><strong>I_400</strong></td>
<td>Test execution error</td>
</tr>
<tr>
<td><strong>I_500</strong></td>
<td>User misunderstanding/error</td>
</tr>
<tr>
<td><strong>I_600</strong></td>
<td>Configuration control error</td>
</tr>
</tbody>
</table>
**Table 4.1 (cont.)**

<table>
<thead>
<tr>
<th>PROJECT 5 ERROR CATEGORIES</th>
<th>Applicable to Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>J_000 OTHER</strong></td>
<td></td>
</tr>
<tr>
<td>J_100 Time limit exceeded</td>
<td></td>
</tr>
<tr>
<td>J_200 Core storage limit exceeded</td>
<td></td>
</tr>
<tr>
<td>J_300 Output line limit exceeded</td>
<td></td>
</tr>
<tr>
<td>J_400 Compilation error</td>
<td></td>
</tr>
<tr>
<td>J_500 Code or design inefficient/not necessary</td>
<td></td>
</tr>
<tr>
<td>J_600 User/programmer requested enhancement</td>
<td></td>
</tr>
<tr>
<td>J_700 Design nonresponsive to requirements</td>
<td></td>
</tr>
<tr>
<td>J_800 Code delivery or redelivery</td>
<td></td>
</tr>
<tr>
<td>J_900 Software not compatible with project standards</td>
<td></td>
</tr>
<tr>
<td><strong>K_000 DOCUMENTATION ERRORS</strong></td>
<td></td>
</tr>
<tr>
<td>K_100 User manual</td>
<td></td>
</tr>
<tr>
<td>K_200 Interface specification</td>
<td></td>
</tr>
<tr>
<td>K_300 Design specification</td>
<td></td>
</tr>
<tr>
<td>K_400 Requirements specification</td>
<td></td>
</tr>
<tr>
<td>K_500 Test documentation</td>
<td></td>
</tr>
<tr>
<td><strong>X0000 PROBLEM REPORT REJECTION</strong></td>
<td></td>
</tr>
<tr>
<td>X0001 No problem</td>
<td></td>
</tr>
<tr>
<td>X0002 Void/withdrawn</td>
<td></td>
</tr>
<tr>
<td>X0003 Out of scope - not part of approved design</td>
<td></td>
</tr>
<tr>
<td>X0004 Duplicates another problem report</td>
<td></td>
</tr>
<tr>
<td>X0005 Deferred</td>
<td></td>
</tr>
<tr>
<td>Error Category</td>
<td>Study %</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
</tr>
<tr>
<td>A</td>
<td>12.1</td>
</tr>
<tr>
<td>B</td>
<td>24.5</td>
</tr>
<tr>
<td>D</td>
<td>11.0</td>
</tr>
<tr>
<td>F</td>
<td>7.0</td>
</tr>
<tr>
<td></td>
<td>54.6</td>
</tr>
</tbody>
</table>

**TABLE 4.2**
TYPES OF ERRORS USED IN THE EXPERIMENT
the number of errors and percent of errors in each category which were successfully detected by assertions (see results section).

In the original study, no attempt was made to match the error type or category to a specific statement type in the program. In generating errors for the experiment, statement types and other descriptive information about the test program were generated automatically using an automated program verification system, SQLAB. The statement types were then matched against errors using the method outlined below.

4.1 THE ERROR SEEDING METHOD

The errors were generated in the following way. First, each statement in the test program was classified by type. Then a table matching the error categories to statement types was constructed. This is shown in Table 4.3. The set of statement types found in the test program was then added to the error-category/statement-type table. This gave a list of available error sites in the test program with associated error categories. From this list of available error sites, potential error sites were randomly selected and matched with the error subcategories by a previously written computer program.

From the list of potential sites and associated error subcategories, errors were developed. The error site was first checked to be sure that the error sub-category was appropriate for the site. For example, if error type A200 (incorrect use of parenthesis) is selected as a subcategory, the statement must contain parentheses in order to include this error.

As each error was constructed, it was included in an "error packet" containing an error number, a comment which identified the error subcategory, and the code which altered the original code of the test program in order to produce the error. Since the test program was

<table>
<thead>
<tr>
<th>Test Software Property</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Computational</td>
<td>Logic</td>
<td>Input</td>
<td>Output</td>
<td>Data Handling</td>
<td>Interface</td>
<td>Data Definition</td>
<td>Data Base</td>
</tr>
<tr>
<td><strong>Statement</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assign</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backspace</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Call</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Common</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Computed GOTO</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Continue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dimension</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>DO</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ENTRY</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equivalence</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>EXTERNAL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FORMAT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FUNCTION</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>GOTO</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>IF end-of-file</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Three-branch IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Two-branch IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PRINT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PROGRAM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>READ</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>REAL</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>RETURN</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>REVERSE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>STOP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>SUBROUTINE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>WRITE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PARAMETER</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>** INVOCATION**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
stored on a program library maintained by CDC UPDATE\(^1\) (a batch source text editor), the error packets could easily be inserted into the test program. Figure 4.1 shows an example of an error packet.

Next the error packets were inserted into the test program and the program was compiled and run. This was done to insure that the errors were not detected by the FORTRAN compiler, the loader or the run-time error routines of the operating system. In this way, twenty-four errors were developed for use during the testing. Table 4.4 shows each of these errors by number, the error subcategory to which it belongs and a short description of the subcategory.

Seven of the errors generated were eliminated from the testing during the grid tests since they could not be detected using assertions. This is discussed in Sec. 6.1.

\*IDENT 13
\*DELETE ORBP.63
C A100
VALUE = VALU-RBODY

Figure 4.1. An Error Packet

\(^1\)UPDATE Reference Manual, Control Data Corporation, Arden Hills, Minn., 1975.
<table>
<thead>
<tr>
<th>Error Number</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A200</td>
<td>incorrect use of parenthesis</td>
</tr>
<tr>
<td>3</td>
<td>A300</td>
<td>sign convention error</td>
</tr>
<tr>
<td>8</td>
<td>A600</td>
<td>incorrect/inaccurate equation used/wrong sequence</td>
</tr>
<tr>
<td>13</td>
<td>A100</td>
<td>incorrect operand in equation</td>
</tr>
<tr>
<td>14</td>
<td>A800</td>
<td>missing computation</td>
</tr>
<tr>
<td>28</td>
<td>B400</td>
<td>missing logic or condition tests</td>
</tr>
<tr>
<td>31</td>
<td>B400</td>
<td>missing logic or condition tests</td>
</tr>
<tr>
<td>36</td>
<td>B200</td>
<td>logic activities out of sequence</td>
</tr>
<tr>
<td>37</td>
<td>B200</td>
<td>logic activities out of sequence</td>
</tr>
<tr>
<td>40</td>
<td>B300</td>
<td>wrong variable being checked</td>
</tr>
<tr>
<td>41</td>
<td>D200</td>
<td>data initialization done improperly</td>
</tr>
<tr>
<td>46</td>
<td>D100</td>
<td>data initialization not done</td>
</tr>
<tr>
<td>47</td>
<td>D100</td>
<td>data initialization not done</td>
</tr>
<tr>
<td>48</td>
<td>D400</td>
<td>variable referred to by the wrong name</td>
</tr>
<tr>
<td>52</td>
<td>D500</td>
<td>incorrect variable type</td>
</tr>
<tr>
<td>54</td>
<td>D600</td>
<td>incorrect variable type</td>
</tr>
<tr>
<td>55</td>
<td>D600</td>
<td>incorrect variable type</td>
</tr>
<tr>
<td>56</td>
<td>D400</td>
<td>variable referred to by the wrong name</td>
</tr>
<tr>
<td>57</td>
<td>D300</td>
<td>variable used as a flag or index not set properly</td>
</tr>
<tr>
<td>62</td>
<td>F100</td>
<td>wrong subroutine called</td>
</tr>
<tr>
<td>64</td>
<td>F100</td>
<td>wrong subroutine called</td>
</tr>
<tr>
<td>67</td>
<td>F700</td>
<td>software/software interface error</td>
</tr>
<tr>
<td>74</td>
<td>F200</td>
<td>call to subroutine not made or made in wrong place</td>
</tr>
<tr>
<td>77</td>
<td>F700</td>
<td>software/software interface error</td>
</tr>
</tbody>
</table>
THE EXPERIMENT

The errors were inserted into the test program one at a time. First, grid tests were performed to identify any errors which could not be detected by assertions or errors for which other assertions had to be written. After the former errors were eliminated from consideration and assertions were added to the code to detect the latter, the grid tests were performed again. The results from these tests were used as a baseline by which to evaluate the search technique. A set of assertions which were violated when the test program was run using the grid test method was associated with each error. After the grid tests were run, the search algorithm was used to test the program by varying only three of the maximum of eight variable parameters. Finally, the search algorithm was allowed to vary all of the parameters.

