NASA Technical Memorandum 101668

SOFTWARE VERIFICATION PLAN FOR GCS

LESLIE A. DENT
ANITA M. SHAGNEA
KELLY J. HAYHURST

JANUARY 1990

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

(NASA-TM-101668) SOFTWARE VERIFICATION PLAN FOR GCS (NASA) 112 p CSCL 09B

N90-18057

Unclas
G3/38 0261671

Preface

The Software Verification Plan for GCS is document # 11 in a series of fifteen documents which fulfill the Radio Technical Commission for Aeronautics RTCA/DO-178A guidelines, "Software Considerations in Airborne Systems and Equipment Certification [1]." The documents are numbered as specified in the DO-178A guidelines. The documents in the series are used to demonstrate compliance with the DO-178A guidelines by describing the application of the procedures and techniques used during the development of flight software. These documents were prepared under contract with NASA-Langley Research Center as a part of their long term research program addressing the fundamentals of the software failure process.

This project consists of two complementary goals: first, to develop software for use by the Research Triangle Institute (RTI) in the software error studies research program sponsored by NASA-Langley Research Center [2]; second, to use and assess the RTCA/DO-178A guidelines for the Federal Aviation Administration (FAA). The two goals are complementary in that the use of the structured DO-178A guidelines in the development of the software will ensure that the test specimens of software have been developed according to the industry standards for flight critical software. The error studies research analyses will then be conducted using high quality software specimens.

The implementations will be subjected to two different software testing environments: verification of each implementation according to the RTCA/DO-178A guidelines and replicated random testing in a configuration which runs more than one test specimen at a time. The term *implementations* refers to bodies of code written by different programmers, while a *version* is a piece of code at a particular state (i.e., version 2.0 is the result of code review). This research effort involves the gathering of product and process data from every phase of software development for later analysis. More information on the goals of the Guidance and Control Software (GCS) project are available in the *GCS Plan for Software Aspects of Certification.*

The series consists of the following documents:

- *GCS Configuration Index* Document no. 1
- *GCS Development Specification* Document no. 2
- *GCS Design Descriptions* One for each software implementation. Document no. 3
- *GCS Programmer's Manual* Document no. 4, includes Software Design Standards, document no. 12.
- *GCS Configuration Management Plan* Document no. 5A
- *Software Quality Assurance Plan for GCS* Document no. 5B
- *GCS Source Listing* One for each software implementation. Document no. 6
- *GCS Source Code* One for each software implementation. Document no. 7
- *GCS Executable Object Code* One for each software implementation. Not available on hardcopy. Document no. 8
- *GCS Support/Development System Configuration Description* Document no. 9
- *GCS Accomplishment Summary* Document no. 10
- *Software Verification Plan for GCS* Document no. 11
- *GCS Development Specification Review Description* Document no. 11A
- *GCS Simulator (GCS_SIM) System Description* Document no. 13
- *GCS Simulator (GCS_SIM) Certification Plan* Document no. 13A
- *GCS Plan for Software Aspects of Certification* Document no. 14

Contents

List of Figures

List of Tables

1 Introduction

According to the Radio Technical Commission for Aeronautics RTCA/DO-178A guidelines, "Software Considerations in Airborne Systems and Equipment Certification"[1], the software verification plan is written to provide instruction to the testers and to explain the verification which will be performed at each stage of the development cycle. The plan should include descriptions of the testing methods to be used and their purposes, discussion of tools used, descriptions of the actual tests, and testing results. Since this release of the plan precedes the actual testing of the software, no test cases or results are presented; they will be added in a later release. This plan is organized by the development phases since each phase has an associated verification activity. The details of the strategy and *general* procedures for each phase of the verification are described. See Figure 1 for an overview of the development phases and reviews. The schedule for verification activities can be found in the *GCS Plan for Software Aspects of Certification*.

A brief description of the software is provided to facilitate understanding of the various phases of the verification process. Following the software description, an overview of the general verification procedures and a description of the tools used during the verification process are given. Next, a detailed description of each phase of verification is presented. The phases which are addressed here are Design, Code, Module Testing, Sub-Frame Testing, Frame Testing, and System Testing. Module Testing and Sub-Frame testing are equivalent to "Module Testing" in the DO-178A guidelines. Frame Testing is equivalent to "Module Integration Testing" in the DO-178A guidelines. System Testing is equivalent to "System Validation Testing" in the DO-178A guidelines. The DO-178A guidelines "HW/SW Integration Testing" are not necessary since no specific target hardware is presently associated with GCS. Since the *GCS Development Specification* existed prior to the incorporation of the DO-178A guidelines into the experiment, the verification plan does not address verifying either the software or system requirements. The *GCS Development Specification Review Description* addresses the verification of the software and system requirements. As with other DO-178A documents, an attempt has been made to reduce redundancy by referring to other documents when appropriate.

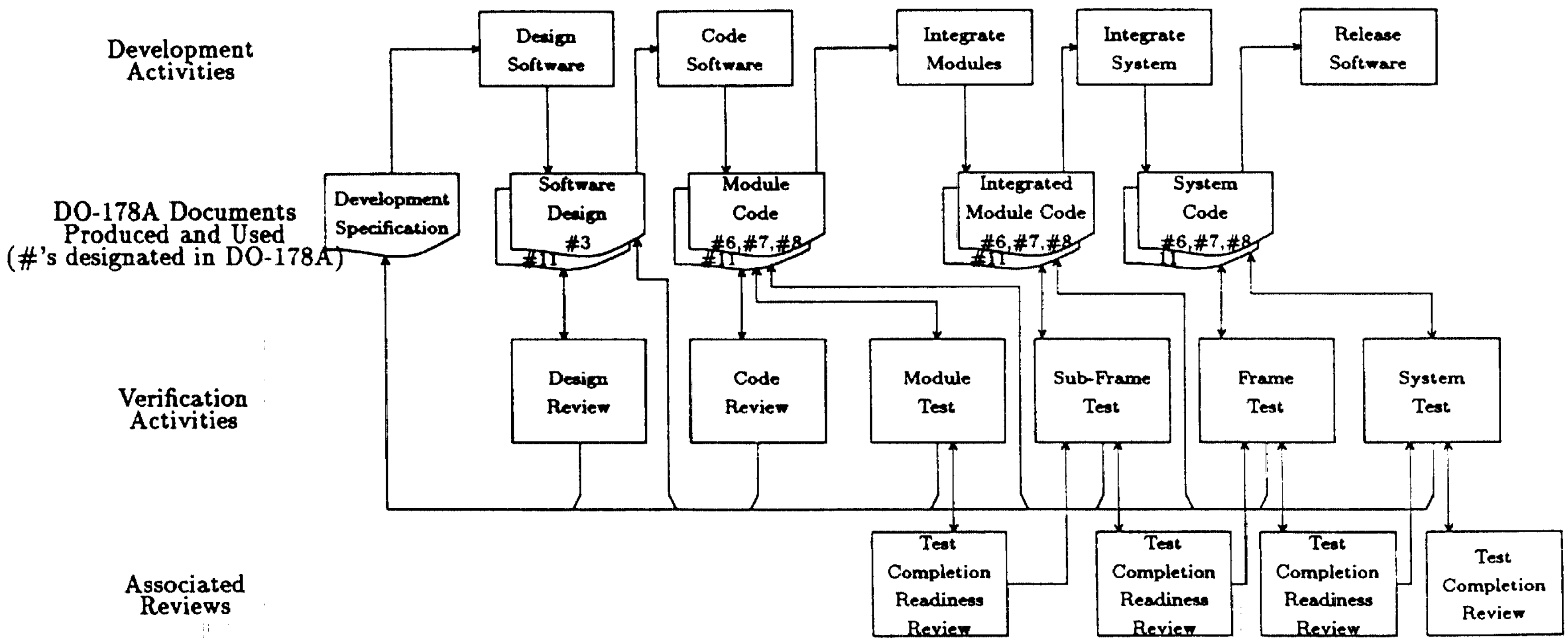

Figure 1: Overview of Development Phases and Reviews

2 Software Description

The Guidance and Control Software(GCS) implementations are being developed according to the *Guidance and Control Software Development Specification*. This software is to provide guidance and engine control for the terminal descent phase of a planetary lander onto a surface and to transmit sensory information about the vehicle and its descent, to a recording device. The vehicle should descend along a predetermined velocity altitude contour that was chosen to conserve fuel and to effect a safe[1] attitude and velocity upon landing.

The GCS Development Specification calls for the software to be divided into three separate processing parts called sub-frames. These sub-frames perform the following functions: Sensor Processing, Guidance Processing, and Engine Control Law Processing. The three sub-frames constitute a frame, and the frame must execute in one time step. For the vehicle to complete a trajectory from the start of descent to landing, a large number of frames must be executed. A trajectory is considered successful if the correct commands are given to the engines to bring the vehicle in alignment with the velocity altitude contour. The velocity altitude contour is defined before the trajectory is started and is designed to bring the vehicle down in the safest and most fuel efficient manner. The starting conditions for the vehicle are called run parameters and include starting velocity, altitude, attitude, and rotation rate of the vehicle.

For the purposes of this project, GCS will run in conjunction with a simulator which is known as GCS_SIM. GCS accesses the run parameters from regions of memory which are common with GCS_SIM. GCS_SIM provides sensor values of current acceleration, altitude, rotation rate, temperature, touchdown position, and velocity. These readings are also stored in the common regions of memory. All data which is passed between GCS and GCS_SIM occurs during the execution of the rendezvous routine. The rendezvous routine is called by GCS after every sub-frame. The GCS controls three opposed pairs of roll engines, three axial thrust engines, and a parachute release actuator during the terminal descent. The descent is

[1] A safe landing is defined as one in which the lander touches down with velocity ≤ 6.2 meters/second normal to the surface, velocity ≤ 3.1 meters/second parallel to the surface, and an angle between the x-axis of the vehicle and the gravity vector $\leq$ some delta.

Table 1: Guidance Phases

PHASE	STATE	EVENT	NEXT PHASE	NEXT STATE
1	Chute attached Engines off Touch down not sensed	Altitude for turning engines on is sensed	2	Chute attached Engines on Touch down not sensed
2	Chute attached Engines on Touch down not sensed	Axial engines become hot and the chute is released	3	Chute Released Axial Engines Hot Touch down not sensed
3	Chute released Axial Engines Hot Touch down not sensed	Altitude for turning engines off is sensed	4	Chute Released Engines off Touch down not sensed
3	Chute released Axial Engines Hot Touch down not sensed	Touch down is sensed	End GCS	Chute Released Engines off Touch down sensed
4	Chute released Engines off Touch down not sensed	Touch down is sensed	End GCS	Chute Released Engines off Touch down sensed

divided into four phases as illustrated in Table 1.

The *GCS Development Specification* does not require any error handling. Any error handling which is present in an individual implementation must allow for GCS_SIM to override it. For more information on this subject see the implementation notes in the *GCS Development Specification.*

Since the functions of GCS impact the safe landing of the planetary landing vehicle, the software is classified by DO-178A standards as criticality level 1 software.

3 General Strategy

This software verification plan governs the formal review and testing of three implementations of GCS currently being developed at RTI. The three implementations are being developed independently of each other with no communication about the project among the three programmers. All three implementations will undergo the testing procedures described in this plan. A unique tester is assigned to each implementation and is instructed not to discuss the implementation with any of the other testers or programmers. A review team, consisting of the programmer, the tester, a Software Quality Assurance (SQA) representative, and a person familiar with the GCS Development Specifications (user/analyst) will attend all verification reviews and testing reviews. The same SQA representative and user/analyst will participate in the reviews for all three implementations. The *GCS Plan for Software Aspects of Certification* gives more information about the interaction of project personnel.

3.1 Participation of SQA

Participation of the SQA representative is an important aspect of the verification process. The purpose of the SQA function is to promote product quality by ensuring that all development, verification, and configuration management activities and products adhere to published policies, procedures, and standards. The *Software Quality Assurance Plan for GCS* describes the SQA activities associated with the software verification. The SQA representative works closely with the testers and ensures that they follow the procedures which are outlined here. The SQA representative determines when the programmer and/or tester are ready to advance to the next development phase. The *SQA Plan for GCS* also gives a description of the Test Completion/Readiness Reviews which are held between the different phases of testing.

3.2 Static Testing Techniques

Both static and dynamic verification techniques will be used for testing the implementations of GCS. Static analysis is "the process of evaluating a pro-

Table 2: Black-box and White-box Testing by Testing Phase

Testing Phase[2]	Black-box Testing	White-box Testing
Sub-frame	x	x
Frame	x	
System	x	

gram without executing the program."[3] Dynamic analysis is "the process of evaluating a program based on execution of the program."[3] The static verification techniques which will be utilized are the design walk-through and the code walk-through. A walk-through is defined as "a review process in which a designer or programmer leads one or more other members of the development team through a segment of design or code that he or she has written, while the other members ask questions and make comments about technique, style, and possible errors, violation of development standards, and other problems."[3] The walk-through is not meant to impose the style of the rest of the development team on the product nor to provide solutions to problems. Errors are identified during the walk-through and are resolved by the programmer after the review. The walk-through is distinguished from the inspection by having the author present and from the formal review by not having the customer present. For the purposes of this verification plan, the design walk-through will be referred to as the design review and the code walk-through as the code review.

3.3 Dynamic Testing Techniques

The DO-178A guidelines call for two different types of testing: requirements-based (black-box) and software structure-based (white-box). Both types of testing will be used for GCS as illustrated in Table 2. *The Art of Software Testing*, by Glenford Myers[4], has been relied on heavily in this plan for descriptions of testing techniques.

[2] Module test cases are designed by the programmer and are discussed separately.

3.3.1 Black-box Testing Techniques

Black-box testing may also be called data-driven or input/output-driven testing.[4, page 8] The tester designs test cases by looking at the *GCS Development Specification* and considering only the input and output of the given segment of code without regard to the internal content of the code. The given segment of the code may be a module, a group of modules, or a whole program depending on the level of testing which is being performed. For GCS, black-box testing will be performed at the sub-frame, frame, and system level. Since the *GCS Development Specification* will be used in creating test cases, the testers will be able to develop sufficient test cases to address all of the requirements contained in the *GCS Development Specification*, thus achieving 100% requirements coverage.

There are systematic methods as well as ad-hoc methods for determining the input for black-box test cases. Myers[4] uses the following classifications.

