CORE
Metadata, citation and similar papers at core.ac.uk
Provided by NASA Technical Reports Server

28p.

Semi-Annual Progress Report

Grant No. NAG-1-511-3
September 1, 1984 - December 31, 1986

A SECOND GENERATION EXPERIMENT IN FAULT-TOLERANT SOFTWARE

IN-31447

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Attention: Dr. Dave E. Eckhardt
ISD M/S 130

Submitted by:

J. C. Knight
Associate Professor

Report No. UVA/528235/CS87/103
August 1986

(NASA-CR-179848) A SECOND GENERATION EXPERIMENT IN FAULT-TOLERANT SOFTWARE Semiannual Progress Report, 1 Sep. 1984 - 31 Dec. 1986 (Virginia Univ.) 28 p CSCL 09B

N87-12242

Unclas
G3/61 44678

DEPARTMENT OF COMPUTER SCIENCE

Semi-Annual Progress Report

Grant No. NAG-1-511-3
September 1, 1984 - December 31, 1986

A SECOND GENERATION EXPERIMENT IN FAULT-TOLERANT SOFTWARE

Submitted to:

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Attention: Dr. Dave E. Eckhardt
ISD M/S 130

Submitted by:

J. C. Knight
Associate Professor

Department of Computer Science
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
UNIVERSITY OF VIRGINIA
CHARLOTTESVILLE, VIRGINIA

Report No. UVA/528235/CS87/103 Copy No. _____

August 1986

TABLE OF CONTENTS

		Page
I.	INTRODUCTION	1
II.	MULTI-VERSION SOFTWARE TEST ENVIRONMENT STRUCTURE	3
III.	MULTI-VERSION SOFTWARE TEST ENVIRONMENT FORMATS	12
IV.	ERROR DETECTION BY SELF TEST EXPERIMENT	23
	REFERENCES	25

PRECEDING PAGE BLANK NOT FILMED

SECTION I

INTRODUCTION

The purpose of the work performed under this grant is to begin to obtain information about the efficacy of fault-tolerant software by conducting two large-scale controlled experiments. In the first, an empirical study of multi-version software is being conducted. This experiment will be referred to as the "MVS" experiment in this report. The second experiment is an empirical evaluation of self testing as a method of error detection and will be referred to as the "STED" experiment.

The MVS experiment is being conducted jointly by NASA, four universities, and Charles River Analytics, Inc. The participating universities are North Carolina State University, the University of California at Los Angles, the University of Illinois at Urbana-Champaign and the University of Virginia. During the current grant reporting period, the work at the University of Virginia in the MVS experiment has centered around the preparation of an environment for testing the subject programs. Other elements of the experiment are being carried out at the other sites.

The purpose of the MVS experiment is to obtain empirical measurements of the performance of multi-version systems. Twenty versions of a program have been prepared at four different sites (the universities) under reasonably realistic development conditions from the same specifications. The experimenters are now preparing to evaluate these programs in various ways, in particular by extensive dynamic testing.

The STED experiment is being conducted jointly by the University of Virginia and the University of California, Irvine. During the current grant reporting period, the work at the University of Virginia in the STED experiment has involved design of the experiment and preparation of the subject test programs.

The purpose of the STED experiment is to obtain empirical measurements of the performance of assertions in error detection. Eight versions of a program have been modified to include assertions at two different universities under controlled conditions. The experimenters are now preparing to evaluate these programs by comparing their error-detection performance in comparison with voting in 2-version systems.

In this report, we describe the overall structure of the testing environment for the MVS experiment and its status in section II. In section III, we describe a preliminary version of the control system that has been implemented for the MVS experiment to allow the experimenter to have control over the details of the testing. We describe our work to date in the STED experiment in section IV.

SECTION II

MULTI-VERSION SOFTWARE TEST ENVIRONMENT STRUCTURE

The basic layout of the test environment was decided at joint meeting of the research members held in Boston in April, 1986. This basic layout has been extended in various ways as a result of numerous discussion and changes in requirements. A fundamental goal of the environment is to be as independent of the machine used for the testing as possible. Thus, although built and distributed as a UNIX based system, the environment should run on other machines with little change.