Recall that of the eight parameters in the orbital element vector, only six of these are independent. The independent variables are: (1) longitude of the ascending node, (2) inclination of the orbit plane, (3) argument (angle) of the perigee, (4) eccentricity, (5) time at perigee, and (6) semi-major axis. These parameters along with MODE and VALUE were the parameters which could be varied by the test driver. For each of the tests, a standard orbit was used as a basic test case. The parameters of the orbit are shown in Table 5.1. The parameters which were not being varied in a test remained fixed at these values.

5.1 GRID TESTS

For the grid tests, three variables were varied, MODE, VALUE, and the eccentricity of the orbit. The tests were performed in the following way. The standard orbit was input to the test driver program. The test driver then varied the values of the parameters and ran tests of ORBP. The data collection routines recorded the number of assertions violated in each test along with the values of the input variables.
The parameter values were varied in the following way. The value of the eccentricity of the orbit was varied from 0.1 to 0.9 in steps of 0.2. (The range and step size of any variable can be varied by the test driver program.) The value of the mode was then varied from 0 to 5. For each value of MODE, the corresponding VALUE parameter was varied over its range from minimum to maximum such that eleven VALUES were generated for each value of MODE. The range of the VALUE parameter for each value of the MODE parameter is shown in Table 2.3. In this way, a coarse "grid" was drawn over the input space of the program for three variables. The values of the variables determine points in the grid and were used as input values to the program during this series of tests.

For error number 52, the time at perigee had to be varied instead of the eccentricity in order for the assertions to detect the error. This parameter was varied from 0 to the period in order to generate eleven test values.
5.2 SEARCH VARYING THREE PARAMETERS

In the second part of the experiment the search routine was used to detect the errors. Again the standard orbit was used as a basis for the testing. It was input to the test driver and the search routine was allowed to vary the values of MODE, VALUE and the eccentricity of the orbit (time at perigee in the case of error 52) in order to locate the error in the test program. All other input parameters to ORBP remained constant. The testing was done by inserting the errors in the test program one at a time. For each error, the assertions violated were recorded along with the values of the parameters.

The search routine was allowed to run until it found the number of assertion violations preset by an input parameter. When this number of assertion violations was detected, it was increased by one and the search algorithm tried to locate a combination of input values which caused the new number of assertions to be violated. In this way, the search algorithm was directed to locate values of the input parameters which caused the maximum number of assertions to be violated. The search routine stopped if it had not located this number of errors in one hundred more tests.

5.3 SEARCH VARYING ALL PARAMETERS

For the final stage of the experiment, the search routine was allowed to vary all of the input parameters in order to locate assertion violations. Again, the standard orbit was used as a starting test case. In addition to this set of input data, the search routine chose random values for the parameters until eleven test cases were identified. A test case consisted of the orbital element vector and the MODE and VALUE parameters. This is one more test case than the number of variables in the input space of the test program and is the number of function values required to construct the complex. The number of assertions violated for each test case was determined by running the test program.
The search continued by varying the input parameters according to the search algorithm until a preset number of assertions was violated. As in the previous search tests, when this occurred the number was increased by one and the search continued in order to locate a new test case which violated this new number of assertions. If the new number of assertions were not violated in any test after one hundred tests, the search was stopped. Each one of the errors was tested in this way.

Figures 5.1 to 5.5 show some of the output produced by the search program when run with error number 13. Figure 5.1 shows a template for interpreting the output. Error information produced in response to the violation of an assertion appears first, as shown by error 9 in Fig. 5.2; or there may be none, as in test 6. Next, the test number and the action performed by the search routine in selecting the new point is printed. The possible search actions are shown in Table 5.2. The values of MODE, the orbital parameters and VALUE are then printed. Finally, the “performance value,” the number of assertions violated in the test is printed.

Figure 5.2 shows the tests used to initialize the complex, that is those which determine the vertices of the complex by obtaining eleven values of the error function. Tests 7 and 9 have already caused assertions to be violated. Note that all the orbital elements, MODE and VALUE are being varied.

Figure 5.3 shows tests in the middle of the testing cycle. The search routine is applying appropriate transformations, rotation, reflection, centroid substitution and contraction in order to remove the worst point from the complex and locate a point where the maximum number of assertions are violated. Note that not all search actions are tried (e.g., expansion, shrinkage), since other parameters of the complex and error function determine which transformations are applied. In Fig. 5.3 tests 44, 45 and 47 located new input values which caused assertions to be violated, whereas test 46 did not.
Error information from assertions

Test Number Search Action
Worst Point Orbital Elements

Mode Longitude of Inclination Angle of Semi-
Ascending Node of the Orbit Perigees Rectum
(Radians) Plane (Radians) (Radians)

Eccentricity Time at Period/2\pi Semi-
Perigee Value
(Seconds/ major
(Radians) Axis (Meters)

Performance Value = Number of assertion violations

Figure 5.1. Search Program Output Template

TABLE 5.2
POSSIBLE SEARCH ACTIONS

<table>
<thead>
<tr>
<th>Search Action</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL</td>
<td>Initialize Complex</td>
</tr>
<tr>
<td>REFLECT</td>
<td>Reflection</td>
</tr>
<tr>
<td>EXPAND</td>
<td>Expansion</td>
</tr>
<tr>
<td>CENTROID</td>
<td>Centroid Substitution</td>
</tr>
<tr>
<td>CONTRACT</td>
<td>Contraction</td>
</tr>
<tr>
<td>SHRINK</td>
<td>Shrinkage</td>
</tr>
<tr>
<td>ROTATE</td>
<td>Rotation</td>
</tr>
<tr>
<td>RE-INITIAL</td>
<td>Re-initialize Complex</td>
</tr>
</tbody>
</table>

5-5
<table>
<thead>
<tr>
<th>Test</th>
<th>Initial</th>
<th>Worst Point</th>
<th>Performance Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>2.228417351</td>
<td>2.228417351</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4610.762172</td>
<td>1517.131149</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5.619764999</td>
<td>730198.092</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>5.179884825</td>
<td>5.179884825</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18085.35416</td>
<td>18085.35416</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.408350499</td>
<td>3.408350499</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>4.219723780</td>
<td>4.219723780</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1409.391451</td>
<td>9251020.763</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>0.697940621</td>
<td>0.697940621</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2397.998640</td>
<td>2397.998640</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>3.285706144</td>
<td>3.285706144</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2341.114231</td>
<td>2341.114231</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>5.689950305</td>
<td>5.689950305</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3561.680710</td>
<td>3561.680710</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>7.720035681</td>
<td>7.720035681</td>
</tr>
<tr>
<td></td>
<td></td>
<td>62045.79446</td>
<td>62045.79446</td>
</tr>
</tbody>
</table>

**ORB has replaced impossible radius with perigee**

<table>
<thead>
<tr>
<th>Test</th>
<th>Initial</th>
<th>Worst Point</th>
<th>Performance Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td>-406816.74491795097</td>
<td>2477545.65879538655</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2286363.2550202612</td>
<td>6375180.0000000000</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>-5216513.50552853860</td>
<td>6375180.0000000000</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>5.689950305</td>
<td>5.689950305</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3561.680710</td>
<td>3561.680710</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>7.720035681</td>
<td>7.720035681</td>
</tr>
<tr>
<td></td>
<td></td>
<td>62045.79446</td>
<td>62045.79446</td>
</tr>
</tbody>
</table>

**ORB has replaced impossible radius with perigee**

<table>
<thead>
<tr>
<th>Test</th>
<th>Initial</th>
<th>Worst Point</th>
<th>Performance Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td>-406816.74491795097</td>
<td>2477545.65879538655</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2286363.2550202612</td>
<td>6375180.0000000000</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>-5216513.50552853860</td>
<td>6375180.0000000000</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>5.689950305</td>
<td>5.689950305</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3561.680710</td>
<td>3561.680710</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>7.720035681</td>
<td>7.720035681</td>
</tr>
<tr>
<td></td>
<td></td>
<td>62045.79446</td>
<td>62045.79446</td>
</tr>
</tbody>
</table>
ORB P HAS REPLACED IMPOSSIBLE RADIUS WITH PERIGEE -