Systematic Methods	Ad-hoc Methods
Equivalence Partitioning	Error Guessing
Boundary-value Analysis	Random
Stress	
Cause-effect Graphing	

The methods of boundary-value analysis, error guessing, and random will be used along with stress testing which is a variation of boundary-value analysis. A discussion of equivalence partitioning is included here since boundary-value analysis utilizes equivalence classes.

3.3.1.1 Equivalence Partitioning The purpose of equivalence partitioning is to develop a minimal set of test cases that invoke as many different input conditions as possible. This is achieved by partitioning the input domain into equivalence classes (see Figure 2). The definition of equivalence class implies that any one input value from the class is as likely to detect a particular error as any other input value in that class. Therefore, a test case using one member of the class, makes testing the other members unnecessary. To identify equivalence classes, the test case designer examines each input to the program and divides it into multiple classes, depending on how

```
Program
    input a
        valid classes
            1
            :
            n
        invalid classes
            1
            :
            n
    input b
    :
    input n
```

Figure 2: Equivalence Class Hierarchy

the input is handled. Input values which would be treated the same way belong in one class. Two types of equivalence classes can be used: valid and invalid, where invalid classes consist of unexpected or invalid conditions. Each input will probably have more than two equivalence classes. There are no formal rules to follow when identifying equivalence classes but the tester should err on the side of caution and identify more equivalence classes when there is a question about input values being treated in the same manner. It should also be noted that equivalence classes for testing are not as formal as those defined in mathematics because they can overlap.[4, page 45]

After the equivalence classes for all inputs are identified, the test case designer identifies the test cases by using the following procedure. For each test case input, only one value is chosen. Therefore if an input has multiple classes identified with it, there must be at least one test case for each different class.

1. Write a test case to cover as many valid classes for different inputs as possible.

2. Repeat until all valid classes are covered.

3. Write a test case to cover one invalid class.

4. Repeat until all invalid classes are covered.

An example of the equivalence partitioning for a GCS process can be found in Appendix A.

The strength of equivalence partitioning is that it yields a set of test cases which avoids repetition and covers error conditions. Its weakness is that it does not combine specific inputs which can often hide errors or take advantage of boundary conditions.

3.3.1.2 Boundary-value analysis Boundary-value analysis utilizes the high payoff boundary conditions which often detect more errors. To design boundary-value test cases, the equivalence classes for input are identified in the same manner as for equivalence partitioning. In addition, a set of equivalence classes of the *output* is identified. The necessary input values to achieve that class of output values are then recorded. The test case designer is also required to create equivalence classes for pseudo boundary conditions, i.e. conditions which may make a difference based on knowledge of the application. For GCS, an example of a pseudo boundary condition is: the small size of the data element AR_COUNTER causes an incorrectly calculated altitude when both the altimeter radar frequency and the relative altitude are high. When the boundary-value test cases are designed, the boundary values of the equivalence classes are selected to be in the test cases. The example in Appendix A shows the equivalence classes for one of the processes of GCS and includes the pseudo boundary conditions plus equivalence classes for the output. Boundary-value analysis produces high-yield[4] test cases. It requires practice and knowledge of the software application. It does not have a method for combining inputs, except where consideration of particular output values causes inputs to be combined. For GCS, the method of error guessing will be used to help detect errors which can be hidden by a combination of inputs.

3.3.1.3 Stress Stress condition test cases are usually meant to overload the system[5, page 108]. For GCS, sensor values are examples of stress conditions. The stress test cases carry the notion of invalid equivalence classes (from boundary-value analysis) farther by combining selected invalid inputs. Since error handling is not required by the *GCS Development Specification*, stress testing may correctly cause the planetary lander to crash.

3.3.1.4 Error Guessing Error guessing is an ad-hoc type of test case design which utilizes the tester's experience and intuition to design test cases. It requires a small amount of effort and yields a high-payoff. When performing error guessing test case design, the tester constructs test cases by selecting any set of inputs which might produce errors. Often the tester is subconsciously using other methods of test case design such as boundary-value analysis. Error guessing does require the skills of an experienced tester.

3.3.1.5 Random Random testing uses randomly generated input for test cases. Random testing has the potential for a high-payoff because conditions which no one might have considered are generated; it can be especially useful during system-level stress testing.[6, page 68] However, random testing is not systematic, there can be repetitiveness in the test cases, making it a less efficient method for most levels of testing. GCS will use random testing for system-level test cases. Some random test generation schemes use a statistical basis for picking the input values for a test case. For GCS, the randomly generated inputs will be picked based on the usage distribution[3] across the input space.

3.3.2 White-box Testing Techniques

White-box testing can also be considered logic-driven testing[4, page 9] because it requires the tester to examine the structure of the code while designing and executing the test cases. White-box testing can be performed at any level but is typically performed at the lowest level of testing so that the amount of code being examined is manageable. For GCS, white-box testing will be performed only at the sub-frame level.

[3]The GCS input space is described in *Usage Distribution for the Guidance and Control Software* by B. Ed Withers, to be published.

An important aspect of white-box testing is determining what comprises adequate code coverage. There are many different criteria for adequate code coverage. The criteria range from simple statement coverage, where every statement in the program is executed at least once, to complete path coverage, where every possible combination of statements is executed. See [4, pages 37–44] for an excellent description of the different criteria of adequate coverage. The criteria used for GCS will be multiple-condition coverage, which requires all combinations of input at each decision statement to be exercised. A decision statement can be a single branch such as a simple if-then statement or a branch which has more than one possible condition, such as a compound if-then, a nested if-then, or a switch statement. All entry points to the code are also required to be exercised. The development of white-box test cases might be considered part of static testing since errors can be discovered while formulating the test cases. For instance, it might be determined that part of the code is unreachable. Most errors, however, will be discovered during the execution of the code; therefore, white-box testing will be considered a dynamic testing technique for this project.

Since each implementation of code will be unique for GCS, the white-box test cases will be specific for each implementation and constructed by the individual testers.

3.4 Module Testing

The test cases for the module testing will be generated by the individual programmers and thus will vary with each implementation of code. The *GCS Development Specification* does not constrain the size of a software module within the separate sub-frames, so the size of a module is determined by the programmer. The programmer conducts his own module testing and is only responsible for conducting a minimum set of test cases using any technique desired.

3.5 Stopping Rules

A stopping rule is a guideline which determines when a testing phase is done. For dynamic test techniques, there are actually two stopping rules. The first stopping rule determines when test case creation is finished. The second stopping rule determines when test case execution is finished. Many test techniques have stopping rules built in. For example, multiple-condition coverage implies a stopping rule by requiring 100% multiple-condition coverage for test case creation and test case execution. Other stopping rules are less obvious. The stopping rules for executing black-box sub-frame and frame testing are that all test cases must execute correctly. Executing correctly means the expected results are achieved. Thus, the burden of thorough testing is focused on the test creation stopping rule. For GCS, the test creation stopping rule decrees there should be 100% requirements coverage. The stress condition testing for system testing will also follow the test execution stopping rule that all test cases must execute correctly. The test creation stopping rule decrees that fifty stress test cases must be developed. The random testing part of system testing will use an adapted form of a mean time between failure (MTBF) rule for test execution. Fifty consecutive random test cases must execute correctly. If one test case fails, the fault is corrected and the count to fifty is restarted while new test cases are executed. Module testing will use test creation and test execution stopping rules that simply require at least three test cases per module for a total of at least twenty test cases per sub-frame.

3.6 Test Case Design

3.6.1 GCS Black-box Test Case Design

All black-box test cases will be designed by the three testers as a group before sub-frame testing is started for any implementation. For ease of test case design and efficiency, the black-box test cases will be designed in the reverse order from that in which they will be executed.

Test Case Design	Test Case Execution
system	sub-frame
frame	frame
sub-frame	system

The system test cases will be designed first because they consist of trajectories whose input is run parameters. The expected results for these test cases will consist of a determination of whether the vehicle should be able to land safely[4] with the given initial conditions and an expected point of crossing the velocity altitude contour. The system test cases will be validated using two prototype implementations which were developed as an earlier part of the GCS project. The frame test cases will be derived by using some of the intermediate results after the system test cases are executed on the prototypes as input for the frame. The intermediate results will also help to determine the output for the frame. In the same manner, the sub-frame test cases will be derived.

GCS black-box test cases at the sub-frame and frame level will be designed using a combination of the boundary-value analysis and error-guessing methods. The testers will construct a chart of all the equivalence classes for the input and output of each sub-frame and frame. No distinction will be made between invalid and valid equivalence classes, since GCS_SIM prevents impossible conditions and error handling is not required. Those classes will be combined to make test cases, utilizing the intermediate results from the system testing, as described above. Full requirements coverage will be achieved by a combination of determining pseudo boundary conditions, determining expected results, and determining the necessary input to achieve a specific output. The *GCS Development Specification* is divided into different processes. The processes make up the sub-frame. Each requirement is tied to a specific process, except those related to process control, timing, and memory. If there is any doubt that all requirements are covered, additional test cases will be created by the testers. The testers may also use error guessing at their discretion to create additional test cases.

GCS black-box test cases at the system level will be divided between stress condition test cases and randomly generated test cases. Some of the randomly generated test cases may include stress conditions. All of the input to these test cases will consist of run parameters and the expected results will be as described above. Some intermediate results may also be examined for stress cases. Those intermediate results will be described in the individual test case.

[4]See footnote 1.

A DTM[5] benchmark file will be created for each test case. More specific procedures for black-box test case design will be added to a later release of this document.

3.6.2 GCS White-box Test Case Design

The GCS white-box test cases will be designed during the sub-frame testing phase of the development cycle by the individual tester for each implementation. The general procedures for test case development are described in the Sub-Frame Testing section of this document.

3.7 Verification Technique Summary

The static technique of a walk-through will be used for reviewing the design and code. Table 3 shows a summary of the dynamic test techniques which will be used for verifying each implementation of GCS.

[5] See the Tools section for a description of the DEC/Test Manager(DTM).

Table 3: Dynamic Testing Summary

Test Phase	Technique	Performed By	(a)Test Creation Stopping Rule (b)Test Execution Stopping Rule
Module	Programmer's Choice	Prog.	(a)$\geq$ 3 Test Cases per Module for a total of $\geq$ 20 Test Cases per Sub-Frame (b)All Test Cases Execute Correctly*
Sub-Frame	White-Box: Multiple-Condition Coverage	Tester	(a)100% Multiple-Condition Coverage (b)100% Statement Coverage All Test Cases Execute Correctly
	Black-Box: Boundary-Value Analysis	Tester	(a)100% Requirements Coverage (b)All Test Cases Execute Correctly
Frame	Black-Box: Boundary-Value Analysis	Tester	(a)100% Requirements Coverage (b)All Test Cases Execute Correctly
System	Black-Box: Stress	Tester	(a)50 Stress Test Cases (b)All Test Cases Execute Correctly
	Black-Box: Random	Tester	(a)Minimum 50 Random Test Cases (b)50 Consecutive Test Cases Execute Correctly

* - All Test Cases Execute Correctly means that the expected results were achieved for all test cases. If any changes needed to be made to the code, the test cases were re-run.

4 Tools

Several tools will be used throughout the verification process. Some of these tools are software packages and some are paper forms which help ensure verification is performed in an organized and standard manner. The tools are presented in the order in which they will be used on the project. Where appropriate, validation of the tool is discussed. See the *GCS Support/Development System Configuration Description* for information about the specific versions of the tools to be used.

4.1 GCS Requirements Traceability Matrix

A matrix will be used to demonstrate requirements traceability. The GCS Requirements Traceability Matrix will provide verification continuity through all phases of development. This matrix consists of a row for each requirement found in the *GCS Development Specification* and a column for each development phase.[6] During the review associated with each phase of development, the material under review will be checked through this matrix to ensure that all requirements are accounted for. Violations will be marked by inserting the appropriate GCS problem report number. A violation occurs when the requirement is not covered or is covered inadequately or incorrectly. Compliance will be marked by inserting the appropriate identifier for the design section, code section, or test case. There will be one copy of the matrix for each implementation of code; and that copy will be used during the design review, all code reviews, and all test readiness and completion reviews. If any requirements change, the matrix must be changed to reflect them. Since the GCS Requirements Traceability Matrix will be under configuration control as part of this verification plan, any changes made to it will have to be approved as described in the *GCS Configuration Management Plan*. A copy of the matrix can be found on page 97 in Appendix B.

[6]There is not a column in the matrix for Module Testing since those test cases are not necessarily linked to requirements. The column for sub-frame testing only applies to the black-box testing, since white-box test cases do not involve the requirements.

4.2 Checklists

Checklists are used as tools to help the design review and code review process. The checklists contain questions which were chosen to bring out potential problems in the design or code being reviewed. The first answer column of the checklist contains the desirable answer to the question. If the answer to the question lies in the second column then the reviewers must consider whether there is a problem in the design or code. As the questions are just guidelines, an answer in the second column does not necessarily signify that an error has occurred. Copies of the GCS Design Review Checklist and the GCS Code Review Checklist can be found on pages 105 and 109 in Appendix B .

4.3 GCS Problem Report Form

The GCS Problem Report (PR) Form is the tool used for error data collection. All errors detected during the development of an implementation will be recorded on a PR form. A sample form is included on page 101 in Appendix B. All errors detected in the specification, the design, the code, or any formal test case[7] are tracked. Instructions for filling out the form can be found in Programmer Instruction #8 – Completing the GCS Problem Report Form. (See the *GCS Programmer's Manual.*) The *SQA Plan for GCS*, and the *GCS Configuration Management Plan* also address the use of the GCS Problem Report Form.

4.4 DEC Code Management System

The Code Management System (CMS) by Digital Equipment Corporation (DEC) will be used to control the versions of the design and code for each implementation. A new version of the design and code will be created after every verification milestone as well as when changes are made to it. The configuration management procedure as described in the *GCS Configuration Management Plan* calls for the programmer to submit the item

[7]Formal test cases are those executed by the tester. The module test cases which the programmer writes and executes are not considered formal; therefore a change to those test cases does not require a PR form to be filed.

to be configured to the configuration manager or SQA representative. The programmer must also check out a configured item before changes can be made to it. During Module Testing, the programmer will have his own CMS library and will be able to check code in and out himself.

4.5 GCS Module Test Log

The GCS Module Test Log will be used to keep track of the testing performed during module testing. One log will be used for each module. The inputs and outputs of the module will be recorded on the first page of the log. The input and expected results of each test case will be recorded before the testing is performed. After the test case has been executed, the actual results will be recorded on the test log. The number of any GCS Problem Report Forms filled out during module testing will also be recorded with the appropriate test case. A copy of the test log can be found on page 115 in Appendix B. Programmer Instruction #9 – Using the GCS Module Test Log, which is part of the *GCS Programmer's Manual*, explains how to fill out the test log.