The philosophy of the test system is to allow the experimenter to specify the initial conditions and sensor failure requirements for a single simulated flight and then to generate a series of acceleration values that are supplied to the programs along with the initial and failure conditions. This is intended to simulate a single flight of an aircraft.

The system allows the experimenter to specify that several (perhaps many) flights are required each with a different (but perhaps similar or related) set of initial conditions. For example, some parameter might have to be varied systematically over some range. In this case, the system will create a sequence of initial conditions in which the required parameter is varied but the same set of acceleration and Euler angles is used for each simulated flight in the set. The systematic variation and the reexecution of the programs on the set of accelerations is handled by the execution environment.

The environment consists of a set of programs that are organized into five tiers or levels. The general form is shown in figure 1. The interfaces between the levels are precisely defined. Each consists of character files thereby permitting the greatest degree of machine independence. The details of the interfaces are referred to here as formats. For example, format 0 describes the interface between tier-0 and tier-1, and consists of a single explicitly named file. Other formats use more than one file, including in most case the standard input and standard output files. Figures 2, 3, 4 and 5 show the input and output details for each tier individually. In these figures, a dashed line represents either standard in or standard out, and a solid line connected to a named ellipse indicates an explicit disk file. The exact content of the formats is described in the section III.

Tier-0

The purpose of the single program in tier-0 is to interact with the experimenter to determine the parameters of the tests that have to be run. This program produces a file of data for control of subsequent programs after gathering the details of the required tests from the experimenter. The program makes no decisions and generates no data itself (except defaults) so the output datafile contains everything that the experimenter supplied. For simple tests, most parameters can take the default values allowing the definition of the tests to be created with very little input from the experimenter.