<table>
<thead>
<tr>
<th>TEST</th>
<th>WORST POINT</th>
<th>PERFORMANCE VALUE</th>
<th>VALUE</th>
<th>BODY</th>
<th>RODY</th>
<th>R = 3635460.63057047321</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>3.966273291</td>
<td>0.606307614</td>
<td>4.763729021</td>
<td>6170490.267</td>
<td>4044234.171</td>
<td></td>
</tr>
</tbody>
</table>

ORB P HAS REPLACED IMPOSSIBLE RADIUS WITH PERIGEE -

<table>
<thead>
<tr>
<th>TEST</th>
<th>WORST POINT</th>
<th>PERFORMANCE VALUE</th>
<th>VALUE</th>
<th>BODY</th>
<th>RODY</th>
<th>R = 2815401.9209545096</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td>5.642159314</td>
<td>3.04470996</td>
<td>6.283185308</td>
<td>4570691.766</td>
<td>0.0</td>
<td></td>
</tr>
</tbody>
</table>

ORB P HAS REPLACED IMPOSSIBLE RADIUS WITH PERIGEE -

<table>
<thead>
<tr>
<th>TEST</th>
<th>WORST POINT</th>
<th>PERFORMANCE VALUE</th>
<th>VALUE</th>
<th>BODY</th>
<th>RODY</th>
<th>R = 1051749.44838433647</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
<td>2.293169953</td>
<td>2.106425797</td>
<td>5.016457085</td>
<td>8438408.516</td>
<td>4533190.345</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3. Intermediate Stage of Search Testing
**ORBPH AS REPLACED IMPOSSIBLE RADIUS WITH PERIGEE -**

<table>
<thead>
<tr>
<th>Test</th>
<th>PT</th>
<th>Worst Point</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Value</th>
<th>Rbody</th>
</tr>
</thead>
<tbody>
<tr>
<td>94</td>
<td>R</td>
<td>4,00000000</td>
<td>2,272290885</td>
<td>1,428911545</td>
<td>3,575033478</td>
<td>13179367,00</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.495453241</td>
<td>19425.43816</td>
<td>3656.918571</td>
<td>17468088.60</td>
<td>11674092.79</td>
<td></td>
</tr>
<tr>
<td>95</td>
<td>R</td>
<td>6110497.79315781593</td>
<td>2,837951677</td>
<td>6,283105308</td>
<td>3954322.723</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.843305149</td>
<td>15942,72284</td>
<td>2937.363146</td>
<td>13690685.60</td>
<td>10860857.79</td>
<td></td>
</tr>
<tr>
<td>96</td>
<td>C</td>
<td>2,027068.40747164011</td>
<td>2,426831755</td>
<td>1,579551477</td>
<td>4,021620299</td>
<td>13147715,03</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5697853136</td>
<td>23299,68466</td>
<td>4298,984417</td>
<td>19505985,64</td>
<td>11115483,73</td>
<td></td>
</tr>
<tr>
<td>97</td>
<td>C</td>
<td>2237810,76380419731</td>
<td>2,208751577</td>
<td>5,15240280</td>
<td>8320417,305</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.704045940</td>
<td>19621,20375</td>
<td>3356,652825</td>
<td>16498310,62</td>
<td>14988170,76</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,00000000</td>
<td>1,00000000</td>
<td>1,00000000</td>
<td>1,00000000</td>
<td>1,00000000</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.4. Later Stage of Search Testing**
## Figure 5.5. Summary of Search Testing for Error 13

### Table 5.5.1: Summary of Search Testing for Error 13

<table>
<thead>
<tr>
<th>#RUN</th>
<th>INPUT1</th>
<th>INPUT2</th>
<th>#FALSE ASSERTION</th>
<th>#DIFFERENT MODE ASSERTION</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>.7526</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>.6048</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>.2700</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>.9000</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>24</td>
<td>.2699</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>.3679</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>30</td>
<td>.2910</td>
<td>0.00</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>34</td>
<td>.7346</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>35</td>
<td>.1852</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>37</td>
<td>.3555</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>44</td>
<td>.6973</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>45</td>
<td>.6235</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>47</td>
<td>.5051</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>49</td>
<td>.9000</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>51</td>
<td>.7234</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>53</td>
<td>.9000</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>55</td>
<td>.7234</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>63</td>
<td>.7053</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>65</td>
<td>.6261</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>73</td>
<td>.7474</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>75</td>
<td>.6471</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>83</td>
<td>.6071</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>85</td>
<td>.6769</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>86</td>
<td>.1774</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>87</td>
<td>.9000</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>89</td>
<td>.7228</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>90</td>
<td>.9000</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>91</td>
<td>.1557</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>92</td>
<td>.5503</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>93</td>
<td>.3530</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>94</td>
<td>.4955</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>95</td>
<td>.8433</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>96</td>
<td>.5648</td>
<td>0.00</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>97</td>
<td>.7040</td>
<td>0.00</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>98</td>
<td>.6108</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>99</td>
<td>.2554</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>100</td>
<td>.5485</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>101</td>
<td>.4019</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>102</td>
<td>.1000</td>
<td>0.00</td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

**MODULE STAT # FAILURE**

| ORBGP 109 ASSERT 34 | OUTCHK 142 ASSERT 38 |

- How many runs each assertion failed in 102 runs

Figure 5.5. Summary of Search Testing for Error 13
Figure 5.4 shows a later stage in the search testing. Here almost every test results in some assertions being violated.

Figure 5.5 is a summary of the results of the search testing for error number 13. Only the tests in which assertions were violated are shown. The summary shows for each test (1) the number of assertions violated, (2) the number of different assertions violated, and (3) the values of MODE and VALUE. (The INPUT1 and INPUT2 columns are used for the grid testing.) The figure also shows the progress of the search routine during the testing. At the beginning of the testing, assertions were violated only every few tests. At the end of the testing, almost every test resulted in assertions being violated. Finally, the location of the assertions that were violated and the number of times that they were violated are printed.
RESULTS OF THE EXPERIMENT

Four major results were found from the experiments: (1) the original set of grid tests found that a number of the errors could not be detected through the use of assertions, (2) the search tests located assertion violations for two errors which the grid tests did not discover but there were two errors for which the grid tests found assertion violations where the search tests did not, (3) the search tests were more efficient than the grid tests in locating assertion violations, and (4) the search tests discovered a number of boundary conditions which caused assertion violations.

6.1 ERROR DETECTION USING ASSERTIONS

Of the twenty-four errors originally used for the testing, only nine (37.5%) of these errors were detected by the first assertions placed in the code. By adding more assertions to the test program, eight more errors were detected (33.3%). The remaining seven errors (29.2%) could not be detected by placing assertions in ORBP. Table 6.1 lists these errors along with their categories, short descriptions and the reason they could not be detected by assertions.

Two of the errors could not be detected by the test method because they occurred only after another error had occurred first. Another error occurred only if values of the input parameters were out of range, a possible source of error, but not one considered in the experiment. Three of the errors could be detected by static analysis techniques such as variable initialization checks, parameter checks and cross-references but are less easily detected using assertions. These errors cannot be easily caught by assertions because of the limits placed on the assertions by the semantics of the programming language. For example, there is no way to state in an assertion that a variable has been initialized to a particular value other than by stating that the variable has that value. If the value happens to be zero, and the compiler assigns this value to the variable automatically, then there is
### TABLE 6.1

**ERRORS NOT DETECTED BY ASSERTIONS**

<table>
<thead>
<tr>
<th>Error</th>
<th>Category</th>
<th>Description</th>
<th>Reason For Not Being Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>A600</td>
<td>Variables assigned values in incorrect order</td>
<td>An error must occur for this section of code to be executed</td>
</tr>
<tr>
<td>36</td>
<td>B200</td>
<td>Test and branch statement deleted</td>
<td>Checks for out of range input values</td>
</tr>
<tr>
<td>40</td>
<td>B300</td>
<td>Variable name mispelled in computed goto</td>
<td>Difficult to state an assertion for this error</td>
</tr>
<tr>
<td>46</td>
<td>D100</td>
<td>Data statement deleted</td>
<td>Fortran compiler initializes all variables to zero</td>
</tr>
<tr>
<td>55</td>
<td>D600</td>
<td>Real variable declared as integer</td>
<td>Difficult to state an assertion for this error</td>
</tr>
<tr>
<td>62</td>
<td>F100</td>
<td>Subroutine call out of place</td>
<td>An error must occur for the section of code to be executed</td>
</tr>
<tr>
<td>77</td>
<td>F700</td>
<td>Wrong number of arguments in subroutine call</td>
<td>Difficult to state an assertion for this error</td>
</tr>
</tbody>
</table>
no way to write an assertion that states that the variable was initialized. Stated another way, we can write an assertion which states that a variable is equal to a certain value, but not that it has been initialized. Similarly, it is difficult for an assertion to state that a subroutine call has a certain number of parameters, or that a variable is spelled correctly. Since assertions are written using the constructs of the programming language, they cannot state things about the program that cannot be stated in the programming language.