4.6 Analysis of Complexity Tool

The McCabe & Associates' Analysis of Complexity Tool (ACT) will be used to help the testers construct white-box test cases. ACT will generate an annotated control flow graph of each module of code and show the code complexity V(G). The code complexity is equivalent to the minimum number of paths necessary to achieve 100% multiple-condition coverage. ACT will determine a set of minimum test paths by applying McCabe's method. See Appendix C for a discussion of code complexity and McCabe's method. ACT will produce a list of the decisions and the conditions which should be satisfied to achieve each path. The tester still has to create the test cases by determining the inputs necessary to achieve the desired conditions.

The use of ACT cannot adversely affect the reliability of the code since it does not interfere with the execution of the code. The only risk involved in using ACT is that an error in ACT may prevent an accurate testing coverage measurement. The specified coverage criteria is 100% multiple-condition coverage.

The GCS project has on file a statement for Tom McCabe & Associates about the development and testing of ACT. In addition, the GCS testers will do a sample test of ACT as part of a training session on performing white-box testing. ACT will be used on three modules from the simulator, one of which has low complexity (less than 5), one of which has medium complexity (between 5 and 15), and one of which has high complexity (greater than 15). All different FORTRAN decision constructs will be represented in the selected code. Together the testers will manually calculate the code complexity and determine the test paths according to the McCabe method. If the complexity and test paths do not match the results given by ACT, any errors will be reported to McCabe & Associates, and further testing will be done. If the complexity and test paths do match, ACT will be considered reliable for generating test paths.

Since ACT does not provide any coverage measurement during test execution, another coverage tool will be used to ensure that all of the identified paths in the code were really traversed. The tester will check to make sure that all appropriate statements, given the input and test path specified by ACT, were executed. If the coverage measurement disagrees with what ACT predicted, the tester will walk through the code, executing it manually to determine if there is a problem with the code or with ACT. This procedure will serve as a second validation check of ACT.

4.7 Coverage Information Tool

A coverage information tool will be identified before sub-frame testing begins. It will be used to collect coverage information during all formal[8] test execution.

4.8 DEC/Test Manager

The DEC/Test Manager (DTM) will be used for organization and configuration control of the test cases. DTM is a software development and maintenance tool that organizes software regression tests and test results.

[8]Formal test cases are executed by the tester.

DTM provides an efficient, automated way to run, review, and store tests.[7] Several different kinds of files are associated with DTM. **Template** files are user-specified command files that DTM invokes to start the test. Typically they contain operating system commands which set up the test environment and a call to the test driver or the actual program being tested. **Results** files contain the output after DTM has run the test. They are time stamped. **Benchmark** files are the standard files to which the **results** files are compared. To run a test with DTM the following general procedure is followed.

1. A **test description** is created by naming the test, specifying an associated **template** file, and entering a comment to briefly explain what the test does.

2. Several related **test descriptions** are combined into a **test collection.**

3. The **test collection** is run interactively or in batch mode.

4. The **results** files (one for each **test description**) are reviewed in DTM.

5. When a satisfactory run of the test is completed, a **benchmark** file is created for each **test description** from the corresponding **results** file.

6. The next time a test is run, DTM automatically compares the current **results** file for each **test description** to its **benchmark** file and shows any differences.

7. If a change needs to be made to a test, the old **test collection** must be recreated after the change is made to the **template** file for the appropriate **test description**. The operator enters a comment describing the reason for the change when recreating the **test collection**.

On this project, one **benchmark** file will be created for each **test description,** and the **results** files of all three implementations will be compared to the same **benchmark** file. DTM will serve as the test log for all phases of testing which are conducted by the testers. Validation of this tool is not necessary because it is only a test management tool and does not affect the outcome of the tests.

4.9 GCS_SIM

Each implementation of software will be tested using the simulator, known as GCS_SIM. A GCS implementation interacts with GCS_SIM when it calls the rendezvous routine after every sub-frame. The rendezvous routine provides synchronization to ensure that all data in the common regions of memory is stable when accessed by a GCS implementation by allowing only one program to have access to the data at any given time. When the rendezvous routine is called by a GCS implementation, it turns control over to GCS_SIM. When GCS_SIM is finished, the rendezvous routine returns control to the GCS implementation. The rendezvous routine is also called at the beginning of the trajectory to initialize the run parameters. GCS_SIM utilizes several data files. INITIAL_CONSTANTS.DAT provides the run parameters, which the testers will change for different test cases. TABULAR_DATA.DAT provides variables in tabular form which are needed by GCS_SIM but are not utilized by the GCS implementation. USAGE_DISTRIBUTION.DAT is a file which contains information on the input space distribution and will be used by the testers to help select the random test cases for system testing. GCS_SIM itself displays little information. It only displays frame numbers and error messages when variables go out of range. All other information is written to GCS_TRACK.DAT. The operator can use the trajectory plotter program or the display program to examine the data in GCS_TRACK.DAT. The trajectory plotter takes parameters for the starting frame number, the ending frame number, the number of frames to skip, and the delay between frames. The plotter program can run from start to finish or can use a stepping function so that the operator signifies when she is ready to continue. The plotter program shows the desirable velocity altitude contour and shows the descent of the vehicle on the same graph. The display program shows the position of the vehicle relative to the planet's x,y, and z axes. A description of GCS_SIM and its validation can be found in the *GCS Support/Development System Configuration Description.* A series of test drivers will also be written by the project management personnel to aid in testing at the sub-frame and frame level. These test drivers will interact with the simulator. The next release of this document will contain more information about the test drivers.

5 Development Phases

The sub-section for each development phase contains a brief prose description of the activities which occur during the phase, an overview flowchart of the phase, and an itemized description of the verification activities conducted during the phase. The itemized description of the phase is separated into the following parts.

- **Purpose** – A one sentence description of the main purpose of this set of verification activities.

- **Method** – The name of the verification technique which is being used.

- **Test Creation Stopping Rule** – The rule that determines when enough test cases have been created.

- **Test Execution Stopping Rule** – The rule that determines when the described verification activities end.

- **Roles** – Describes what activities the programmer, tester, user/analyst, and SQA representative perform during the phase.

- **Input** – Lists the input to the procedures which are conducted during this phase. As described earlier, many of the black-box test cases are written before testing begins. Those black-box test cases are considered input. White-box test cases and module test cases whose development is considered part of the procedures are not considered input.

- **Output** – Lists the output from the procedures which are conducted during the phase.

- **Test Readiness Review** – The Test Readiness Review is considered part of the verification procedure but is listed separately because it is conducted by the SQA representative. A brief description of the checks conducted is included.

- **Test Completion Review** – The Test Completion Review is considered part of the verification procedure but is listed separately because it is conducted by the SQA representative. A brief description of the checks conducted is included.
- **General Procedures** – An enumerated listing of the general verification procedures.
- **Tools** – Lists the tools used during the phase to help the verification process.

5.1 Design Phase

5.1.1 Overview

During the design phase, the static analysis technique of a design walkthrough or review will be employed. The purposes of the Design Review are to verify that the requirements have been correctly translated into the design, no additional functionality has been added, the interfaces are fully and correctly specified, and the design standards have been followed. Only one Design Review will be held per implementation, instead of a separate Preliminary Design Review and Critical Design Review. The size of the project makes it feasible to conduct the entire Design Review in three hours. To aid in the Design Review process, a GCS Design Review Checklist has been defined.[9] A copy of the GCS Design Review Checklist can be found on page 105 in Appendix B. The checklist together with the GCS Requirements Traceability Matrix will guide the review.

There is a classification for design elements called derived requirements. As interpreted by the FAA, a derived requirement is an element of a lower order of decomposition that does not trace to the next higher level but is required for proper operation of the system. Derived requirements should be justified because adding unnecessary functionality to a design contributes to unreliability. When derived requirements are introduced they should be noted so that they can be traced through the later stages of the development cycle. The *GCS Development Specification* is very detailed and already imposes many constraints on the designer, reducing the likelihood that there will be any derived requirements. The constraints imposed are caused by developing multiple implementations which must look identical on the sub-frame level and by the well-defined interface with GCS_SIM. It is the review team's responsibility to determine if any derived requirements exist. If the review team determines that there are derived requirements then they should be added to GCS Requirements Traceability Matrix for

[9]The GCS Design Review Checklist was compiled using example checklists obtained from the Software Product Assurance Group at Jet Propulsion Laboratory, *The Art of Software Testing*[4], and *The Complete Guide to Software Testing*[8]. The authors of this verification plan chose items for the checklist based on their experiences at Research Triangle Institute and elsewhere. To make the checklist items more useful, they were made specific for this project.

that implementation and be tracked in the same manner as the other requirements.

The attendees to the review are the programmer, tester, user/analyst, and SQA representative. While all attendees have different specialty areas, they will all participate in the review process and are encouraged to comment outside their specialty area. Figure 3 shows the Design Review Procedure. The specifics of the Design Review are described following the figure. Any additional SQA procedures are described in the *SQA Plan for GCS.*

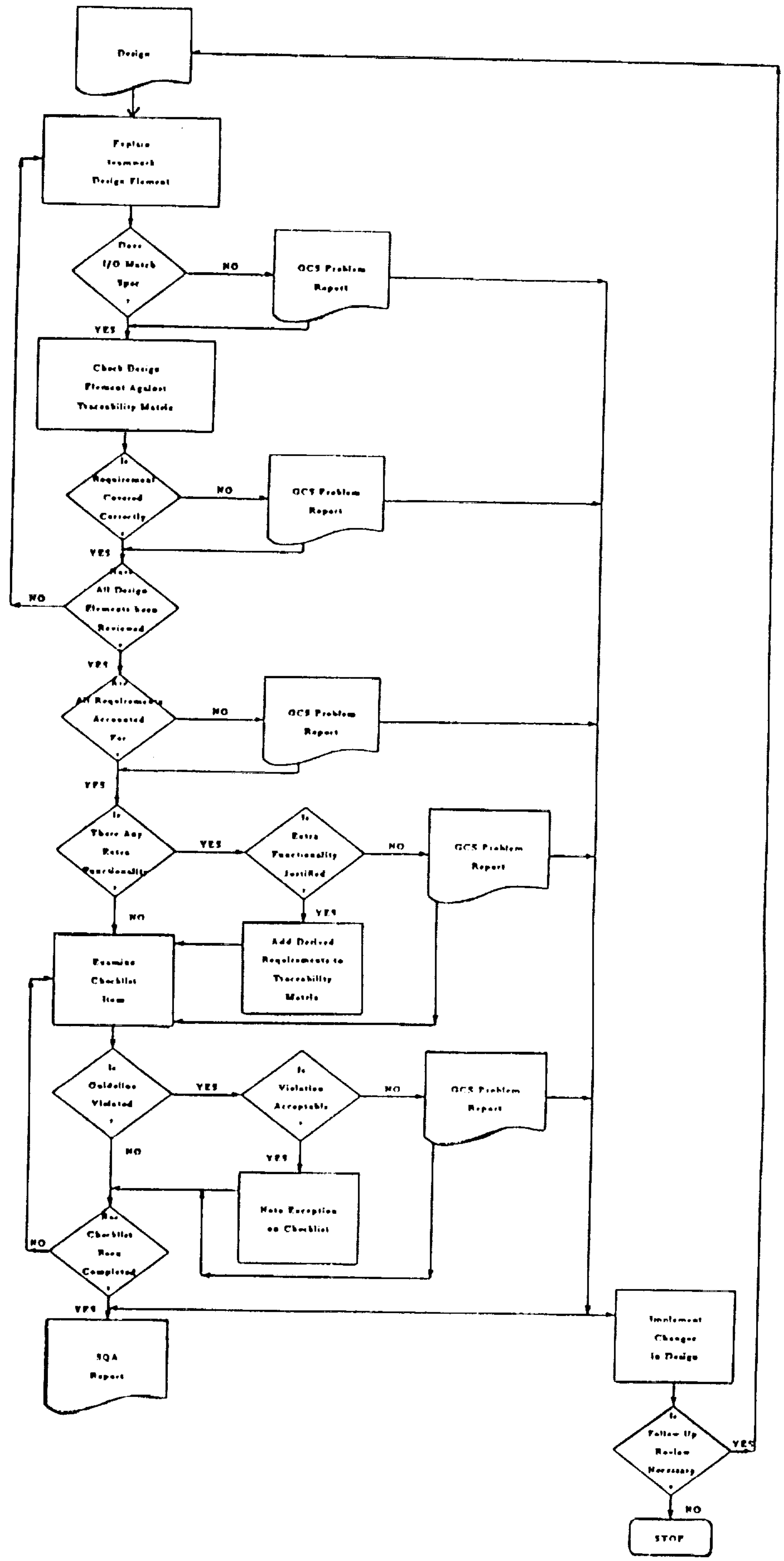

Figure 3: Design Review Procedure

5.1.2 Design Review Description

- Purpose
  - Verify that the requirements have been correctly translated into the design, no additional functionality has been added, the interfaces are fully and correctly specified, and the design standards have been followed.
- Method
  - Design Walk-through
- Test Creation Stopping Rule
  - N/A
- Test Execution Stopping Rule
  - All GCS Design Review Checklist questions have been answered.
  - All Requirements from the Traceability Matrix are accounted for in the design and no extra functionality is present.
- Roles
  - Programmer: Only the design is under review, not the programmer. The programmer is present to interpret the design and answer any questions about it.
  - Tester: The tester's responsibility is to ensure that the purpose of the review is fulfilled. The tester also fills out any necessary problem reports noting only the problem, not the solution.
  - User/Analyst: The user/analyst is the specification expert and answers any questions about the requirements which arise.
  - SQA Representative: The SQA representative ensures that the tester follows the procedures documented in this plan and acts as moderator to determine the order of the review. The SQA representative also is in charge of the GCS Design Review Checklist and the GCS Requirements Traceability Matrix. Please see the *SQA Plan for GCS* for any additional responsibilities.