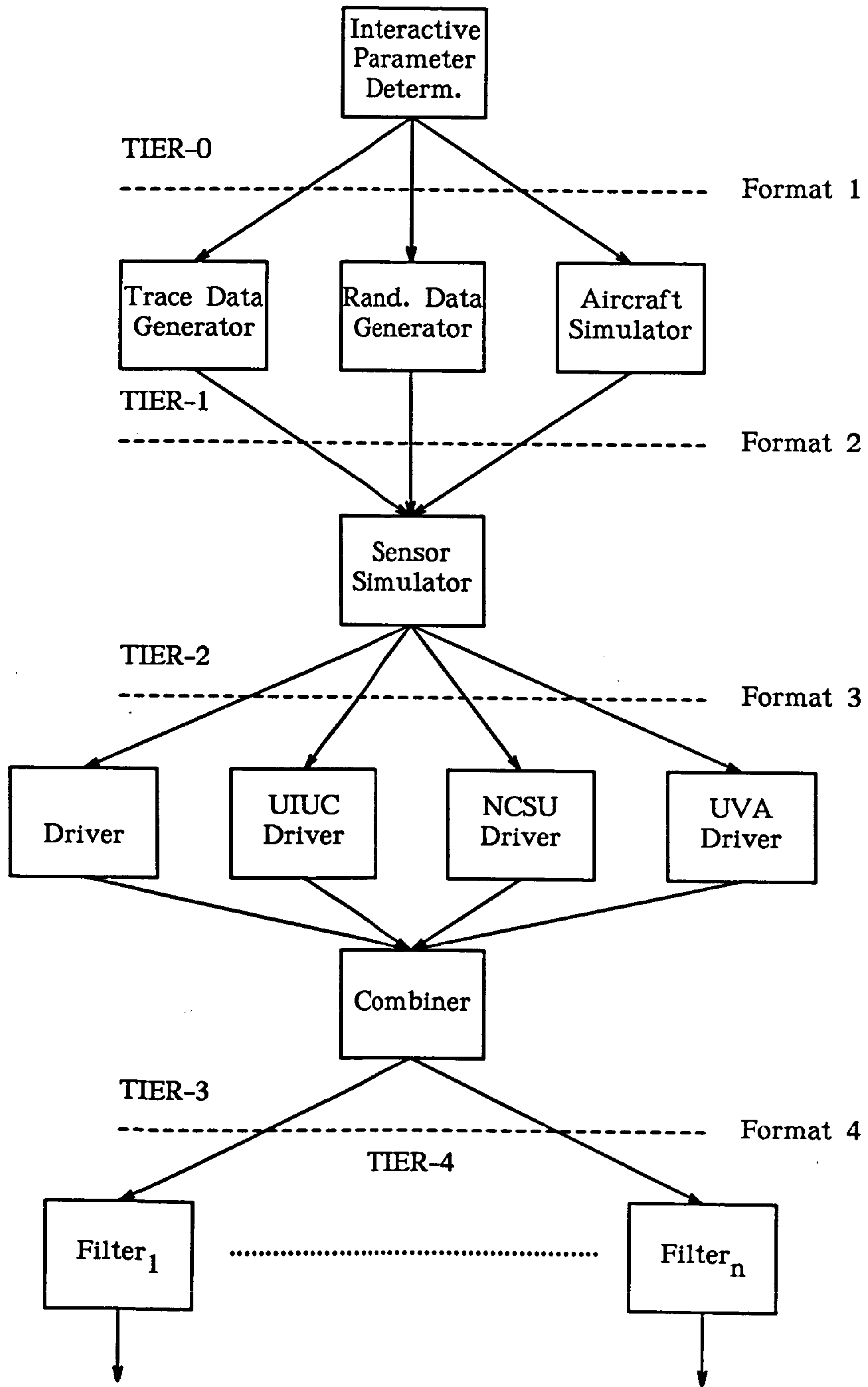

Fig. 1 - Overall Environment Layout

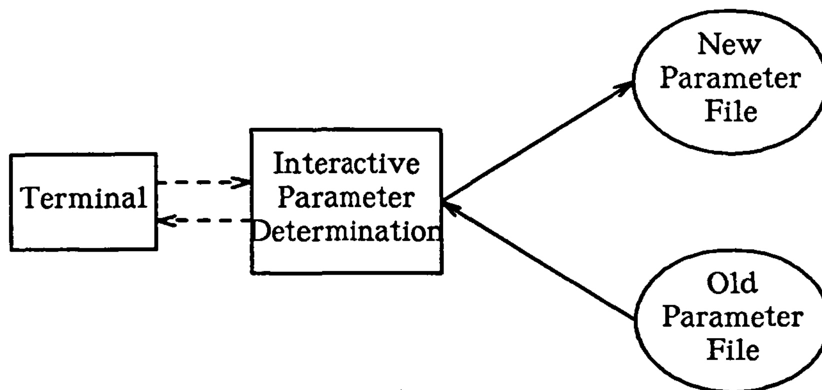

Fig. 2 - Input And Output For Tier-0

Optionally, an existing parameter file can be read by the program to provide a set of initial conditions for the interaction with the experimenter. Thus if two sets of tests are to be run with minimal change between runs, the data file from one can be read in to set values initially for the experimenter during the interaction. A second data file that differs little from the first can the be generated merely by indicating the changes.

As will be seen from the discussion in section III, the interaction carried out by the tier-0 program could result in the specification of a large number of tests. The number is computed and supplied to the experimenter for confirmation.

The tier-0 program is written in Pascal. Although it is interactive, the program operates in a very simple menu style to ensure independence of terminal characteristics,

Tier-1

The programs in tier-1 obtain the specifications for the initial conditions from the file that the tier-0 program creates. They then generate the series of accelerations and angles that is required for the specified test flight(s).

There are three programs in tier-1. Each operates with the same input and output interfaces (formats 0 and 1), and as far as the rest of the environment is concerned, they are equivalent. The first generates a series of accelerations from a trace file. It merely reads accelerations obtained from measurement on the B737 aircraft and converts them to the format required