The other error which was not used caused a run-time error to occur in a library routine. This could be detected in the library routine by an assertion, but not in the test routine. Again, the specific error indicates the limited power of assertions. In this case, a REAL variable was declared as INTEGER. There is no way using assertions to state that the type of a variable is REAL. Again, this error might have been located by a static analysis check for invalid parameter types.

6.2 EFFECTIVENESS OF THE SEARCH TECHNIQUES

For most errors, the search technique (using three parameters and all parameters) identified the same assertion violations as the grid testing technique. In four cases (errors 14, 28, 47, and 74), however, this did not occur. For two of the errors (28 and 47), the search technique did not identify as many assertion violations as the grid technique. In the other two cases (errors 14 and 74), the search technique identified assertion violations that were not detected in the grid tests.

In error 28, a statement is deleted which tests for a zero divisor. The sequence of code and the assertion that is violated is shown in Fig. 6.1. The statement

\[
\text{IF}(X2 \leq 0) \text{ GOTO 48}
\]


```fortran
42 X2 = 1. + COS (EA)
Q = PI
IF (X2 .LE.0.) GO TO 48
ASSERT ( X2 .GT. 0.0 )
X1 = SQRT ( (1. + E) / (1. - E) ) * SIN (EA)
Q = 2. * ATAN2 (X1, X2)
48 CONTINUE
```

Figure 6.1. Error 28

which was deleted to cause the error, is used to prevent a zero divisor in the call to the arctangent subroutine. The documentation with this system support routine states that the sum of the parameters \(X_1\) and \(X_2\) squared must not be equal to zero, and that the arctangent of \(X_1\) divided by \(X_2\) is computed (see Fig. 6.2). An assertion violation is detected by the grid test for this error but by neither of the search tests (three-parameter or all-parameter). The reason for this is that the grid test uses values for the time parameter which locate the point on the orbit as being at apogee whereas neither of the search tests used this value. For the apogee point, the value of the angle \(EA\) becomes equal to PI and the value of \(X2\) becomes 0 (see Fig. 6.3). No run-time error was detected by the arctangent routine for this value.

Error 47 is the deletion of a data statement. This statement initializes the value of the error tolerance for the iterative computation of the angle from perigee when the \(VALUE\) parameter indicates time. The statement which this effects and the assertions violated are shown in Fig. 6.4. Since the FORTRAN compiler initializes all variables to
\[ Y = \text{ATAN2} \left( X_1, X_2 \right) \]

Function: Computes arctangent of \( X_1/X_2 \)

Constraint: \( X_1^2 + X_2^2 \neq 0 \)

Figure 6.2. Arctangent Function

\[
\begin{align*}
X_2 &= 1. + \cos (EA) \\
\text{for } EA &= \pi : \\
X_2 &= 1. + \cos (\pi) \\
X_2 &= 1. + (-1) \\
X_2 &= 0
\end{align*}
\]

Figure 6.3. Value of Divisor at \( \pi \)

Data statement deleted

\[
\text{DATA EMISS / 1.E-7 /}
\]

Loop exit statement

\[
\text{IF ( ABS (EA1-EA) .LE. EMISS) GO TO 42}
\]

Assertions violated

\[
\begin{align*}
\text{ASSERT ( ABS (EA-EA1) / (EA1-EA2) .LT. 1.0) } \\
\text{ASSERT ( ABS (EA+EA2 - 2.0 *EA1) .GT. 0.0) }
\end{align*}
\]

Figure 6.4. Error 47
zero, this variable is by default initialized to zero also. This changes the termination condition of the loop so that it only ends if the value of EA equals the value of EAI. Again, both assertions will be violated only if the computation is being performed for a particular point on the orbit, apogee. In this case, both the grid test and the search using three-parameters found values of the input parameters which violated both assertions. The all-parameter search did not locate a value which violated the second assertion.

Error 14 is caused by the deletion of a statement. In this case however, the three-parameter search found one more assertion violation (2) than the grid test technique (1), and the all-parameter search found one more assertion violation than the three-parameter search (3). Figure 6.5 shows the sequence of statements and the assertions associated with this error. By removing the statement

\[ Q = \text{ACOS} (QPRIME) \]

error 14 causes the value of Q to be undefined. This error is detected by the assertions in the OUTCHK routine when the orbits described by the

---

Code Segment

\[
QPRIME = \text{ADIV}(P-R, R*E) \\
Q = \text{ACOS} (QPRIME)
\]

Assertions violated

\[
\text{ASSERT ( RELERR(A, ORBIT(9) ) .GE. - EPS) }
\]
\[
\text{ASSERT (RELERR(A, ORBIT(9) ) .LE. EPS) }
\]
\[
\text{ASSERT (OE(4) .LE. TWOPI + TWOPI) }
\]

Figure 6.5. Error 14
initial orbital parameters and the state vector representation are com-
pared. The grid test technique located values in the input space which
causess the first assertion to be violated. That is, the semi-major axes
of the two orbits did not agree. The three parameter search located
other values for which this was also true and caused the second asser-
tion to be violated. The all-parameter search, since it also varied the
value of the argument of the perigee in the orbit element vector, was
able to locate values in the input space which caused the third asser-
tion to be violated.

Error 74 is caused by the deletion of a call to a subroutine which
copies the input orbital element vector to another array. Assertions
were written to compare the values of these two arrays after the point
of the call in the code. The grid test technique and the three-param-
eter search detected assertion violations for all of the variables in
the original orbital element vector except one. This value was equal to 0 in the
original orbital element description and was not varied by the two test
methods. Since the FORTRAN compiler initialized the values of the
receiving array to zero, the fact that this variable was not copied was
not detected. When the all-parameter search was allowed to vary this
parameter, the assertion violation for this parameter occurred also.
Figure 6.6 shows the code and assertions for this error.

Table 6.2 summarizes the results for these errors, showing the
error number and the number of assertion violations detected by each of
the three testing methods.

6.3 EFFICIENCY OF THE SEARCH METHOD

Data which could be used to measure the efficiency of the search
methods relative to the grid testing method were not collected during
the experiment. However, a rough estimate of the relative efficiencies
of the two methods is shown in Table 6.3. Except in the case of error
52, in which it took 683 tests to perform the entire grid test, all of
the errors required 317 tests. Table 6.3 shows for each error, the
Statement

CALL XMIT (8, ORBEL(2), OE(2))

Assertions violated

ASSERT ( OE(2) .EQ. ORBEL(2) )
ASSERT ( OE(3) .EQ. ORBEL(3) )
ASSERT ( OE(4) .EQ. ORBEL(4) )
ASSERT ( OE(5) .EQ. ORBEL(5) )
ASSERT ( OE(6) .EQ. ORBEL(6) )
ASSERT ( OE(7) .EQ. ORBEL(7) )
ASSERT ( OE(8) .EQ. ORBEL(8) )
ASSERT ( OE(9) .EQ. ORBEL(9) )

Figure 6.6. Error 74

---

TABLE 6.2
ASSERTION VIOLATIONS DETECTED BY EACH TESTING METHOD

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Number of Invalid Assertions Detected by Testing Technique</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3-Variable Search</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>47</td>
<td>2</td>
</tr>
<tr>
<td>74</td>
<td>7</td>
</tr>
<tr>
<td>Error Number</td>
<td>Test Number of First Assertion Violation</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
</tr>
<tr>
<td>28</td>
<td>*</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
</tr>
<tr>
<td>37</td>
<td>5</td>
</tr>
<tr>
<td>41</td>
<td>3</td>
</tr>
<tr>
<td>47</td>
<td>57</td>
</tr>
<tr>
<td>48</td>
<td>3</td>
</tr>
<tr>
<td>52</td>
<td>3</td>
</tr>
<tr>
<td>54</td>
<td>3</td>
</tr>
<tr>
<td>56</td>
<td>5</td>
</tr>
<tr>
<td>57</td>
<td>7</td>
</tr>
<tr>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>67</td>
<td>5</td>
</tr>
<tr>
<td>74</td>
<td>2</td>
</tr>
</tbody>
</table>

*No assertion violations detected*
number of the test in which the all-parameter search testing technique detected the first assertion violation. This table shows that for 15 of the 17 errors the all-parameter search technique detected the first assertion violation on or before the seventh test.