- Input
  1. *GCS Development Specification* including Modifications
  2. Programmer Instruction #5 – Use of Error Handlers (See *GCS Programmer's Manual.*)
  3. Programmer Instruction #6 – Design Document Outline (See *GCS Programmer's Manual.*)
  4. Programmer Instruction #7 – Design Standards (See *GCS Programmer's Manual.*)
  5. GCS Design – includes Programmer's *GCS Design Description* and team*work*[10] Design Diagrams
  6. GCS Design Review Checklist (See Appendix B page 105.)
  7. GCS Requirements Traceability Matrix (See Appendix B page 97.)
- Output
  1. Completed GCS Design Review Checklist
  2. Partially Completed GCS Requirements Traceability Matrix
  3. Completed GCS Problem Report Forms (See Appendix B page 101.)
  4. GCS Design under Configuration Control
- General Procedures
  1. The programmer and tester decide when the design is ready for review.
  2. The tester arranges the date, time, and place for the review.
  3. The programmer makes one copy of the design document and team*work* design diagrams. The copy and original[11] are circulated to the review team 48 hours before the review.

[10]Team*work* is a registered trademark of Cadre Technologies Inc. Team*work* is a computer aided software engineering tool which allows the user to analyze or design a system using data flow diagrams or structure charts, according to the Hatley[9] method. The *GCS Plan for Aspects of Software Certification* also describes the use of team*work* on this project.

[11]Each attendee does not receive his own copy, because the document will average about 50 pages.

4. The tester brings blank copies of problem report forms to the review.
5. The SQA representative brings a blank copy of the GCS Design Review Checklist and a blank copy of the GCS Requirements Traceability Matrix.
6. The SQA representative declares the general order in which the design will be checked. It is not necessary for the review to strictly follow the specified order but the SQA representative should act as moderator to keep the review flowing as efficiently as possible.
7. The programmer starts the review by giving a brief overview of the design.
8. The programmer leads the review team through the design by explaining each team*work* design diagram.
9. All input for each process described in the *GCS Development Specification* will be checked by comparing the input table list to the input of the appropriated team*work* design element specification(s). All input to a process specification must appear as input to the team*work* design element and in the body of the design element specification. The type declaration of each input will be checked against the data element description from the *GCS Development Specification.* If unused global input is shown, it will be written on a problem report form. All local variables should be specified in lowercase.
10. All output for each process described in the *GCS Development Specification* will be checked by comparing the output table list to the output of the appropriated team*work* design element specification(s). All output from a process specification must appear as output from the team*work* design element and in the body of the design element specification. The type declaration of each output will be checked against the data element description from the *GCS Development Specification.* If unmodified global output is shown, it will be written on a problem report form.
11. The SQA representative completes the design column of the GCS

Requirements Traceability Matrix by filling in the team*work* identification number for the design element which meets the requirement. All requirements must be accounted for in the design. If a requirement is not met, a problem report must be filled out by the tester, with the problem report number recorded on the GCS Requirements Traceability Matrix by the SQA representative. The SQA representative also records derived requirements, if any, on the GCS Requirements Traceability Matrix. Derived requirements should be justified. If there is extra functionality which is not traceable to any requirement; a problem report should be written.

12. The SQA representative circles the appropriate response to each question on the GCS Design Review Checklist. If any checklist guideline is not followed, a problem report must be filled out by the tester, with the problem report number recorded on the GCS Design Review Checklist by the SQA representative, or the SQA representative must initial the guideline to show that the violation was acceptable.

13. After the entire design has been traversed and the GCS Design Review Checklist and GCS Requirements Traceability Matrix have been completed, the SQA representative will determine if a follow-up review is necessary. If one is necessary, the SQA representative will inform the programmer of what changes need to be made to the design for the follow-up review.

14. After the review(s) is finished, all problem reports must be completed. The programmer is responsible for completing those problem reports whose problems originated in the design, after he has fixed the design. The user/analyst is responsible for completing those problem reports whose problems originated in the specification, after he has made the necessary specification modifications and generated a problem report form for the design, if necessary.

15. The SQA representative is responsible for ensuring that all problem reports are satisfactorily completed.

16. After all design changes are made, the design will be placed under configuration control.

- Tools
  - GCS Requirements Traceability Matrix
  - GCS Design Review Checklist
  - GCS Problem Report Form
  - DEC Code Management System (CMS)

5.2 Code Phase

5.2.1 Overview

During the code phase, the static analysis technique of a code walkthrough or review will be employed. The purposes of the Code Review are to verify that the design, which includes the interface, hierarchy, and pseudo-code, has been correctly implemented and the coding standards have been followed. A series of code reviews will be held after all of the code has been written and compiled without error.[12] All code modules will be subjected to this Code Review. In addition, any module which has more than 20 lines[13] of executable code added or modified due to a single change any time during the verification process will be reviewed. The programmer will choose the modules to be reviewed at each gathering with the following constraints:

- the review should last no more than two hours, and
- all modules for one sub-frame will be reviewed before the next sub-frame is started.

To aid in the Code Review process, a GCS Code Review Checklist has been defined.[14] The checklist is specific to FORTRAN, the language in

[12]Some development cycles might call for a Code Review to be held as soon as the first module of code is ready so that the programmer would be able to apply what was learned at the review to subsequent modules. Due to the experimental nature of this project, it was decided that no Code Review should be held until all the code for that implementation is written. One of the goals of the project, as discussed in the *GCS Plan for Software Aspects of Certification*, is to compare repetitive run testing to the procedures documented in this plan. Versions of the code before and after the Code Review will be put in the repetitive run harness. It was felt that the constraint to have all of the code written before the first review was not an unreasonable one since the programmer is free to make changes to previously reviewed code modules based on techniques learned in subsequent Code Reviews.

[13]More than twenty lines of code were chosen as the boundary, because 20 lines represents a screenful of text.

[14]The GCS Code Review checklist was compiled using example checklists obtained from the Software Product Assurance Group at JPL, *The Complete Guide to Software Testing*[8], *The Art of Software Testing*[4] and a previous RTI testing experiment. (See *Software Reliability Measurement/Test Integration Techniques: Instructions for Testers* prepared by RTI for SAIC and submitted to Rome Air Development Center. The authors

which all three implementations at RTI will be coded. A copy of the GCS Code Review Checklist can be found on page 109 in Appendix B. The checklist together with the GCS Requirements Traceability Matrix will help to guide the review. The design will be the main guiding force of the review. It should be remembered that the pseudo-code of the design is merely a way to express the intentions of the designer and the actual code does not have to strictly adhere to the pseudo-code, it just needs to satisfy the same purpose. However if the designer's intentions change, the design must be changed to match the actual code.

Attendees to the Code Review are the same as those for the design review: the programmer, tester, user/analyst, and SQA representative. While all attendees have different specialty areas, they will all participate in the review process and are encouraged to comment outside their specialty area. The specifics of the Code Review process are described following the figure. Any additional SQA procedures are described in the *SQA Plan for GCS.*

of this verification plan chose items for the checklist based on their experiences at Research Triangle Institute and elsewhere. To make the checklist items more useful, they were made specific for this project.

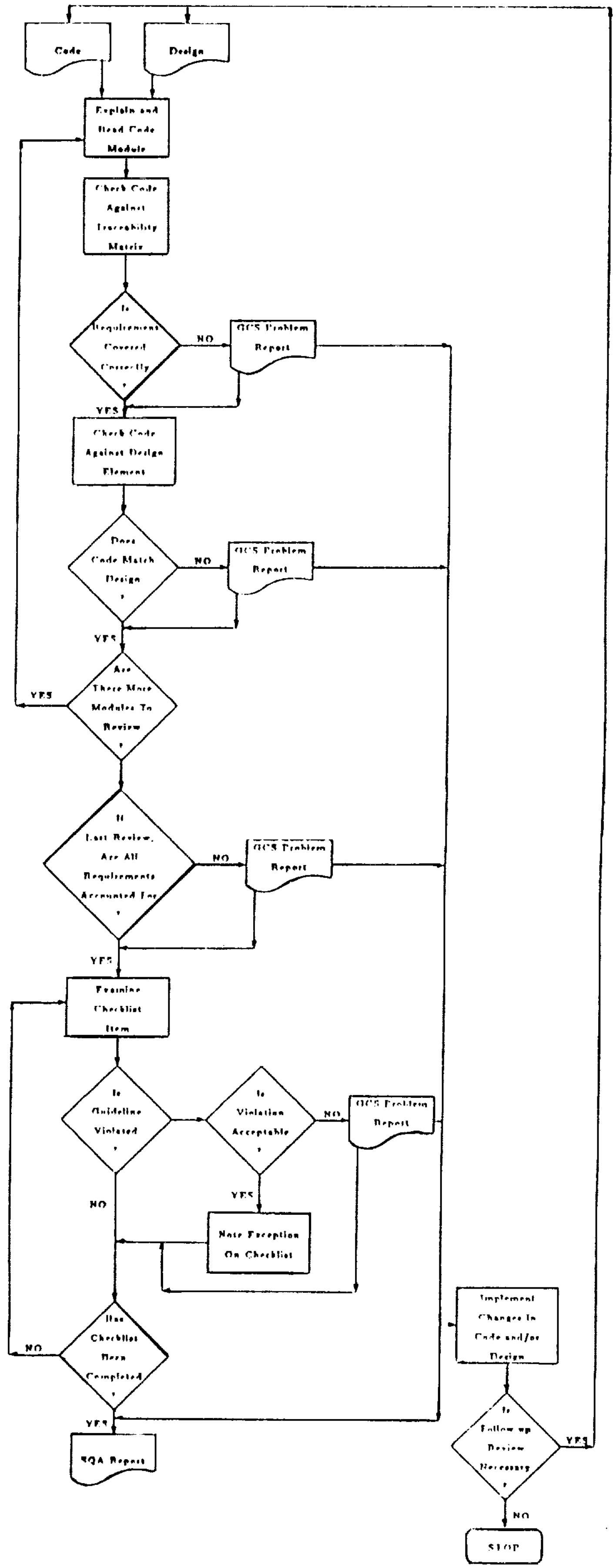

Figure 4: Code Review Procedure

5.2.2 Code Review Description

- Purpose
  - Verify that the design has been correctly implemented and the coding standards have been followed.
- Method
  - Code Walk-through
- Test Creation Stopping Rule
  - N/A
- Test Execution Stopping Rule
  - All elements of the design which are applicable to the portion of the code being reviewed have been accounted for in the code.
  - All applicable requirements from the GCS Requirements Traceability Matrix have been accounted for in the code.
  - All GCS Code Review Checklist questions have been answered.
- Roles
  - Programmer: Only the code is under review, not the programmer. The programmer is present to interpret the code and answer any questions about it.
  - Tester: The tester's responsibility is to ensure that the purpose of the review is fulfilled. The tester also fills out any necessary problem reports noting only the problem, not the solution.
  - User/Analyst: The user/analyst is the specification expert and answers any questions about the requirements which arise.
  - SQA Representative: The SQA representative ensures that the tester follows the procedures documented in this plan and acts as moderator to determine the order of the review. The SQA representative also is in charge of the GCS Code Review Checklist and the GCS Requirements Traceability Matrix. Please see the *SQA Plan for GCS* for any additional responsibilities.

- Input

  1. *GCS Development Specification* including Modifications
  2. GCS Design – includes Programmer's *GCS Design Description* and team*work* Design Diagrams
  3. Programmer Instruction #3 – Coding Standards for GCS Applications (See *GCS Programmer's Manual.*)
  4. Programmer Instruction #5 – Use of Error Handling (See *GCS Programmer's Manual.*)
  5. Cleanly Compiled GCS Code Module(s)
  6. GCS Code Review Checklist (See Appendix B page 109.)
  7. GCS Requirements Traceability Matrix – the same one used for the Design Review and previous Code Reviews (See Appendix B page 97.)

- Output

  1. Completed GCS Code Review Checklist
  2. Partially Completed GCS Requirements Traceability Matrix
  3. Completed GCS Problem Report Forms (See Appendix B page 101.)
  4. GCS Code Modules under Configuration Control

- General Procedures

  1. The programmer and tester decide when the code is ready for a review.
  2. Before the first Code Review, the programmer submits a copy of all of the code to configuration control.
  3. The tester arranges the date, time, and place for the review.
  4. The programmer gives a copy of the selected modules to each attendee at least 24 hours before the Code Review.
  5. The tester brings blank copies of problem report forms to the review.

6. The SQA representative brings a blank copy of the GCS Code Review Checklist and the GCS Requirements Traceability Matrix which was used for the Design Review and previous Code Reviews.

7. The SQA representative declares the general order in which the design will be checked. It is not necessary for the review to strictly follow the specified order, but the SQA representative should act as moderator to keep the review flowing as efficiently as possible.

8. The programmer starts the review by giving an overview of each of the code modules. The discussion should include the function of the module, how it fits in the sub-frame, and its relationship to the design.

9. The programmer leads the review team through the code by reading it aloud, line by line. Review team members should interrupt any time they have a question.

10. The SQA representative completes the appropriate code section of the GCS Requirements Traceability Matrix by filling in the name of the code module in which the requirement is satisfied. At the last Code Review for each sub-frame, the SQA representative must check to be sure that all requirements for that sub-frame have been fulfilled. If a sub-frame requirement is not met, a problem report form must be filled out by the tester with the problem report number recorded on the GCS Requirements Traceability Matrix by the SQA representative. If a sub-frame requirement is met more than once, the reviewers should determine if some functionality is being repeated and fill out a problem report form, following the previous procedure, if the duplicated functionality is inappropriate.

11. The SQA representative circles the appropriate response to each question on the GCS Code Review Checklist. If any checklist guideline is not followed, a problem report must be filled out by the tester, with the problem report number recorded on the GCS Code Review Checklist by the SQA representative, or the

SQA representative must initial the guideline to show that the violation was acceptable.

12. After all of the code has been traversed and the GCS Code Review Checklist[15] and GCS Requirements Traceability Matrix have been completed, the SQA representative will determine if a follow-up review is necessary. If one is necessary, the SQA representative will inform the programmer of what changes need to be made to the code for the follow-up review.

13. After all of the Code Reviews are complete, the SQA representative will ensure that all necessary changes have been made. The programmer is responsible for completing those problem reports whose problems originated in the code after fixing the code. If the problem originated in the design, the programmer is responsible for completing the problem report after fixing the design and then generating another problem report form for the code. This second problem report is completed by the programmer after the necessary code changes have been made. The user/analyst is responsible for completing those problem reports whose problems originated in the specification after the necessary specification modifications have been made, and then generating a problem report form for the design or code depending on which one, if any, reflects the problem first.