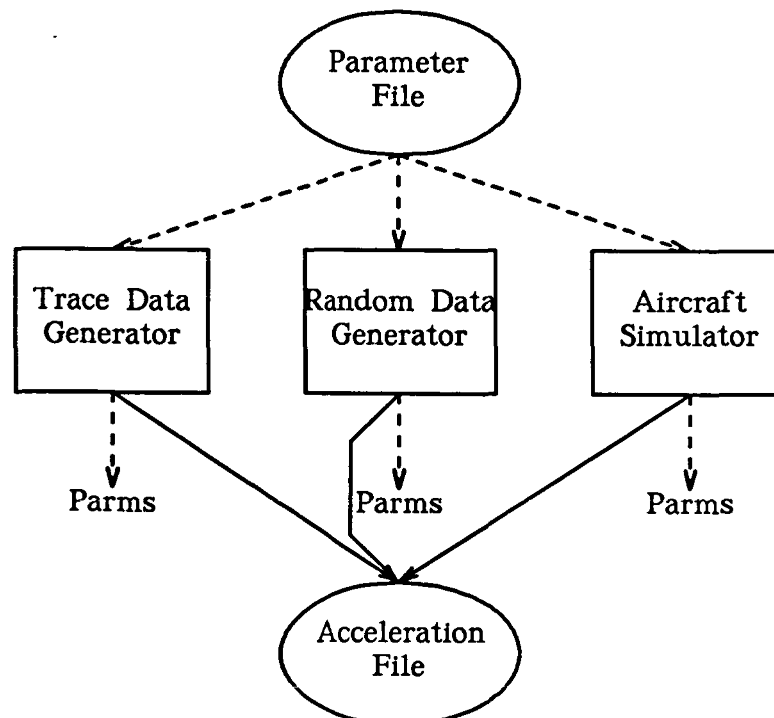

Fig. 3 - Input And Output For Tier-1

by the following tier. This program is written in Pascal.

The second program generates accelerations and angles randomly. Although these values are unrealistic, they are adequate for testing. This program is written in Pascal.

The third program is an aircraft simulator that generates realistic values for the required data. This program is being prepared by Charles River Analytics. It is written in FORTRAN.

Tier-2

There is a single program in tier-2, the sensor simulator. This program is written in FORTRAN and the original version was supplied by Charles

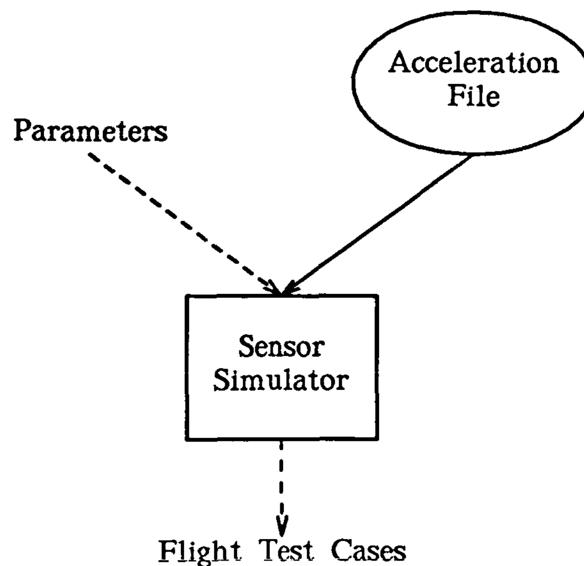

Fig. 4 - Input And Output For Tier-2

River Analytics. This program has been modified by the University of Virginia to include the necessary loops for driving the following tiers where parameters are being varied in a series of simulated flights. The program takes an acceleration value and other parameters supplied from the data file generated by the tier-0 program and generates the corresponding sensor values.

Tier-3

Tier-3 contains the actual versions under test, several driver programs, and a utility program. It was deemed inadvisable to attempt to run all 20 versions together with a single driver. Merely compiling a major program of that size would take considerable resources. We have found that several

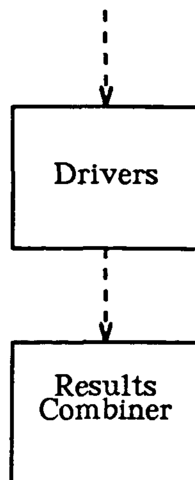

Fig. 5 - Input And Output For Tier-3

Pascal compilers available to us were unable to compile such a program.

To avoid these problems, tier-3 contains four drivers, one for each university. They read the same inputs but create their own output files and so can be run in parallel. A utility program (the combiner) is then used to combine the output files so that they appear to have come from a driver that executed all twenty programs together. The combiner merely reorganizes the output files of the four drivers. It makes no content changes to the data.