6.4 SPECIAL CASES

During the experiment a number of assertions were revised. These assertions were changed because of the results from the search tests. In three cases, the search technique discovered input values for which assertions were violated. In each case it was later discovered that the assertions were incorrect. These special values were not discovered by the grid testing technique. This illustrates one important result of the testing method, that the development of assertions and the testing occur as a coupled iterative process. The original assertions help to locate errors, the search technique locates new assertion violations which are either errors in the software or in the assertions. Throughout the testing process, the accuracy of the assertions was improved along with the ability to detect errors.

The first assertion which was discovered as being incorrect was one which checks the value of the angle from perigee (FM) computed from the time (VALUE), time at perigee (TP) and period (PP). The code, the original assertion and the corrected assertion are shown in Fig. 6.7. This assertion violation was found by the all-parameter search by varying the time at perigee (TP). This caused the value of the angle from perigee to become negative. The time at perigee had not been varied by either of the other two test methods.

The second incorrect assertion was found by the three-parameter search method. This assertion violation was due to the nature of the orbital descriptions and the inherent inaccuracy of the calculations. In the orbital descriptions, a value of \(2\pi\) is equivalent to 0, or stated another way, an orbit which begins at perigee angle equal to 0, is again
Original Code and Assertions

\[ FM = \frac{VALUE - TP}{PP} \]

\[ \text{ASSERT ( } FM \geq 0.0 \) \]

\[ \text{ASSERT ( } FM \leq \text{TWOPI} + \text{EPS} \) \]

Revised Code and Assertion

\[ FM = \frac{VALUE - TP}{PP} \]

\[ \text{ASSERT ( } \text{ABS(FM)} \leq \text{TWOPI} + \text{EPS} \) \]

Figure 6.7. First Incorrect Assertion

The problem becomes evident when checking the output from the test program. It is necessary to determine if the point described by the state vector is on the part of the orbit where the radius is increasing or the part where the radius is decreasing. (See Fig. 6.8.) This result is compared with the value of the MODE parameter when the VALUE parameter is interpreted as radius (MODE equal to 1 or 2) or altitude (MODE equal to 4 or 5).
The point can be located on the increasing ($M=1$) or decreasing ($M=2$) radius by comparing the time at apogee ($T_A$), the time at perigee ($T_P$), and the time given in the state vector ($T_S$). This is done in the code segment in Fig. 6.9. In order to make this calculation correctly, it is not only necessary to correct for the fact that $0$ equals $2\pi$ but also for the case in which the calculations give results very close to these values. These corrections are also shown in Fig. 6.10.

Another interesting result revealed by this assertion was that the calculation of the state vector time ($T_S$) was not corrected to be less than or equal to the period. This is a quirk of an algorithm in a support routine and was not revealed by the documentation. Again, the search routine identified input values which caused these assertions to be violated. It is difficult to see how test cases could have been constructed to illustrate these errors.
IF ( TP .EQ. 0.0 )
  IF ( TS .GE. TP .AND. TS .LE. TA)
    M = 1
  ELSE
    M = 2
  ENDIF
ELSE
  M = 2
ENDIF
ORIF ( TA .EQ. 0.0 )
  IF ( TS .GT. TA .AND. TS .LT. TP)
    M = 1
  ELSE
    M = 2
  ENDIF
ORIF ( TP .GT. TA )
  IF ( TS .GT. TA .AND. TS .LT. TP )
    M = 2
  ELSE
    M = 1
  ENDIF
ELSE:
  IF ( TS .GE. TP .AND. TS .LE. TA)
    M = 1
  ELSE
    M = 2
  ENDIF
ENDIF
ENDIF

Figure 6.9. Code to Locate Point on Radius
Corrections for time greater than period

\[ TP = \text{AMOD(ORBIT(7), PERIOD)} \]
\[ TA = \text{AMOD(TP+ORBIT(8)*PI, PERIOD)} \]
\[ TS = \text{AMOD (STATE(1), PERIOD)} \]

Corrections for time close to period

\[ \text{IF ( PERIOD - TP .LE. DELTAT )} \]
\[ TP = 0.0 \]
\[ \text{ENDIF} \]
\[ \text{IF ( PERIOD - TA .LE. DELTAT )} \]
\[ TA = 0.0 \]
\[ \text{ENDIF} \]
\[ \text{IF ( PERIOD - TS .LE. DELTAT )} \]
\[ TS = 0.0 \]
\[ \text{ENDIF} \]

Assertion to check MODE

\[ \text{IF ( TS .NE. TA) .AND. (TS .NE. TP) )} \]
\[ \text{ASSERT ( MODE .EQ. M )} \]
\[ \text{END IF} \]

Figure 6.10. Checking the Value of the MODE Parameter
The final assertion inconsistency also had to do with errors accumulated over computations and the fact that 0 is equal to 2π. This error arose in checking the angle from perigee, which is calculated when MODE equals 0. The angle from perigee is calculated from the input orbital elements and the radius. The radius can be calculated from the output state vector representation. This calculated value is then compared with the original value as input to ORBP in the VALUE parameter. The code to calculate the angle from perigee and the modified assertions are shown in Fig. 6.11. These assertions take into account that the calculated value may differ slightly from the original value and that 0 and 2π are equivalent. This inconsistency was discovered by the all-parameter search technique.

Code Segment

\[ Q = \frac{\text{ORBIT}(5)}{R - 1.0} / \text{ORBIT}(6) \]
\[ Q = \text{ACOS}(Q) \]

Corrections for angle near 2π

\[
\begin{align*}
\text{IF ( ABS(TWOPI-Q) .LE. DTHETA )} \\
& Q = \text{TWOPI} \\
\text{ENDIF} \\
\text{IF ( ABS(TWOPI-VALUE) .LE. DTHETA )} \\
& \text{VALUE} = \text{TWOPI} \\
\text{ENDIF}
\end{align*}
\]

Assertions Violated

\[
\begin{align*}
\text{ASSERT (AMOD(Q,TWOPI) .GE. AMOD(VALUE,TWOPI)-DTHETA)} \\
\text{ASSERT (AMOD(Q,TWOPI) .LE. AMOD(VALUE,TWOPI)+DTHETA)}
\end{align*}
\]

Figure 6.11. Checking the Angle from Perigee
DISCUSSION

The results from the experiment show that it is possible to detect errors automatically using assertions and search techniques. The major limitation of the technique as we see it is the difficulty in writing the assertions. The number of assertions which need to be written, the conditions they should describe and where they should be placed are all questions which are difficult to answer. In addition, the assertions are difficult to write and the task of writing them is not pleasant. On the other hand, the search testing technique aids in the refinement of the assertions.

Unfortunately, our results have also shown the limitations of assertions. There is sometimes no way to easily express exactly what is wanted by using the current semantics. In some cases, it seems that other techniques are more suited to detecting certain types of errors.

One may also argue with the technique of "error seeding," but we believe it to be a very effective way in which to control some of the problems in an experiment such as this. Using programs from actual development efforts containing unknown errors would introduce factors into the experiment which could not be controlled. Interpreting the results of such an experiment would therefore be more difficult.

Equating assertion violations with errors is also a point which may be argued. In this experiment, it was assumed that once an assertion violation was detected, the error would become self-evident. This is obviously not the case. This will be true only if assertions are placed in the correct spot and describe the nature of the error. Again, only further experimentation can determine how useful the technique is at locating errors.

The way in which the error function was constructed to allow the search routine to be used can also be questioned. Simply summing the
number of assertions to determine the value of the function is a crude technique. The search technique is thereby driven to select input values which maximize the number of assertions violated. We have found some evidence to indicate that errors are not randomly distributed; that they occur in groups. Therefore, searching for maximums of the error function should locate most of the errors in a program. However, this is still a crude method. We are investigating a method which takes the content of the assertions into account in generating new input values. This technique is taken from artificial intelligence research and will be the basis for further experiments.

In addition to the new experiments described above, we also believe that the techniques need to be applied to cases where more than one error occurs in the software, and to types of programs other than arithmetic computations (e.g. compilers). The efficiency of the technique relative to other types of testing should also be investigated.

We believe that the experiment successfully demonstrated the value of the search testing method. We were able to locate errors in a program automatically and relieved ourselves of the necessity of inventing test cases. In addition, the technique identified errors in our conception of the operation of the program as embedded in the assertions.

1Benson, op. cit.
Bibliography


S( & NEST

LOGICAL FUNCTION INPCHK ( MODE, VALUE, ORBIT, STATE )

CASE MODE = 1

CASE ( MODE = 0 ) $ VALUE IS ANGLE

CASE ( 1, 2 ) $ VALUE IS RADIUS

CASE ( 3 ) $ VALUE IS TIME

CASE ( 4, 5 ) $ VALUE IS ALTITUDE

CALL SMIT ( 10, ORBIT, ORB )

INPCHK = .TRUE.