14. The SQA representative is responsible for ensuring that all problem reports are satisfactorily completed.

15. After all code changes are made, the code will be placed under configuration control.

- Tools
  - GCS Requirements Traceability Matrix
  - GCS Code Review Checklist
  - GCS Problem Report Form
  - DEC Code Management System (CMS)

[15]There is a separate GCS Code Review Checklist for each session.

5.3 Module Testing

5.3.1 Overview

Since the programmer is free to use any division of modules for the processing which comprises a sub-frame, the programmer will be responsible for module testing. The programmer is free to use any testing method on the code provided a minimum of twenty test cases are executed per sub-frame, including three for each module.[16] No Test Readiness Review is held for Module Testing since there are no specific guidelines for creating test cases. A Test Completion Review will be held after Module Testing is complete. During the module testing, the programmer will have a personal CMS library as described in the *GCS Configuration Management Plan*. The programmer will not be required to obtain approval to make any necessary changes to the code as long as the reason for the change is put in the comment for the CMS library and a problem report form is completed. While testing, the programmer will be required to keep a testing log, for each module, which will include information about the test case and its results, both expected and actual, and a cross reference number to any problem reports which are generated. A copy of the GCS Module Test Log can be found on page 115 in Appendix B. Detailed Instructions for completing the test log can be found in Programmer Instruction #9 - Using the GCS Module Test Log. (See the *GCS Programmer's Manual.*) Figure 5 shows the module testing procedure.

[16]With at least three test cases, the inputs can be at opposite boundaries. The twenty case minimum was chosen in the event that there are few modules in the sub-frame. For such cases, the individual modules would need more testing since they would perform more functions.

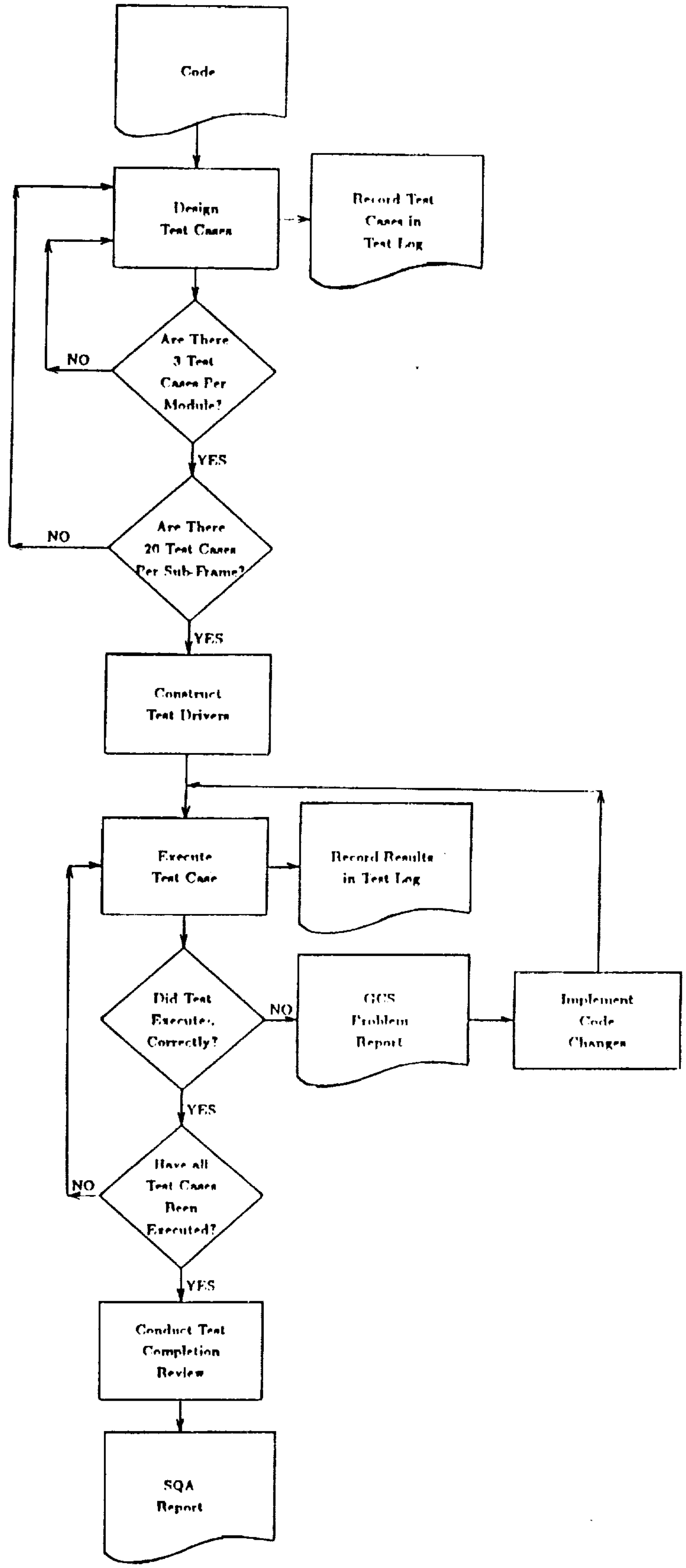

Figure 5: Module Testing Procedure

5.3.2 Module Testing Description

- Purpose
  - The programmer should be satisfied that the code correctly performs the functions specified in the design.
- Method
  - Programmer's Choice
- Test Creation Stopping Rule
  - At least three test cases per module have been created for a total of at least twenty test cases per sub-frame.
- Test Execution Stopping Rule
  - At least three test cases per module have been correctly executed for a total of at least twenty test cases per sub-frame have been correctly executed.
- Roles
  - Programmer: Designs and executes tests for each module.
  - Tester: Acts as an advisor to answer any questions about testing.
  - User/Analyst: Makes any necessary modifications to the *GCS Development Specification*. Provides a set of initial conditions which are also known as run parameters.
  - SQA Representative: Approves any changes which need to be made to the programmer's design and conducts the Test Completion Review.
- Input
  1. *GCS Development Specification* including Modifications
  2. GCS Design – includes Programmer's *GCS Design Description* and team*work* Design Diagrams
  3. GCS Code Modules under Configuration Control

4. Programmer Instruction #3 – Coding Standards for GCS Applications (See *GCS Programmer's Manual.*)

- Output
  1. Completed GCS Module Test Logs (See Appendix B page 115.)
  2. Completed GCS Problem Report Forms (See Appendix B page 101.)
  3. GCS Code Modules under Configuration Control
- Test Readiness Review
  - None
- Test Completion Review checks that
  - All test cases are documented in the GCS Module Test Log along with their expected and actual results.
  - All discrepancies between actual and expected test results are documented in a problem report.
  - The expected results were calculated correctly[17] and the calculations are included in the GCS Module Test Log.
  - The problem report number is contained in the GCS Module Test Log.
  - All problems have been corrected and are appropriately marked in the code as specified in Programmer Instructions #3 – Coding Standards for GCS Applications. (See *GCS Programmer's Manual*)
  - Any changes to test cases (including the addition of new test cases) are documented in the *Notes* section of the log.
  - The programmer has executed a minimum of three test cases per module and twenty test cases per sub-frame.
- General Procedures

[17]Due to resource constraints at most ten test cases per module and at most a total of forty per sub-frame will have their expected results checked.

1. The user/analyst provides the programmer with one set of run parameters which the programmer can use during testing.
2. The programmer checks the code into the personal CMS library after all of the Code Review changes have been made.
3. The programmer initiates a test log for each module by writing down the test cases and expected results.
4. The programmer constructs any necessary test drivers to exercise the module in a stand-alone fashion. The programmer does *not* have access to GCS_SIM or any of the test drivers which were discussed in the tools section.
5. The programmer executes the specified test cases and records the actual results.
6. The programmer fills out a problem report form for each test case whose actual results did not agree with the expected results. A reference to the test case number from the log is cited.
7. The programmer makes any necessary changes to the code after checking it out of the CMS library. The corrected code is checked back into CMS by the programmer with a comment as to why the change was made, and then the programmer completes the problem report form.
8. If there was a problem with the test case, the programmer corrects the test case, documenting the correction in the *Notes* section of the log. A GCS Problem Report Form does not need to be filled out by the programmer for a test case change.
9. If a problem is discovered which originated in the design, the programmer must fill out a problem report form for the design and get the approval of the SQA representative to make the change to the design and then complete the problem report form. The programmer can then fill out the necessary problem report form for the code and follow step 7 for fixing the code.
10. The programmer must re-execute all test cases for a module if any changes were made to the code of that module. If any actual results do not agree with the expected results, the cycle

of recording and fixing problems is repeated. Problems which were previously identified and not correctly fixed do not need a new problem report form but additional information about the test case run should be added to the existing problem report form.

11. The Test Completion Review is held.

- Tools
  - GCS Problem Report Form
  - DEC Code Management System (CMS)
  - GCS Module Test Log

5.4 Sub-Frame Testing

Sub-frame testing is the lowest level of testing which the testers will perform. Since the programmers are free to divide up the processing in a sub-frame into any division of modules, the sub-frame is the lowest common denominator between implementations. Both white-box and black-box testing will be performed on the sub-frame level by the testers. The Test Readiness Review before this phase will be held after the tester has devised the white-box test cases. The black-box test cases will already be designed. The white-box testing will be performed first. After white-box testing is completed, the black-box testing will be performed on the same version which the white-box testing was started on; i.e., without the corrections which were made during white-box testing.[18] The Test Completion Review after this phase will result in one version of code. The fixes for all faults discovered by both testing methods will be combined and put into the code. See the *SQA Plan for GCS* for more detail about how this will be done. Figure 6 shows how the black-box and white-box sub-frame testing relate to each other. As the illustration shows, code version[19] 3.0 is the input for both white-box and black-box testing. After those sets of testing are complete, the versions are combined to create version 4.0 of the code. Both sets of test cases (white-box and black-box) will be re-run on the resulting version of code as a regression test to ensure that all problems were fixed and no new ones were introduced.

[18]This procedure will allow the effectiveness of white-box and black-box testing to be compared. It also requires that the programmer not make additional changes when fixing faults in the code.

[19]See the *GCS Plan for Configuration Management* for an explanation of version numbers.

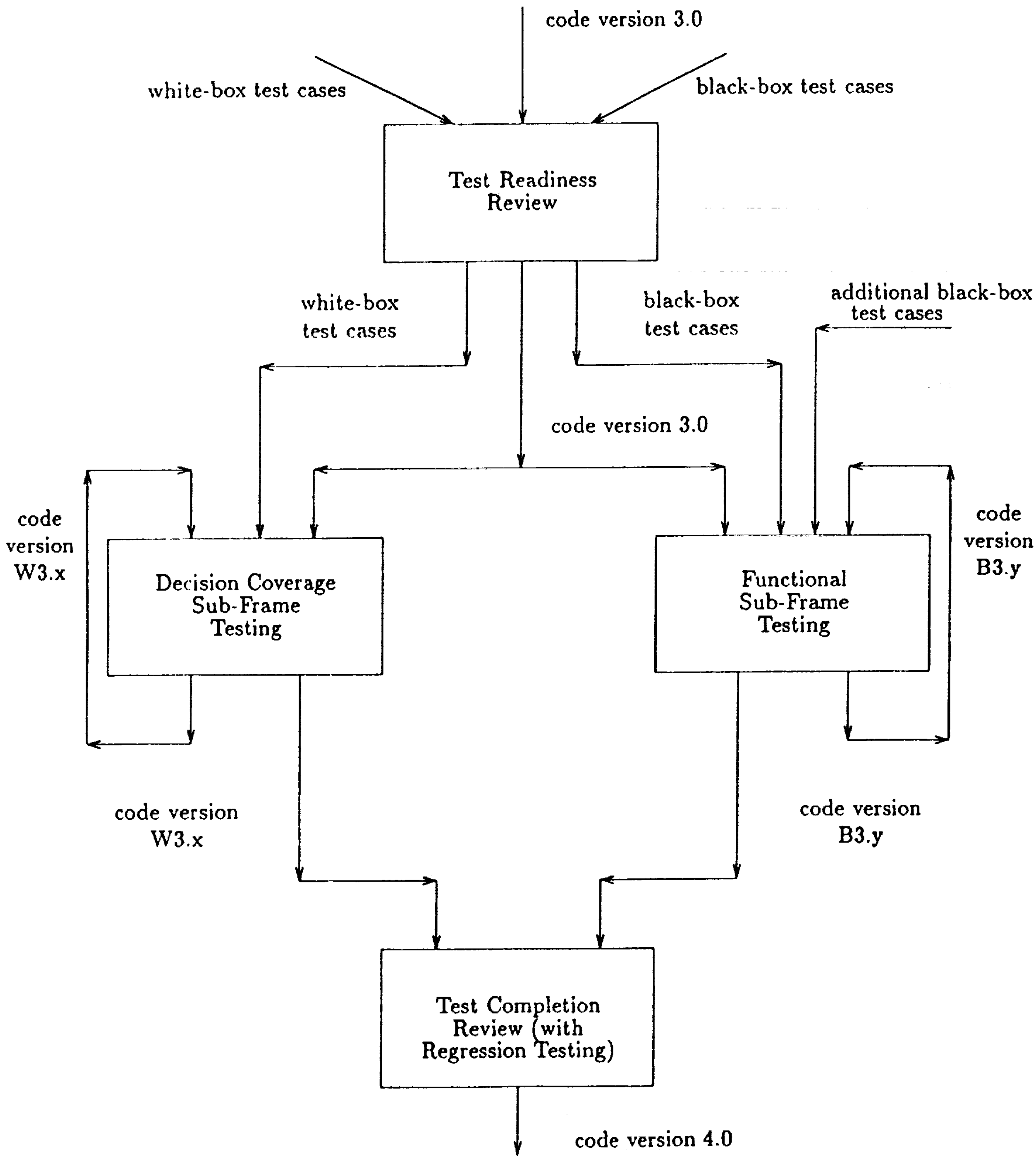

Figure 6: Relationship between White-Box and Black-Box Sub-Frame Testing

5.4.1 White-Box Testing

5.4.1.1 Overview The white-box test cases will be designed using McCabe's method of decision analysis. Please see Appendix C for a discussion of the method and the procedures which will be used to implement it on this project. The tester may structure the test case inputs to execute the test for the entire sub-frame or may structure the test cases to execute each module individually.[20] If changes to the code during white-box testing cause the structure to be altered, new white-box test cases will be constructed. The test cases will be incorporated into a later release of this document to comply with the DO-178A guidelines. Figure 7 shows the white-box sub-frame testing procedure.