The tier-3 drivers keep the initial conditions for a particular flight as global data while executing the required versions. This data includes the calibration data. Thus although each program thinks that it will operate on a single flight acceleration value, the calibration and other parametric information is identical for each acceleration for a flight and so the effect is to have the program do calibration followed by a series of acceleration values.

Tier-4

Tier-4 consists of an arbitrary number of "filter" programs that read the output of tier-3 and do useful things with the output. As new functions are required, new filters will be added. Present filters include programs to allow formatted printing of the raw test results in various forms. Filters are being developed to allow the raw data produced by the tests to be stored in as compact a form as possible on tape. The purpose of these filters is to allow tests to be run and their entire output to be saved for

later analysis. This will permit tests to be performed without all possible analyses being performed at the same time.

SECTION III

MULTI-VERSION SOFTWARE TEST ENVIRONMENT FORMATS

In this section the detailed contents of the interface formats are discussed.

Previously, the set-up for testing the RSDIMU versions allowed only minimal control over the generation of the input variables. Consequently, it was not possible to study the effects of gradually changing the values of such variables without repeatedly recompiling the test programs, which would be, of course, senseless. What follows describes the means used to give the experimenter greater control over generation of input values to the versions in the present environment. With such control the effects of each RSDIMU input variable can be studied individually or in combination with other selected input variables in whatever ways might be deemed desirable during the course of the testing.

Each set of input variables, as generated by this control information, is interpreted as a set of *initial conditions*. Given this set of initial conditions, a series of testcases is generated, each with a different acceleration value and set of vehicle frame Euler angles. The number of acceleration values used is given by a control parameter. Within the series of testcases, sensor failure also is simulated as specified by the control information. By keeping the same set of initial conditions for a sequence of acceleration values, the calibration data is kept the same, and the resulting effect is that of performing one calibration of the sensors and then saving that information to

be used while performing a series of in flight sensor readings and sets of calculations. In this manner, the capability is achieved for what is hoped to be a reasonable simulation of "flight", with successive sensor readings taken over a period of time.

Sensors are failed on each specified testcase according to their control values (see below). The test drivers in tier-3 have been modified so that for each two consecutive acceleration value and angle sets with the same set of initial conditions, the values for linfailin input to each version are the values for linfailout computed by that version on the previous acceleration and angle set. In this way the various responses of the versions to sensor failure can be studied over a sequence of acceleration values.

The variables that can be controlled are as follows:

linstd : noise standard deviation for accelerometers

linfailin : accelerometer failure initial conditions

rawlin : raw sensor data for acceleration computation

dmode : display mode

temp : current temperature on each face

scale0,

scale1,

scale2 : linear accelerometer slope coefficients

misalign : accelerometer misalignment angles

nsigt : noise tolerance

phii,

thetai,

psii : Euler angles for rotation from the vehicle frame to the instrument frame

Rawlin cannot be controlled directly, as it must be generated by the sensor simulator based on the acceleration, Euler angle, and misalignment angle values. However, whether its value for a given sensor should reflect failure during calibration or failure during flight can be controlled directly, and so can the value for noise which is used in generating the rawlin value for a sensor which is to be found noisy by the RSDIMU versions. Offraw, the calibration data for the eight accelerometers, can also be indirectly controlled in similar ways, but that control is being left for a later modification of the test control.

For all of the variables above, except for linfailin, misalign, scale0, scale1, scale2, and the sensor failure control information, there are defined three control modes:

0 : to indicate that a value should be randomly-generated within a specified range,

1 : to indicate that the variable should be varied over a specified range while all other variables except the acceleration values are held constant,

2 : to indicate that the variable should be set to a certain specified constant.

For each of these variables the control information is contained on one line of standard input, with the formats as follows:

for mode 0: 0 min max

for mode 1: 1 lowerbound upperbound step

for mode 2: 2 constant

"Min" and "max" specify the range within which the value is to be randomly-generated. "Lowerbound" and "upperbound" specify the range over which the variable is to be varied for mode 1 and "step" specifies the increments by which it is to be varied. "Constant" is the specified value to which the variable is to be set when mode 2 is used. Min, max, lowerbound, upperbound, step, and constant will each be assumed to be of the same type as the RSDIMU input variable which they are being used to control.