CALL SMIT ( 10, ORBIT, ORB )

INPCHK = .FALSE.

CASE OF ( MODE )

CASE ( 0 ) $ VALUE IS ANGLE

CASE ( 1, 2 ) $ VALUE IS RADIUS

CASE ( 3 ) $ VALUE IS TIME

CASE ( 4, 5 ) $ VALUE IS ALTITUDE

A-2
SEQ NEST SOURCE

58 1 C *
60 1 C END CASE
61 1 C
62 1 C INVCRC ( ELLIPSE )
63 1 C BLOCK ( INPCPK = .FALSE. )
65 1 C END BLOCK
66 1 C BLOCK ( ELLIPSE )
67 1 C 
68 1 C 
69 1 C 
70 1 C 
71 1 C 
72 1 C 
73 1 C 
74 1 C 
75 1 C 
76 1 C 
77 1 C 
78 1 C 
79 1 C 
80 1 C 
81 1 C 
82 1 C END BLOCK
83 1 C BLOCK ( PRINT ELEMENTS )
84 1 C 
85 1 C 1000: FORMAT ( ŔSEMI-MAJOR AXIS = * G24.18 /
86 1 C * GCON = * G24.18 / * PERIOD / 2 PI = * G24.18 )
87 1 C END BLOCK
88 1 C RETURN
89 1 C

******************************************************************************

A-3
SIG NEST SOURCE

LOGICAL FUNCTION OUTCHK (MODE, VALUE, ORBIT, STATE)

1  C
2  CASON
3  CMODN OUTCHK
4  CCON B
5  C
6  C CHECKS FINAL CONDITIONS FOR SUBROUTINE SORBPI
7  C CONCN
8  C CONCN / CONCN /
9  1 PI, SRG, SLV, SMF, SKP, ABODY;
10  2 XCC, GCON, WBODY, ANGZRD, TWOP, HAPPI
11  CCMPK, CONCN
12  INTEGER NOCE $ INPUT NOCE FLAG
13  REAL VALUE $ INPUT VALUE PARAMETER
14  REAL ORBIT(10) $ INPUT ORBITAL ELEMENT VECTOR
15  REAL STATE(10) $ OUTPUT STATE VECTOR
16  C
17  TCONST C
18  ETA EPS / 1E-6 /
19  ETA DELTAT / 1E-2 / $ ABSOLUTE TIME TOLERANCE
20  ETA OTHERA / 1E-9 /
21  CCMPK, TCONST
22  C
23  C
24  C
25  C
26  C
27  C
28  C
29  C
30  C
31  C
32  C
33  C
34  C
35  C
36  C
37  C
38  C
39  C
40  C
41  C
42  C
43  C
44  C
45  C
46  C
47  C
48  C
49  C
50  C
51  C
52  C
53  C
54  C
55  C
56  C
57  C
58  1

A-4
```plaintext
59  ENDIF
60  IF ( PERIOD - TA .LE. DELTAT )
61    TA = 0.0
62  ENDIF
63  IF ( PERIOD - TS .LE. DELTAT )
64    TS = 0.0
65  ENDIF
66  IF ( TP .GE. 0.0 )
67    IF ( TS .GE. TP .AND. TS .LE. TA )
68      M = 1  $ T ON INCREASING RADIUS
69      ELSE
70      M = 2  $ T ON DECREASING RADIUS
71  ENDIF
72  CRIF ( TA .LE. 0.0 )
73    IF ( TS .GT. TA .AND. TS .LT. TP )
74      M = 2 $ T-ZERO ON INCREASING RADIUS
75      ELSE
76      M = 1 $ T-ZERO ON DECREASING RADIUS
77  ENDIF
78  CRIF ( TP .GT. TA )
79    IF ( TS .GT. TA .AND. TS .LT. TP )
80      M = 1 $ T ON INCREASING RADIUS
81      ELSE
82      M = 2 $ T ON DECREASING RADIUS
83  ENDIF
84  ELSE
85    IF ( TS .GE. TP .AND. TS .LE. TA )
86      M = 1 $ T-ZERO ON INCREASING RADIUS
87      ELSE
88      M = 2 $ T-ZERO ON DECREASING RADIUS
89  ENDIF
90  ENDIF
91  C
92  T = ORBIT(7) + ORBIT(3) + ORBIT(1) / R, ORBIT(9), ORBIT(6), ORBIT(8) )
93  C
94  T = AMODIT(PERIOD)
95  IF ( PERIOD - T .LE. DELTAT )
96    T = 0.0
97  ENDIF
98  IF ( TS .GE. T - DELTAT )
99  FAIL ( TIME )
100  ASSERT ( TS .LE. T + DELTAT )
101  FAIL ( TIME )
102  C
103  C
104  CASE OF ( MODE )
105  C
106  CASE ( 0 )
107  C
108  G = ORBIT(5) / R - 1.0 ) / ORBIT(6)
109  IF ( ABS(G) .GT. 1.0, AND. ABS(ABS(G) .LT. 1.0 + EPS )
110    G = SIGN(1.0, G)
111  C
112  G = ACOS ( G )
113  C
114  SEE RXIES PAGE 75
115  C
116  IF ( M .GE. 2 ) G = TWOPI - G
117  C
118  IF ( ABS(TWOPI - Q) .LE. DTHETA )
119    Q = TWOPI
120  C
121  ENDIF
122  C
123  IF ( ABS(TWOPI - VALUE) .LE. DTHETA )
```

A-5
**SLO NEST SOURCE**

**LOGICAL FUNCTION**

**OUTCHK** (MODE, VALUE, ORBIT, STATE)

```plaintext
118 2      . VALUE = TWOPI
119 1      ENDIF
120 1      ASSERT (MOD(G: TWOPI), GE, MOD(VALUE,TWOPI)+OIFHETA)
121 1      FAIL (PRINT Q VALUE)
122 1      ASSERT (MOD(TWOPI), LE, MOD(VALUE,TWOPI)+OIFHETA)
123 1      FAIL (PRINT Q VALUE)
124 1  C
125 2      CASE (1: 2)
126 1      IF ( (TS = NE) TA) AND (TS = NE. TP) )
127 2      ASSERT ( MODE =EQ, M )
128 2      FAIL (MODE ERROR)
129 1      END IF
130 1      ASSERT ( R =GE VALUE = EPS)
131 1      FAIL ( R VALUE )
132 1      ASSERT ( R =LE VALUE + EPS)
133 1      FAIL ( R VALUE )
134 1  C
135 2      CASE (3)
136 1  C
137 2      CASE (4: 5)
138 1      IF ( (TS =NE TA) AND (TS =NE. TP) )
139 2      assertion ( MODE =EQ, M=3 )
140 2      FAIL (MODE ERROR)
141 1      END IF
142 1      ASSERT ( R =RBDY =GE VALUE = EPS)
143 1      FAIL ( R RBDY VALUE )
144 1      ASSERT ( R =RBDY =LE VALUE + EPS)
145 1      FAIL ( R RBDY VALUE )
146 1  C
147 2      CASE ELSE
148 1      ASSERT ( ,FALSE, )
149  END CASE
150  C
151  C
152 1      BLOCK (SEMI-MAJOR AXIS)
153 1      WRITE(6:1000) R: = R: GCH: A = ORBIT(0)
154 1 1000 FORAT (0=24.16, 5X =24.16) 5X = =CON== 624.16 / 1. = =24.16, 5X =ORBIT(9) = = 624.16
155  END BLOCK
156  C
157  C
158 1      BLOCK (TIME)
159 1 1001 FORAT(+OTS== 624.16, 5X =24.16) 5X =PERIOD== 624.16)
160  END BLOCK
161  C
162 1      BLOCK (PRINT Q VALUE)
163 1 1003 FORAT(+GG== 624.16, 5X =VALUE== 624.16)
164  END BLOCK
165  C
166 1      BLOCK (MODE ERROR)
167 1      WRITE (6: 1005) R: = R: T: = T: STATE(9)
168 1 1005 FORAT(+PDATE== 624.18, 5X =T== 624.18, 5X =PERIOD== 624.18)
169 1 1005 FORAT(+PG=" = 624.18, 5X =VALUE== 624.18)
170  END BLOCK
171  C
172 1      BLOCK ( R VALUE)
173 1 1005 FORAT(+O=0, 624.18, 5X =VALUE== 624.18)
174  END BLOCK
175  C
176 1      BLOCK ( R RBDY VALUE)
177 1 1006 FORAT(+O=0, RBDY=, 624.18, 5X =VALUE== 624.18)
178 1 1006 FORAT(+O=0, RBDY=, 624.18, 5X =RBDY== 624.18)
179 1 1006 FORAT(+O=0, RBDY=, 624.18, 5X =RBDY== 624.18)
180  END BLOCK
181  RETURN
182  END
```