[20]It must be remembered that the sub-frame is the lowest common denominator between the implementations, so each tester is testing the same sub-frame but the individual modules may be different. With a coverage technique, it makes sense for the tester to be allowed to test each module individually.

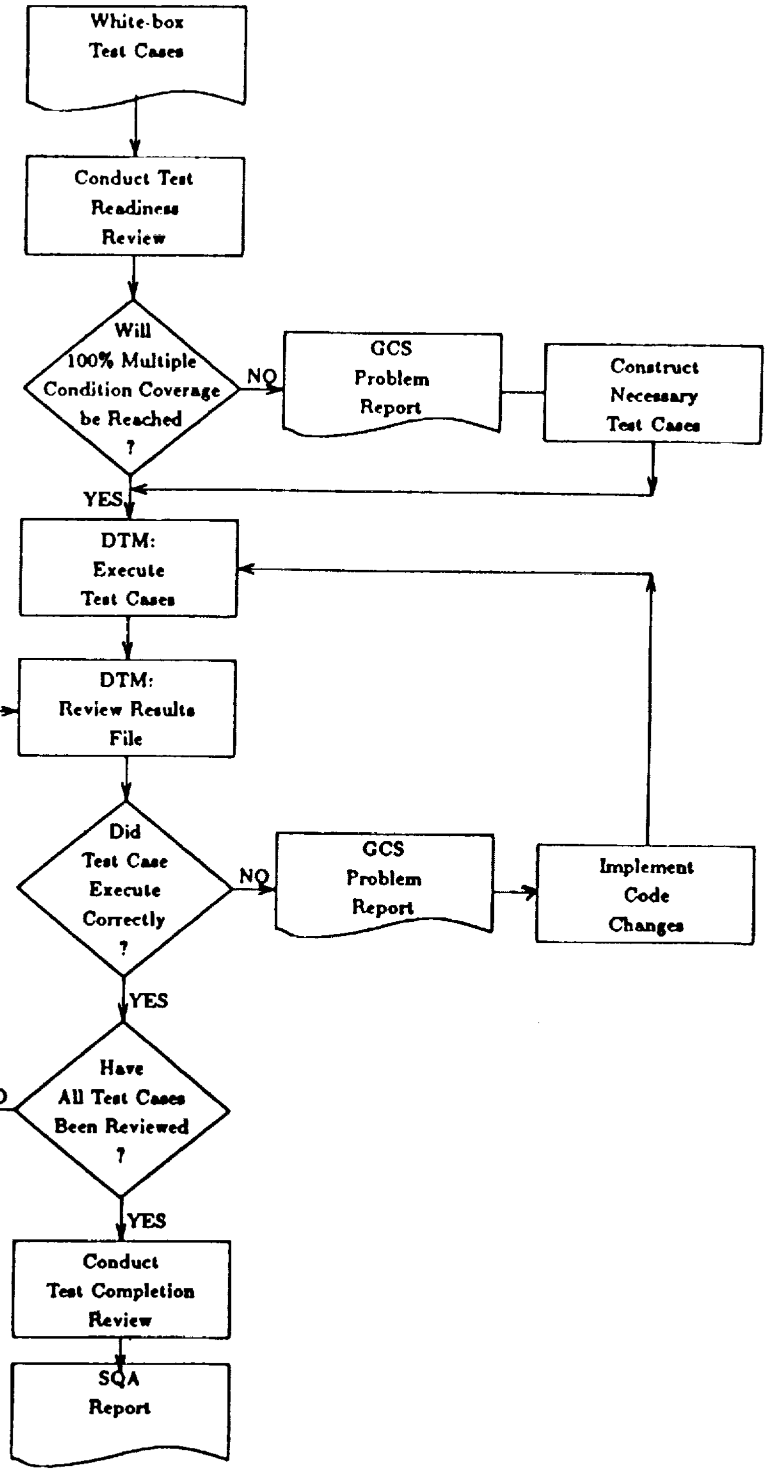

Figure 7: White-Box Sub-Frame Testing Procedure

5.4.1.2 White-Box Sub-Frame Testing Description

- Purpose
  - Test the sub-frame code from a structural perspective in order to check that every statement executes correctly.
- Method
  - White-Box: Multiple-Condition Coverage with McCabe's Method to determine test paths
- Test Creation Stopping Rule
  - Enough test cases have been created to achieve one hundred percent multiple-condition coverage.
- Test Execution Stopping Rule
  - All test cases have been correctly executed.
- Roles
  - Programmer: Fixes any problems found after a complete test run.
  - Tester: Designs and executes test cases.
  - User/Analyst: Assists tester in designing test cases by answering questions about the *GCS Development Specification* and simulator. Makes any necessary modifications to the *GCS Development Specification*.
  - SQA Representative: Approves any changes which need to be made to the programmer's design or code or the test cases and conducts the Test Readiness and test Completion Reviews.
- Input
  1. *GCS Development Specification* including Modifications
  2. GCS Design – includes Programmer's *GCS Design Description* and team*work* Design Diagrams

3. GCS Module Code under Configuration Control
4. Programmer Instructions #3 – Coding Standards for GCS Applications (See *GCS Programmer's Manual.*)

- Output
  1. Completed GCS Problem Report Forms (See Appendix B page 101.)
  2. GCS Sub-Frame Code under Configuration Control
  3. Coverage Analyzer Output (See Section refcovertool.)
  4. White-Box Test Cases and Results under Configuration Control in DTM

- Test Readiness Review checks that
  - All test cases are fully documented including all inputs and expected results.
  - The test cases appear to achieve 100% multiple-condition coverage.

- Test Completion Review checks that
  - The test results are documented.
  - All discrepancies between actual and expected test results are documented in a problem report.
  - All problems are corrected and are appropriately marked in the code as specified in Programmer Instruction #3 – Coding Standards for GCS Applications. (See *GCS Programmer's Manual.*)
  - All test case errors are corrected and documented in a problem report.
  - All new test cases, necessitated by changes to the structure of the code, are fully documented including all inputs and outputs.
  - At least 100% statement coverage was achieved.
  - All problem report forms are completed and approved by SQA.

- General Procedures, repeated for each sub-frame

1. The tester uses the programmer's code to determine the basis path for each module using the McCabe methodology. The remaining paths are also determined. See Appendix C for the procedures to find the paths.
2. The paths for the modules are combined to create sub-frame paths at the tester's discretion.
3. The tester uses the *GCS Development Specification* to determine the input and expected output for each sub-frame test path.
4. A DTM **benchmark** file is created for each white-box test case.
5. The Test Readiness Review[21] is held.
6. The tester executes all test cases in DTM using the test case driver which interacts with GCS_SIM and the coverage tool. The test case **results** files are compared to the existing **benchmark** files for that test case. The test case driver only acts for the sub-frame; if the tester wishes to stay on the module level, she may need to do additional development on the driver.
7. A problem report is filled out for each test case whose actual results did not agree with the expected results.
8. The programmer fixes all problems which were discovered and completes the accompanying GCS Problem Report Form. If a problem is traced back to the design, an additional GCS Problem Report Form must be filled out for the design.
9. Changes which are not related to observed failures cannot be made during sub-frame testing.
10. After all fixes are made, the programmer submits the new code for configuration control. The SQA representative determines if the code fixes are acceptable and re-enters the code into configuration control.
11. At the discretion of the SQA representative, significantly modified code should go through another Code Review as described

[21] Actually one Test Readiness Review will be held for all sub-frame testing. All white-box sub-frame test cases will be reviewed as well as all black-box sub-frame test cases.

in Section 5.2. If more than 20 lines of code have been added or modified, due to a single change, the code must be re-reviewed.

12. If the structure of the code has changed, new test cases must be constructed.
13. When the tester gets the new copy of code, all test cases are re-executed in DTM. If any actual results do not agree with the expected results, the cycle of reporting and fixing problems is repeated. Problems which were previously identified and not fixed do not need a new problem report form, but additional information about the test case run will be added to the existing problem report form.
14. The Test Completion Review[22] is held.

- Tools
  - GCS Problem Report Form (See Appendix B page 101.)
  - DEC Code Management System (CMS)
  - McCabe Analysis of Complexity Tool (ACT)
  - Coverage Information Tool (See Section 4.7.)
  - DEC/Test Manager (DTM)
  - GCS_SIM
  - Sub-Frame Test Case Driver

[22] Actually only one Test Completion Review is held for all sub-frame testing. Both white-box and black-box testing results will be reviewed

5.4.2 Black-Box Testing

5.4.2.1 Overview The sub-frame test cases based on the requirements will be designed by the three testers as a team.[23] Every requirement will be covered by at least one test case. The test case will include the input and expected results. The tests will be executed on each implementation separately by the appropriate tester. Each implementation must correctly execute all test cases. Coverage information will be gathered on each test case. A version of code will be configured after all test cases run correctly. If more than six serious or critical errors are found during the testing of a single sub-frame, the tester has the option to write additional test cases and execute them. The constraint is placed on the tester that the amount of effort may not cause the schedule to slip. A copy of the schedule can be found in the *GCS Plan for Software Aspects of Certification.* The test cases will be incorporated into a later release of this document to comply with the DO-178A guidelines. Figure 8 shows the black-box sub-frame testing procedure.

[23]The system and frame test cases will have been designed first. It is easier for the testers to start writing test cases at the highest level and then use existing prototypes of GCS to help determine intermediate values (which are frame outputs) that would make sense for sub-frame test cases.

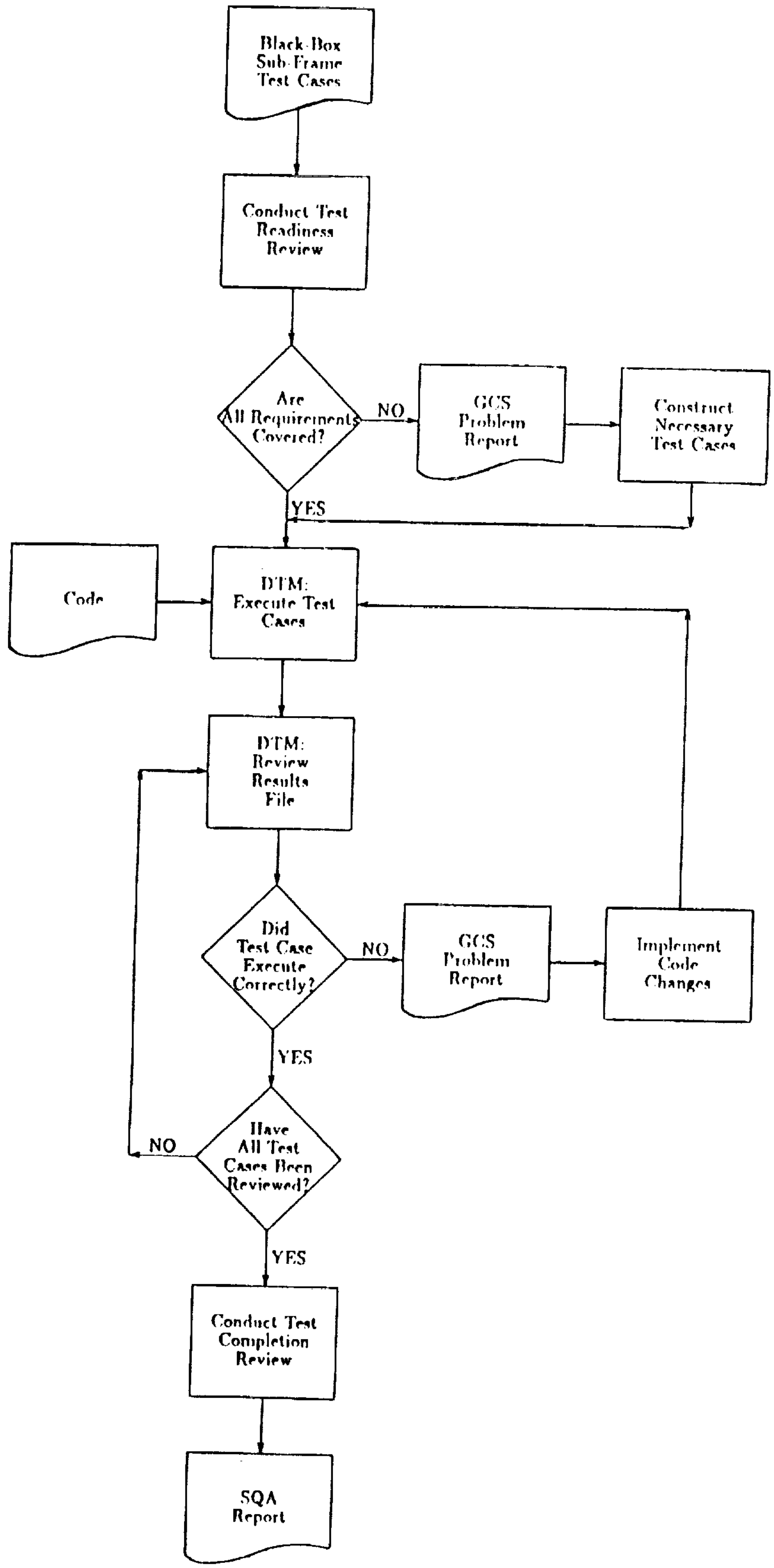

Figure 8: Black-Box Sub-Frame Testing Procedure

5.4.2.2 Black-Box Sub-Frame Testing Description

- Purpose
  - Check that all functions of the sub-frame execute correctly.
- Method
  - Black-Box: Boundary-Value Analysis and Error Guessing
- Test Creation Stopping Rule
  - Enough test cases have been created to achieve one hundred percent requirements coverage.
- Test Execution Stopping Rule
  - All test cases execute correctly.
  - Additional testing does not cause a schedule impact.
- Roles
  - Programmer: Fixes any problems found after a complete test run.
  - Tester: Designs and executes test cases.
  - User/Analyst: Assists tester in designing test cases by answering questions about the *GCS Development Specification* and simulator. Makes any necessary modifications to the *GCS Development Specification.*
  - SQA Representative: Approves any changes which need to be made to the programmer's design or code or the test cases and conducts the Test Readiness and Completion Reviews.
- Input
  1. *GCS Development Specification* including Modifications
  2. GCS Design – Programmer's *GCS Design Description* and team*work* Design Diagrams

3. GCS Module Code under Configuration Control
4. Black-Box Sub-Frame Test Cases[24]
5. Programmer Instruction #3 - Coding Standards for GCS Applications (See *GCS Programmer's Manual.*)

- Output
  1. Completed GCS Problem Report Forms (See Appendix B page 101.)
  2. GCS Sub-Frame Code under Configuration Control
  3. Black-Box Sub-Frame Test Cases and Results under Configuration Control in DTM

- Test Readiness Review checks that
  - All Requirements are covered by one or more test cases.
  - The test cases are fully documented including all inputs and expected results.