For linfailin and for the control information regarding which sensors will fail during calibration (equivalent to the output variable "linnoise") the format is slightly different:

for mode 0: 0 lowerbound upperbound

for mode 1: 1 number

for mode 2: 2 bool1 bool2 bool3 bool4 bool5 bool6 bool7 bool8

Here the modes are defined as follows:

0 : specifies that a randomly-generated number of the eight values be set to true. This number will be between "lowerbound" and "upperbound"

inclusive. Which of the sensor values are set will be randomly determined.

1 : specifies that "number" of the eight values be set to true. Which of the sensor values are set will be randomly determined.

2 : specifies that the eight values be set to the respective constant values, "bool1" through "bool8"

"Number" is assumed to be an integer and "bool1" through "bool8" will be assumed to be either 0's or 1's, with "1" representing true and "0" representing false. "Lowerbound" and "upperbound" will be assumed to be integers between 0 and 8 inclusive.

For misalign, scale0, scale1, and scale2 the format is as follows:

min max

In this case, since there is only one mode, mode numbers are unnecessary. That mode specifies that each of the 24 misalign values (or each of the 8 scaleX values) be randomly-generated between the values "min" and "max", which are assumed to be real numbers and which may be equal.

In addition to the ability to control the values of RSDIMU input variables, it is desirable to have the ability to control which sensors fail during "flight" and during which iteration of sensor reading during the "flight" each sensor fails. To this end the following format is used:

int1 int2 int3 int4 int5 int6 int7 int8

"Int1" through "int8" are integers which represent the sensor reading iteration during which sensors 1 through 8 respectively will fail. A value of 0 for "intX" indicates that sensor X will not fail during this test of the RSDIMU procedures. The non-zero values for "int1" through "int8" must be distinct from one another, as it is assumed in the RSDIMU specifications that at most one sensor will fail on a given sensor reading. The indicated sequence of sensor failures will be simulated once for each set of initial conditions (i.e. for each "flight"). Sensor failure simulation will be accomplished by modifying the generated value for rawlin in such a way that it will appear too noisy to be functional. The modifying value used will be the value for "noise" generated by its control information. Sensors are made to fail not only on the desired sensor reading, but also on all successive readings within a given "flight", so that if a particular RSDIMU version fails to mark that sensor as having failed on that iteration, it may still do so on a subsequent iteration. The value for "intX" should not exceed the value for the number of acceleration values per "flight".

The known acceleration values and the values for phiv, thetav, and psiv are no longer obtained from standard input. Instead, the tier-1 programs write them to a temporary file named by the control parameter "AccelerationFileName" so that they can be used repeatedly (i.e. for each set of "initial conditions").

These control formats have been implemented in such a way that, if two or more variables are being varied over ranges, testcases are generated for all combinations of all the values over which each is being varied, and at the same time conform to any control specifications for other variables.

FILE FORMAT 0 (format for input to tier-1 programs):

```
CONTROL INFO  * | Number of Acceleration Values for each flight
      BLOCK     | Seed1, Seed2 for random numbers, tiers 1 and 2
                | Version Selection Vector (1 element per version, 1 selects
                |         corresponding version for execution, 0 bypasses)
                | AccelerationFileName
                | control info for linstd
                | control info for linfailin
                | control info for number of sensors to fail in calibration
                | control info for sensor failure during flight
                | control info for noise
                | control info for dmode
                | control info for temp[1]
                | control info for temp[2]
                | control info for temp[3]
                | control info for temp[4]
                | control info for scale0
                | control info for scale1
                | control info for scale2
                | control info for misalign
                | control info for nsigt
                | control info for phii
                | control info for thetai
                | control info for psii
```

FILE FORMAT 1 (format for input to tier-2 programs):

CONTROL INFO BLOCK (see • above)