---

A-6
SLU nest source

1 C
2 C
3 C/ LIST=ALL
4 C SLU MGR TIME OREP (MODE, VALU, ORBELE, STATE)
5 CASCA
6 CMOCK CRBP
7 LECM A
8 C
9 C SOURCE DATE 69.1231 SET UP ACCEL COMPONENTS
10 C SOURCE DATE 69.0709 REMOVE CALL OF RITEF
11 C SOURCE DATE 65.0522 REVIS S TEST FOR APOGEE/PERIGEE
12 C SOURCE DATE 66.0611 CALL RITEF, NOT CRI; CHECK FOR ILLEGAL RAD
13 C SOURCE DATE 67.1121 CALL RITEF, NOT CRI; CHECK FOR ILLEGAL RAD
14 C SOURCE DATE 68.0920 USE ECC ANOMALY AS ITERATION VARIABLE
15 C SOURCE DATE 69.0601 USE RADIUS/AN GLE/TIME AS ITERATION VARIABLE
16 C
17 C
18 C
19 C
20 C
21 C
22 C
23 C
24 C
25 C
26 C
27 C
28 C
29 C
30 C
31 C
32 C
33 C
34 C
35 C
36 C
37 C
38 C
39 C
40 C
41 C
42 C
43 C
44 C
45 C
46 C
47 C
48 C
49 C
50 C
51 C
52 C
53 C
54 C
55 C
56 C

RETURN STATE VECTOR OF A POINT ON A KEPPLER ORBIT

MODE - SELECTS SPECIFICATION OF POINT IN ORBIT
0 - VALUE IS TRUE ANOMALY
1 - VALUE IS RADIUS (RADIUS INCREASING IN TIME)
2 - VALUE IS RADIUS (RADIUS DECREASING IN TIME)
3 - VALUE IS TIME AT WHICH POINT IS REACHED
4 - VALUE IS ALTITUDE (INCREASING IN TIME)
5 - VALUE IS ALTITUDE (DECREASING IN TIME)

VALUE - PARAMETER VALUE SPECIFYING POINT IN ORBIT

ORBELE - ORBITAL ELEMENT VECTOR

STATE - STATE VECTOR AT SPECIFIED POINT

WRITTEN 12/7/64

COMPKGCChCNh

CM'Et, CN OREELTIo).STATE(IO), AXAS(lO.3), SP(10 .OE(1I0

LOGICAL IF CH MK

$ VERIFY INICIAL CONDITIONS

LOGICAL QUICK

$ VERIFY FINAL CONDITIONS

REAL ORB(IO)

EQUIVALENCE (CRB(11), OE(11))

CATA EPS / 1E-6 /
CATA DELTAT / LE-2 /
$ ABSOLUTE TIME TOLERANCE
CATA SP / 1040.0 /
CATA AX ES / 3000.0 /
CATA EMISS / 1.0 /

RELEAlox(7) = ABS(ABS(X) - ABS(Y))/ANAXL(ABS(X), ABS(Y))

INITIAL ( IF (.IFCHMK MODE, VALU, ORBELE, STATE ))

POOC=MODE

VALU=VALU

IF(POOC+LT(3)) Go TO 2
SEQ NEST SOURCE

C 100
VAL = VAL - KODE
POCC = POCC - 3
2 CONTINUE

57 C 60
58 ASSERT ( POCC .GE. 0 )
59 ASSERT ( POCC .LE. 3 )
60 KODE = POCC + 1
61 CALL X(10,CRBEL(2),OE(2))
62 ASSERT ( OE(2) .GE. ORBEL(2) )
63 FAIL ( FIX 2 )
64 ASSERT ( OE(3) .EQ. ORBEL(3) )
65 FAIL ( FIX 3 )
66 ASSERT ( OE(4) .EQ. ORBEL(4) )
67 FAIL ( FIX 4 )
68 ASSERT ( OE(5) .EQ. ORBEL(5) )
69 FAIL ( FIX 5 )
70 ASSERT ( OE(6) .EQ. ORBEL(6) )
71 FAIL ( FIX 6 )
72 ASSERT ( OE(7) .EQ. ORBEL(7) )
73 FAIL ( FIX 7 )
74 ASSERT ( OE(8) .EQ. ORBEL(8) )
75 FAIL ( FIX 8 )
76 ASSERT ( OE(9) .EQ. ORBEL(9) )
77 FAIL ( FIX 9 )
78 ASSERT ( E .GT. 0.0 )
79 ASSERT ( E .LT. 1.0 )
80 IF ( E.GE.1.) GO TO 250
81 ASSERT ( KODE .GE. 1 )
82 FAIL ( FIX KODE )
83 ASSERT ( KODE .LE. 4 )
84 FAIL ( FIX KODE )
85 GO TO (10,20,20,30) + KODE
C 90 C VAL = VALUE
C 91
10 CONTINUE
92 C 93
93 ASSERT ( VALUE .GE. 0.0 )
94 ASSERT ( VALUE .LE. TWOP )
95 INVCHE ( ELLIPSE )
96 T = CRBTIM( POOC, VALUE, A, E, PP ) + TP
97 ASSERT ( T .GE. 0.0 )
98 E = VALUE
99 R = P/1.0 + EPS
100 ASSERT ( R .GE. A*(1.0 + E) - EPS )
101 ASSERT ( R .LE. A*(1.0 + E) + EPS )
102 GO TO 200
103 C 155 C VAL = VALUE
104 C 156
15 CONTINUE
156 INVCHE ( ELLIPSE )
157 ASSERT ( VALUE .GE. ( A + ( 1.0 - E ) )
158 ASSERT ( VALUE .LE. ( A + ( 1.0 + E ) )
159 IF ( VALUE .LT. ( A + ( 1.0 - E ) ) ) GOTO 24
160 IF ( VALUE .GT. ( A + ( 1.0 + E ) ) ) GOTO 26
161 T = ORBTIM( POOC, VALUE, A, E, PP ) + TP
162 ASSERT ( T .GE. 0.0 )
163 R = VALUE

A-8
NEW NEST SOURCE

417 C THIS CHANGE CORRECTS AN ERROR WHICH OCCURS WHEN THE
418 C RADIUS IS LARGE TO THE PERIGEE DISTANCE. IN THIS
419 C CASE, ERRORS ACCUMULATE DURING THE COMPUTATION CAUSE
420 C THE VALUE OF THE ARGUMENT OF THE ARC COSINE FUNCTION
421 C TO BE SLIGHTLY LARGER THAN ONE.
422 C
423 C IF (DUSYMPRIME.GT.1.0 .AND. ABSPRIME.LT.1.0 .EPS)
424 C ASSERT (SHPRIME .LT. 1.0)
425 C ASSERT (SHPRIME .LT. 1.0) 1
426 C ASSERT (SHPRIME .LT. 1.0)
427 C ASSERT (SHPRIME .LT. 1.0)
428 C ASSERT (SHPRIME .LT. 1.0)
429 C IF (ABS(PHI) .LT. 1.0 .EPS)
430 C GO TO 200
431 C)
432 C)
433 C)
434 C)
435 C)
436 C)
437 C)
438 C)
439 C)
440 C)
441 C)
442 C)
443 C)
444 C)
445 C)
446 C)
447 C)
448 C)
449 C)
450 C)
451 C)
452 C)
453 C)
454 C)
455 C)
456 C)
457 C)
458 C)
459 C)
460 C)
461 C)
462 C)
463 C)
464 C)
465 C)
466 C)
467 C)
468 C)
469 C)
470 C)
471 C)
472 C)
473 C)
474 C)
475 C)
476 C)
477 C)
478 C)
479 C)
480 C)
481 C)
482 C)
483 C)
484 C)
485 C)
486 C)
487 C)
488 C)
489 C)
490 C)
491 C)
492 C)
493 C)
494 C)
495 C)
496 C)
497 C)
498 C)
499 C)
500 C)
501 C)
502 C)
503 C)
504 C)
505 C)
506 C)
507 C)
508 C)
509 C)
510 C)
511 C)
512 C)
513 C)
514 C)
515 C)
516 C)
517 C)
518 C)
519 C)
520 C)
521 C)
522 C)
523 C)
524 C)
525 C)
526 C)
527 C)
528 C)
529 C)
530 C)
531 C)
532 C)
533 C)
534 C)
535 C)
536 C)
537 C)
538 C)
539 C)
540 C)
541 C)
542 C)
543 C)
544 C)
545 C)
546 C)
547 C)
548 C)
549 C)
550 C)
551 C)
552 C)
553 C)
554 C)
555 C)
556 C)
557 C)
558 C)
559 C)
560 C)
561 C)
562 C)
563 C)
564 C)
565 C)
566 C)
567 C)
568 C)
569 C)
570 C)
571 C)
572 C)
573 C)
574 C)
SEQ NEST SOURCE