- Test Completion Review checks that
  - The test results are documented.
  - All discrepancies between actual and expected test results are documented in a problem report.
  - All problems are corrected and are appropriately marked in the code as specified in Programmer Instructions #3 - Coding Standards for GCS Applications. (See *GCS Programmer's Manual.*)
  - All test case errors are corrected and documented in a problem report.
  - All problem report forms are completed and approved by SQA.

- General Procedures, repeated for each sub-frame
  1. The Test Readiness Review[25] is held.

[24] The test cases are considered input because they were designed before this phase began.

[25] Actually one Test Readiness Review will be held for all sub-frame testing. All black-box sub-frame test cases will be reviewed as well as all white-box sub-frame test cases.

2. The tester executes all test cases in DTM using the test case driver which interacts with GCS_SIM and the coverage information tool. The test case **results** files are compared to the existing **benchmark** files for that test case.
3. A problem report is filled out for each test case whose actual results did not agree with the expected results.
4. The programmer fixes all problems which were discovered and completes the accompanying GCS Problem Report Form. If a problem is traced back to the design, an additional GCS Problem Report Form must be filled out for the design.
5. Changes which are not related to observed failures cannot be made during sub-frame testing.
6. After all fixes are made, the programmer submits the new copy of code for configuration control. The SQA representative must approve the fixes before the code is actually configured.
7. At the discretion of the SQA representative, significantly modified code should go through another Code Review as described in Section 5.2. If more than 20 lines of code have been added or modified, due to a single change, the code must be re-reviewed.
8. When the tester gets the new copy of code, all test cases are re-executed in DTM. If any actual results do not agree with the expected results, the cycle of reporting and fixing problems is repeated. Problems which were previously identified and not fixed do not need a new problem report form, but additional information about the test case run will be added to the existing problem report form.
9. If more than six serious or critical problems are discovered, the tester may optionally design additional cases. The same procedure as described above will be followed for executing the additional test cases, reporting any problems, and re-executing test cases after problems are fixed.
10. The Test Completion Review[26] is held.

[26]Actually only one Test Completion Review is held for all sub-frame testing. Both black-box and white-box testing results will be reviewed

- Tools
  - GCS Requirements Traceability Matrix
  - GCS Problem Report Form
  - DEC Code Management System (CMS)
  - Coverage Information Tool (See Section 4.7.)
  - DEC/Test Manager (DTM)
  - GCS_SIM
  - Sub-Frame Test Driver

5.4.3 Regression Testing

Regression testing is performed on the one version of code for each sub-frame which results from the Test Completion Review after sub-frame testing as shown in Figure 6. All white-box[27] and black-box sub-frame test cases will be used. The regression testing is not performed in a formal manner like the other testing, but it is necessary to ensure that no errors have crept into the system. The SQA report for sub-frame testing will include an error log. (See *SQA Plan for GCS.*) The following procedures will be followed during regression testing.

1. The tester will use DTM to execute all of the white-box and black-box sub-frame test cases on the new versions of sub-frame code.

2. If the actual results of the test case do not agree with the expected results, the programmer and SQA representative are called in to fix the code.

3. Although a problem report form does not have to be filled out, the fix to the code should still be documented as described in Programmer Instruction #3 - Coding Standards for GCS Applications. (See *GCS Programmer's Manual.*) The problem will also be noted by the SQA representative in his sub-frame testing report. [28]

4. When a fix is made, all test cases for that sub-frame must be re-run.

5. When all test cases for all three sub-frames have executed correctly, the SQA representative will report that sub-frame testing is finished.

---

[27] 100% multiple-condition coverage may not be assured if the structure of the code has changed as a result of the combination of the two versions.

[28] Problem report forms do not have to be filled out because the error was caused by the Test Completion Review team not the individual programmer. These errors will not show up during the repetitive run testing because the version of the code that regression testing is performed on will never be put in the repetitive run harness. The problem will still be documented as described in the *SQA Plan for GCS.*

5.5 Frame Testing

5.5.1 Overview

Frame Testing is equivalent to DO-178A Module Integration Testing. Since there are only three sub-frames, they can be integrated simultaneously. The three testers will design the frame test cases based on the requirements as a team. The input for each test case will be taken from expected intermediate results of the system test cases.[29] Every requirement will be covered by at least one test case. The tests will be executed on each implementation separately by the appropriate tester. Each implementation must correctly execute all test cases. The test cases will be incorporated into a later release of this document to comply with the DO-178A guidelines. Figure 9 shows the frame testing procedure.

[29]The system test cases were designed before the frame test cases.

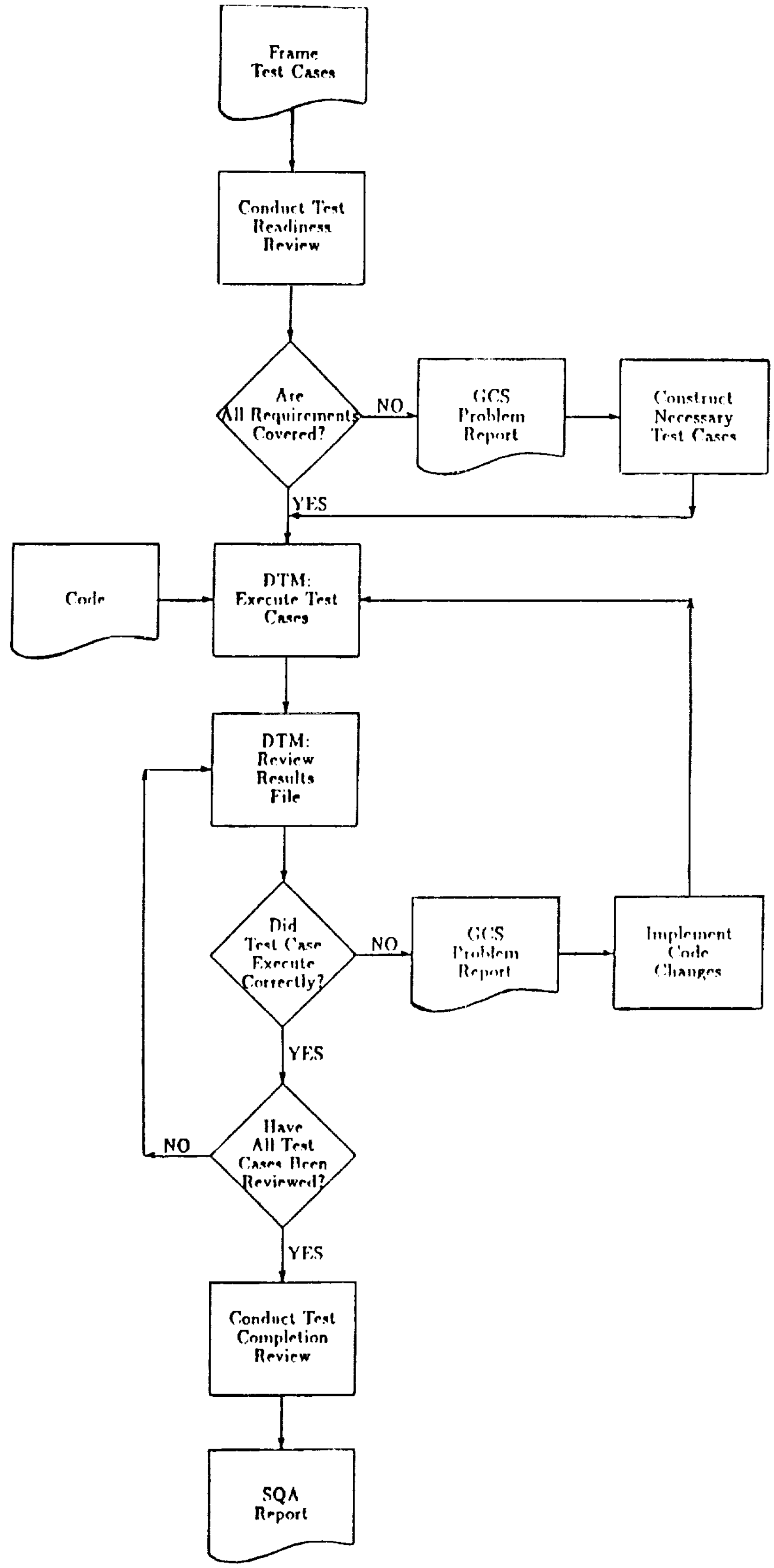

Figure 9: Frame Testing Procedure

5.5.2 Frame Testing Description

- Purpose
  - Check that all sub-frames execute together correctly.
- Method
  - Black-Box: Boundary-Value Analysis and Error Guessing
- Test Creation Stopping Rule
  - Enough test cases have been created to achieve one hundred percent requirements coverage.
- Test Execution Stopping Rule
  - All test cases executed correctly.
- Roles
  - Programmer: Fixes any problems found after a complete test run.
  - Tester: Designs and executes test cases.
  - User/Analyst: Assists tester in designing test cases by answering questions about the *GCS Development Specification* and simulator. Makes any necessary modifications to the *GCS Development Specification*.
  - SQA Representative: Approves any changes which need to be made to the programmer's design or code or the test cases and conducts the Test Readiness and Test Completion Reviews.
- Input
  1. *GCS Development Specification* including Modifications
  2. GCS Design – includes Programmer's *GCS Design Description* and team*work* Design Diagrams
  3. GCS Sub-Frame Code under Configuration Control

4. Black-Box Frame Test Cases[30]

- Output
  1. Completed GCS Problem Report Forms (See Appendix B page 101.)
  2. GCS Integrated Code under Configuration Control
  3. Black-Box Frame Test Cases and Results under Configuration Control in DTM
- Test Readiness Review checks that
  - All requirements are covered by one or more test cases.
  - The test cases are fully documented including all inputs and expected results.
- Test Completion Review checks that
  - The test results are documented.
  - All discrepancies between actual and expected test results are documented in a problem report.
  - All problems are corrected and are appropriately marked in the code as specified in Programmer Instruction #3 – Coding Standards for GCS Applications. (See *GCS Programmer's Manual.*)
  - All test case errors are corrected and documented in a problem report.
  - All problem reports forms are completed and approved by SQA.
- General Procedures
  1. The Test Readiness Review is held.
  2. The tester executes all test cases in DTM using the frame-level test driver which interacts with the simulator and the coverage information tool. The test case **results** files are compared to the existing **benchmark** files for that test case.

[30]The test cases are considered input because they were designed before this phase began.

3. A problem report is filled out for each test case whose actual results did not agree with the expected results.
4. The programmer fixes all problems which were discovered and completes the accompanying GCS Problem Report Form. If a problem is traced back to the design, an additional GCS Problem Report Form must be filled out for the design.
5. After all fixes are made, the programmer must get approval of the fixes before submitting the new copy of code to be configured.
6. When the tester gets the new copy of code, all test cases are re-executed in DTM. If any actual results do not agree with the expected results, the cycle of reporting and fixing problems is repeated. Problems which were previously identified and not fixed do not need a new problem report form but additional information about the test case run will be added to the existing problem report form.
7. The Test Completion Review is held.

- Tools
  - GCS Requirements Traceability Matrix
  - GCS Problem Report Form
  - DEC Code Management System (CMS)
  - Coverage Information Tool (See Section 4.7.)
  - DEC/Test Manager (DTM)
  - GCS_SIM
  - Frame Test Driver

5.6 System Testing

5.6.1 Overview

System testing will consist of executing entire trajectories in GCS_SIM. Test case input will consist of run parameters. There will be 100 test cases which will be divided equally between initial conditions which are considered stress conditions and random conditions, some of which may be stress conditions. Some of the stress conditions are likely to correctly cause the vehicle to crash. The stress test cases will be carefully chosen to exercise all parts of the software. For the random test cases, initial conditions will be distributed randomly across the input space. Some intermediate results will be checked, including where the velocity altitude contour is crossed. Each test case description will specify which intermediate results are examined. An emphasis will be placed on checking timing requirements. The test cases will be designed by the three testers together.[31] The tests will be executed on each implementation separately by the appropriate tester. The implementation must correctly execute all 50 stress cases, and 50 consecutive random test cases. The stress test cases will be executed first. If an error is found by a stress test case, the problem is corrected immediately, and all previous stress test cases are re-executed in DTM. If an error is found by a random test case, the problem is corrected immediately, and that test case is re-executed in DTM. The count of test cases (towards 50) is restarted on new random test cases. The new random test cases will be chosen by the three testers as a team. A final review will be performed after this phase of testing. The test cases will be incorporated into a later release of this document to comply with the DO-178A guidelines. Figure 10 shows the system testing procedure.

[31]The system test cases will be designed first. Then the frame and sub-frame test cases will be constructed from them.