+ a file containing Number_of_Acceleration_Values lines of

VehicleAccel[x] VehicleAccel[y] VehicleAccel[z] phiv thetav psiv

FILE FORMAT 2 (format for input to tier-3 programs):

```
CONTROL INFO BLOCK   (see • above)
FLIGHT BLOCK  (repeated once for each flight)
        | obase
        | offraw[1,1]  ...  offraw[8,1]
        |    .                 .
        |    .                 .
        |    .                 .
        | offraw[1,50] ...  offraw[8,50]
        | linstd
        | linfailin[1] ... linfailin[8] {encoded as integers 0..1}
        | dmode
        | temp[1] ... temp[4]
        | scale0[1] ... scale0[8]
        | scale1[1] ... scale1[8]
        | scale2[1] ... scale2[8]
        | misalign[1,1] ... misalign[1,6]
        | misalign[2,1] ... misalign[2,6]
        | misalign[3,1] ... misalign[3,6]
        | misalign[4,1] ... misalign[4,6]
        | nsigt
        | phii
        | thetai
        | psii
        | ACCELERATION BLOCK (repeated once per accel value)
        |        | rawlin[1] ... rawlin[8]
        |        | normface[1] ... normface[4]
        |        | phiv  thetav  psiv
        |        | KnownBestest.acceleration[x] ... KnownBestest.acceleration[z]
```

FILE FORMAT 3 (format for input to tier-4 programs):

```
CONTROL INFO BLOCK  (see • above)
The Following Repeated For Each Flight:
    | KnownBestest.accel[x] ... KnownBestest.accel[z]
    | The Following Repeated for Each Version Selected For Execution:
    |      | linoffset[1] ... linoffset[8]
    |      | linnoise[1] ... linnoise[8] {encoded booleans}
    |      | linfailout[1] ... linfailout[8] {encoded booleans}
    |      | linout[1] ... linout[8]
    |      | dismode
    |      | disupper[1] ... disupper[3]
    |      | dislower[1] ... dislower[3]
    |      | bestest.status bestest.acceleration[1] ... bestest.acceleration[3]
    |      | chanest[1].status chanest[1].accel[1]...chanest[1].accel[3]
    |      | chanest[2].status chanest[2].accel[1]...chanest[2].accel[3]
    |      | chanest[3].status chanest[3].accel[1]...chanest[3].accel[3]
    |      | chanest[4].status chanest[4].accel[1]...chanest[4].accel[3]
    |      | chanface[1] ... chanface[4]
    |      | systatus {encoded boolean}
```

SECTION IV

ERROR DETECTION BY SELF TEST EXPERIMENT

In the second experiment, the empirical evaluation of self testing for error detection, we are attempting to determine how well programmers can prepare assertions for the detection of execution-time errors. This study is empirical.

From the set of twenty-seven programs written for the Knight and Leveson experiment [1], eight were chosen for modification. Each of these eight was supplied to three programmers who worked separately to add assertions to the programs. The effort expended by each programmer was one week. The experiment protocol was:

(1) The program specification was supplied to the programmers and they were given a presentation describing the goals of the experiment, the protocol, and the schedule. Each programmer was also supplied with a copy of the chapter on error detection from the text by Anderson and Lee [2].

(2) The programmers were required to study the specification and the text on error detection, and then to attempt to develop assertions based purely on knowledge of the specifications.

(3) When the specification-based assertions were complete, the programmers were supplied with the source text of the program they were to modify.

The programs were then modified to include assertions.

(4) After the assertions had been added, the programmers were supplied with fifteen test cases that executed correctly prior to the addition of the modifications. These test cases should have executed correctly after the addition of the assertions. The programmers were requested to test the modifications and assertions in any way they chose in addition to the fifteen test cases.

(5) Finally, the modified programs were subjected to the same set of acceptance tests that had been used in the original experiment [1].

The programmers were asked to keep detailed logs of their effort during the time they were working on the project, and each was required to complete a background technical and educational questionnaire.

At this time, all three copies of each of the eight programs has been prepared and accepted. The modified programs are being tested using the same test driver and test cases that were used in the original experiment.

REFERENCES

(1) Knight, J.C., N.G. Leveson, and L.D. St.Jean, "A Large-Scale Experiment In N-Version Programming", Digest of Papers FTCS-15: *Fifteenth Annual International Conference on Fault Tolerant Computing*, Ann Arbor, Michigan, June, 1985, pp. 135-139.

(2) Anderson T., and P.A. Lee, "Fault Tolerance: Principles and Practice", Prentice Hall International, 1981.