175 ASSERT (ABS(E+EA*2.0-EA*1) =GT. 0.0)
176 FAIL (SMALL CIVISOR)
177 EA1=EA-EA*1+2/(EA+EA*2.0-EA*1)
178 ASSERT (MTRY =LT. MTRY)
179 GO TO 41
180 46 EA2=EA
181 48 EA=EA
182 ASSERT (MTRY =LT. MTRY)
183 GO TO 41
184 42 X2=1.0+COSE(A)
185 ASSERT (X2 =GE. 0.0)
186 ASSERT (X2 =LE. 2.0)
187 IF(X2 LE. 0) GO TO 48
188 ASSERT (X2 =LT. 0.0)
189 X1=SQRT((1.0-E1*(1.0-E)) *sin(EA))
191 C*2./ATAN2(X1.X2)
192 48 CONTINUE
193 C KERLESM EQUATION
194 ASSERT (FM =GE. EA = E+sin(EA) - EPS)
195 ASSERT (FM =LE. EA = E+sin(EA) + EPS)
196 R = P / (1.0 - E * COS(Q))
197 ASSERT (R =GE. A = (1.0 - E) - EPS)
198 ASSERT (R =LE. A = (1.0 + E) + EPS)
199 GO TO 200
200 C
201 C
202 C EUREKD ... AGN SET UP ANSWER AND RETURN
203 200 VG=SQRT(GCON)/P/R
204 ASSERT (VG =LE. SORH ( (1.0-E1*GCON)/(1.0-E1*A1) = EPS))
205 FAIL (PRINT VG GCON A E)
206 ASSERT (VG =LE. SORH ( (1.0-E1*GCON)/(1.0-E1*A1) = EPS))
207 FAIL (PRINT VG GCON A E)
208 V0=VX*SNR101=VG/R
210 C AT PERIGEE
211 C ASSERT (ABS(VR) =GE. ABS(E / (1.0 + E) * VG * sin(Q)) - EPS)
212 FAIL (PRINT VR E VR)
213 C AT APOGEE
214 C ASSERT (ABS(VR) =LE. ABS(E / (1.0-E) * VG * sin(Q)) = EPS)
215 FAIL (PRINT VR E VR)
216 GE(N)=GE(N) + C
217 ASSERT (GE(N) =GE(N) = PI)
218 FAIL (PRINT GE(N))
219 ASSERT (GE(N) =LE. TWOPI + TWOPI)
220 FAIL (PRINT GE(N))
221 C
222 C CALL EULANG(1, AXES, OT, 0)
223 C NDERV = 0
224 C NAIVRE (6 AXES)
225 C CALL TRANSFM(STATE =XES, SP =1.1)
226 CALL GRAVSTATE + STATE(81)
230 C FINAL (OUTCHKI MODE, VALU, ORBEL STATE)
232 C RETURN

A-10
**Block ( Ellipse )**

2431 C

* verifies that orbital vector vector is an elliptic orbit

2451 C

* assert ( ORB(5) .GT. -0.1 )

2471 C

* assert ( ORB(9) .LT. EPS )

2501 C

* assert ( ORI(1) .LT. EPS )

2511 C

* fail ( print perigee angle )

2531 C

* assert ( ORB(5) .GT. ORB(9) .LT. EPS )

2551 C

* assert ( ORB(9) .LT. EPS )

2571 C

* assert ( RELERR(ORB(9) .GT. EPS .AND. ORB(6) .LT. 1.0) )

2591 C

**End block**

**Block ( Axes )**

2601 C

* verifies direction cosine arrays

2631 C

* initial = NDERIV .GE. 0 .AND. NDERIV .LE. 2

2661 C

* N = 3 .AND. NDERIV = 4

2691 C

* if ( N = 2 .AND. NDERIV .GE. 0 )

2721 C

* do ( i = 1 . . N )

2751 C

* got = 0.0

2781 C

* cq ( k + 3 )

2841 C

* dot = got + axes(i,k) * axes(i,j)

2901 C

* end do

2931 C

* if ( i .EQ. j )

2971 C

* assert ( abs(dot) .LE. 1.0 )

3031 C

* else

3071 C

* assert ( abs(dot) .LT. EPS )

3151 C

* end if

3191 C

**End block**

**Block ( fix 2 )**

3231 C

* ge(2) = cabel(2)

3261 C

**End block**

**Block ( fix 3 )**

3301 C

* ge(3) = cabel(3)

3331 C

**End block**

**Block ( fix 4 )**

3461 C

**End block**
292 C
293 # CE(4) = CRBEL(4)
294 END BLOCK
295 C
296 # BLOCK (FIX 5)
297 C
298 # CE(5) = CRBEL(5)
299 END BLOCK
300 C
301 # BLOCK (FIX 6)
302 # CE(6) = CRBEL(6)
303 END BLOCK
304 C
305 # BLOCK (FIX 7)
306 # CE(7) = CRBEL(7)
307 END BLOCK
308 C
309 # BLOCK (FIX 8)
310 # CE(8) = CRBEL(8)
311 END BLOCK
312 C
313 # BLOCK (FIX 9)
314 # CE(9) = CRBEL(9)
315 END BLOCK
316 C
317 # BLOCK (PRINT PERIGEE ANGLE)
318 # WRITE (6, 1000) ORB(4)
319 C
320 # FORMAT (#, ARGUMENT OF THE PERIGEE = **) 624.18
321 END BLOCK
322 C
323 # BLOCK (PRINT SMALL CIVISOR)
324 # GOTO 42
325 END BLOCK
326 C
327 # BLOCK (PRINT VO GCON A E)
328 # WRITE (6, 1001) VO, GCON, A, E
329 C
330 # 1000: FORMAT (#, VQ=** G24.18, /, GCON=** 624.18, /, A=** 624.18, /, E=** 624.18)
331 END BLOCK
332 C
333 # BLOCK (PRINT GPRIME)
334 # WRITE (6, 1002) GPRIME
335 C
336 # 1002: FORMAT (#, GPRIME=** 624.18)
337 END BLOCK
338 C
339 # BLOCK (PRINT VR E VO)
340 # WRITE (6, 1003) VR, E, VO
341 C
342 # 1003: FORMAT (#, VR=** 624.18, 5X, E=** 624.18, 5X, VO=** 624.18, /, A=** 1
343 END BLOCK
344 C
A-12
SEQ NEST SOURCE

350 BLock ( PRINT 0(4 ) )
351 1  WRITE (6,1004) COE(4), G
352 1 1004: FORMAT(1OE(4))= G24.18, 5X *G=G24.18)
353 END BLOCK
354 BLock ( TIME MAX )
355 1  WRITE ( 6, 1005 ) VALUE, PP*TP*OPI
356 1 1005: FORMAT ( * VALUE= * G24.18, 5X, *10PP*TP*OPI= G24.18 )
357 END BLOCK
358 BLock ( F P MAX )
359 1  WRITE ( 6, 1006 ) FP, TP*OPI
360 1 1006: FORMAT ( * FP = *, G24.18, 5X, * TP*OPI= G24.18 )
361 END BLOCK
362 BLock ( FM )
363 1  WRITE (6,1007) FM, VALUE, TP, PP
365 END BLOCK
366 C
367 RETURN
368 END

*******************************************************************************
APPENDIX B

Chronological List of Papers Submitted
The following collection of papers and reports was supported by AFOSR contract number F49620-79-C-0115.


B-2
APPENDIX C

Personnel Associated with the Project
The following persons participated in the research and experiments:

1. Dorothy M. Andrews, MSEE, University of California, Santa Barbara.
2. Jeoffrey P. Benson, PhD., University of California, Santa Barbara.
3. Nancy B. Brooks, MS, University of Illinois.
4. Reginald N. Meeson, MSEE, University of California, Santa Barbara.
5. Dennis W. Cooper, MSEE, Stanford University.