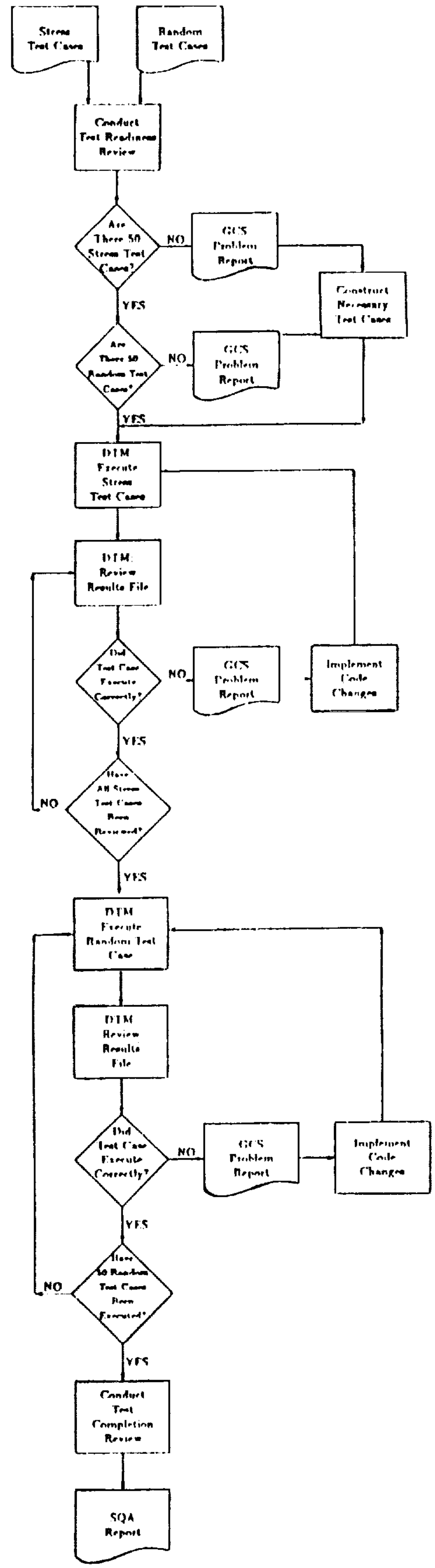

Figure 10: System Testing Procedure

5.6.2 System Testing Description

System Testing Description

- Purpose
  - Check that the code correctly executes trajectories and that a minimal mean test case to failure criteria is achieved.
- Method
  - Black-Box: Stress and Random within Input Space
- Test Creation Stopping Rule
  - Fifty stress test cases have been created.
  - At least fifty random test cases have been created.
- Test Execution Stopping Rule
  - Fifty stress test cases have been executed correctly.
  - Fifty consecutive random test cases have been executed correctly.
- Roles
  - Programmer: Fixes any problems found during test run.
  - Tester: Designs and executes test cases.
  - User/Analyst: Assists tester in designing test cases by answering questions about the *GCS Development Specification* and simulator. Makes any necessary modifications to the *GCS Development Specification*.
  - SQA Representative: Approves any changes which need to be made to the programmer's design or code or the test cases and conducts the Test Readiness and Test Completion Reviews.
- Input
  1. *GCS Development Specification* including Modifications

2. GCS Design – includes Programmer's *GCS Design Description* and team*work* Design Diagrams
3. GCS Integrated Code under Configuration Control
4. Black-Box System Test Cases[32]: 50 stress and 50 random
5. Programmer Instructions #3 – Coding Standards for GCS Applications (See *GCS Programmer's Manual.*)

- Output

  1. Completed GCS Problem Report Forms (See Appendix B page 101.)
  2. GCS System Code under Configuration Control
  3. System Test Cases and Results under Configuration Control in DTM

- Test Readiness Review checks that

  – There are 50 stress test cases and 50 random test cases.

  – The test cases are fully documented including all inputs and expected results.

- Test Completion Review checks that

  – The test results are documented.

  – All discrepancies between actual and expected test results are documented in a problem report.

  – All problems are corrected and are appropriately marked in the code as specified in Programmer Instruction #3 - Coding Standards for GCS Applications (See *GCS Programmer's Manual.*)

  – All test case errors are corrected and documented in a problem report.

  – All problem reports forms are completed and approved by SQA.

  – All additional necessary random test cases are fully documented including all inputs and expected results.

[32]The test cases are considered input because they were designed before this phase began.

- Fifty random test cases did execute without error.

- General Procedures

  1. The Test Readiness Review is held.
  2. The tester executes each stress test case using DTM in GCS_SIM with the coverage information tool. The test case **results** files are compared to the existing **benchmark** files for that test case.
  3. A problem report is filled out for each stress test case whose actual results did not agree with the expected results.
  4. The programmer fixes any errors immediately and completes the accompanying GCS Problem Report Form. If a problem is traced back to the design, an additional GCS Problem Report Form must be filled out for the design.
  5. After the fix is made, the programmer submits the new code for configuration control. The SQA representative must approve the change before the code is actually configured.
  6. At the discretion of the SQA representative, significantly modified code should go through another Code Review as described in Section 5.2. If more than 20 lines of code have been added or modified, due to a single change, the code must be re-reviewed.
  7. When the tester gets the new copy of code, the test case which detected the error is re-executed in DTM. All previously executed stress test cases are also re-executed in DTM. If the actual results still do not agree with the expected results, the cycle of reporting and fixing the problem is repeated. Problems which were previously identified and not fixed do not need a new problem report form, but additional information about the test case run will be added to the existing problem report form.
  8. When all 50 stress test cases are correctly executed, the tester executes each random test case using DTM in GCS_SIM with the coverage information tool. The test case **results** files are compared to the existing **benchmark** files for that test case.
  9. A problem report is filled out for each random test case whose actual results did not agree with the expected results.

10. The programmer fixes any errors immediately and completes the accompanying GCS Problem Report Form. If a problem is traced back to the design, an additional GCS Problem Report Form must be filled out for the design.

11. After the fix is made, the programmer submits the new code for configuration control. The SQA representative must approve the change before the code is actually configured.

12. At the discretion of the SQA representative, significantly modified code should go through another Code Review as described in Section 5.2 . If more than 20 lines of code have been added or modified, due to a single change, the code must be re-reviewed.

13. When the tester gets the new copy of code, the test case which detected the error is re-executed in DTM. If the actual results still do not agree with the expected results, the cycle of reporting and fixing the problem is repeated. Problems which were previously identified and not fixed do not need a new problem report form, but additional information about the test case run will be added to the existing problem report form.

14. New test cases are then executed with the count towards 50 random test cases restarted.

15. The Test Completion Review is held.

- Tools
  - GCS Requirements Traceability Matrix
  - GCS Problem Report Form
  - DEC Code Management System (CMS)
  - Coverage Information Tool (See Section 4.7.)
  - DEC/Test Manager (DTM)
  - GCS_SIM

6 Summary

The extensive verification plan documentation required by the DO-178A guidelines helps ensure that the verification procedures are fully specified. An effort has been made to make the review and testing procedures as similar to those found in industry as possible, within the constraints of the experimental environment. A description of the constraints imposed by the experimental environment are discussed in the *GCS Plan for Software Aspects of Certification.* The constraints of financial resources and schedule resources are present on all projects, whether in industry or research. Test methodologies were selected with regards to efficiency and thoroughness. Since any verification procedure is only as good as the competence of those who administer it, the verification team for GCS was carefully chosen and consists of people who have experience in testing software for industry, government, and research. These factors contribute to a realistic software testing environment and enhance the integrity of the GCS project.

A Equivalence Partitioning and Boundary-Value Analysis Example for GCS

A.1 Introduction

This example will use the specification description for Altimeter Radar Sensor Processing (ARSP) from the *GCS Development Specification* version 2.0. The input to ARSP and their corresponding ranges from the Data Element Descriptions

input	data type	range
AR_ALTITUDE	array (0...4) of real*8	[0,2000]
AR_COUNTER	integer*2	$[0, 2^{10} - 1]$
AR_FREQUENCY	real*8	$[1, 10^9]$
AR_STATUS	array (0...4) of logical*1	[healthy or failed]

The inputs AR_ALTITUDE and AR_STATUS are five element arrays. The tables which show their equivalence classes and boundary values only have one entry for the array. When the actual test cases are identified, values for all elements of the array must be specified.

A.2 Input Equivalence Classes

After the inputs are identified, the equivalence classes for each input are identified. The equivalence classes for ARSP are shown below.

input	equivalence classes
AR_ALTITUDE	$0 \leq AR\_ALTITUDE \leq 2000$ $AR\_ALTITUDE < 0$ $AR\_ALTITUDE > 2000$
AR_COUNTER	$0 \leq AR\_COUNTER \leq 2^{10} - 1$ $AR\_COUNTER < 0$ $AR\_COUNTER > 2^{10} - 1$
AR_FREQUENCY	$1 \leq AR\_FREQUENCY \leq 10^9$ $AR\_FREQUENCY < 1$ $AR\_FREQUENCY > 10^9$
AR_STATUS	healthy failed

Notice that for the ranges where the input is expected to be between two values, there is one class for inside the range, one class for below the range, and one class for above the range. For the inputs that can be one of a set of discrete values, there is an equivalence class for each set of values which is treated differently.

A.3 Boundaries for Input

The boundary-value analysis technique uses the boundary values of the equivalence classes to pick input. The table below shows the the input and equivalence classes again. The last column of the table shows the actual value which would be picked when making a test case.

input	equivalence class	chosen input value
AR_ALTITUDE	$0 \leq AR\_ALTITUDE \leq 2000$	(a) 0*
		(b) 2000
	$AR\_ALTITUDE < 0$	(c) -0.000010†
	$AR\_ALTITUDE > 2000$	(d) 2000.000010
AR_COUNTER	$0 \leq AR\_COUNTER \leq 2^{10} - 1$	(e) 0
		(f) $2^{10} - 1$
	$AR\_COUNTER < 0$	(g) -1
	$AR\_COUNTER > 2^{10} - 1$	(h) 2^{10}
AR_FREQUENCY	$1 \leq AR\_FREQUENCY \leq 10^9$	(i) 1
		(j) 10^9
	$AR\_FREQUENCY < 1$	(k) 0‡
	$AR\_FREQUENCY > 10^9$	(l) $10^9 + 1$
AR_STATUS	healthy	(m) healthy
	failed	(n) failed

A.4 Pseudo-Boundary Conditions

A set of boundary conditions known as pseudo-boundary conditions can also be identified. They require a knowledge of the specification of the system. Pseudo-boundary conditions are not included in the regular range but represent a boundary which causes a different output. Setting AR_FREQUENCY to 0 would be a pseudo-boundary condition because AR_FREQUENCY is a divisor and normally dividing by 0 would cause an error. A high value for AR_ALTITUDE when AR_FREQUENCY is high may cause the new AR_ALTITUDE to be calculated incorrectly because AR_COUNTER is limited to 10 bits. Another pseudo-boundary condition is that AR_COUNTER is set to all ones ($2^{15} - 1$ = -1) when no echo is received. The pseudo-boundary conditions for ARSP are summarized in the table below.

*All values which do not show decimal places are assumed to be exact.

†Values which show decimal places are calculated with an accuracy of 1 significant digit less than is required in the specification.

‡No accuracy was specified for AR_FREQUENCY because it is a run parameter. Integer boundary values will be used.

input	equivalence class	chosen input value
AR_FREQUENCY	AR_FREQUENCY = 0	(o)0
AR_FREQUENCY	$0 \leq AR\_FREQUENCY < 7.5 * 10^7$ $7.5 * 10^7 \leq AR\_FREQUENCY \leq 10^9$	 (p)$7.5 * 10^7$
AR_COUNTER	AR_COUNTER = -1	(q)-1

A.5 Equivalence Classes for Output

The boundary-value analysis technique also calls for the output space to be examined. The output for ARSP is listed in the following table.

output	data type	range
AR_ALTITUDE	array (0...4) of real*8	[0,2000]
AR_STATUS	array (0...4) of logical*1	[healthy, failed]
K_ALT	array (0...4) of integer*4	[0,1]

For the output we want to achieve a value in each equivalence class as shown in the table below.

output	equivalence classes
AR_ALTITUDE	$0 \leq AR\_ALTITUDE \leq 2000$ $AR\_ALTITUDE < 0$ $AR\_ALTITUDE > 2000$
AR_STATUS	healthy failed
K_ALT	0 1

Next we identify the combinations of input required to achieve those output values. This requires looking at the specification. We have to work backwards from desired outputs to desired inputs. If values for some of the input are not specified, that means they are "don't cares."

desired output	required input values
$0 \leq AR\_ALTITUDE \leq 2000$	$(r) 0 \leq AR\_COUNTER \leq 1000$
$AR\_ALTITUDE < 0$	(s)$AR\_FREQUENCY < 0$
$AR\_ALTITUDE > 2000$	(t)$AR\_COUNTER > 1000$
AR_STATUS = healthy	(u) $0 \leq AR\_COUNTER \leq 2^{10} - 1$
AR_STATUS = failed	(v)AR_COUNTER = -1
K_ALT = 0	(w)AR_COUNTER = -1 AR_STATUS[n] = failed
K_ALT = 1	(x)AR_COUNTER = -1 AR_STATUS[0...4] = healthy
K_ALT = 1	(y)$0 \leq AR\_COUNTER \leq 2^{10} - 1$

A.6 Test Cases

Finally we identify the actual test cases by combining as many equivalence classes as possible until all of them are used. All equivalence classes from the input, pseudo-boundary conditions, and all of the required inputs for the output equivalence classes are used, as shown in the second column.

test case	inputs used	AR_ALTITUDE	AR_COUNTER	AR_FREQUENCY	AR_STATUS
1	a,e,i,m	$[0 \ldots 4] = 0$	0	0	$[0 \ldots 4]$ = healthy
2	b,f,j,n,y	$[0 \ldots 4] = 2000$	$2^{10} - 1$	10^9	[0] = failed $[1 \ldots 4]$ = healthy
3	c,g,k,q,o,v	[0] = -0.00001 $[1 \ldots 4] = 0$	-1	-1	$[0 \ldots 4]$ = healthy
4	d,h,l,t,u	[0] = 2000.00001 $[1 \ldots 4] = 2000$	2^{10}	$10^9 + 1$	[1] = failed [0,2,3,4] = healthy
5	p	$[0 \ldots 4] = 2000$	$0 \leq \ \leq 2^{10} - 1$	$7.5 * 10^7$	$[0 \ldots 4]$
6	r	$0 \leq [0 \ldots 4] \leq 2000$	$0 \leq \ \leq 1000$	$0 \leq \ \leq 10^9$	$[0 \ldots 4]$ = healthy
7	s	$0 \leq [0 \ldots 4] \leq 2000$	$0 \leq \ \leq 2^{10} - 1$	-1	$[0 \ldots 4]$ = healthy
8	w,x	$0 \leq [0 \ldots 4] \leq 2000$	-1	$0 \leq \ \leq 10^9$	[2] = failed [0,1,3,4] = healthy

B GCS Forms

B.1 GCS Requirements Traceability Matrix

GCS REQUIREMENTS TRACEABILITY MATRIX

Process	Requirement	Design	Code	Sub-Frame Test	Frame Test	System Test
ARSP	Use correct data elements					
	Read sensor					
	Determine altitude					
	Set altimeter radar status to healthy					
ASP	Use correct data elements					
	Read accelerometer					
	Remove characteristic bias					
	Correct for misalignment					
	Determine accelerations and accelerometer status					
CP	Use correct data elements					
	Construct packet					
	Send packet					
	Set communicator status to healthy					
GSP	Use correct data elements					
	Rotate array values					
	Adjust Gain					
	Convert G_COUNTER					
	Set gyroscope status to healthy					
TDLRSP	Use correct data elements					
	Rotate values					
	Determine radar beam states					
	Determine beam velocities					
	Average beam velocities and convert to body velocities					
	Set TDLR_STATUS to healthy					

1 of 3

GCS REQUIREMENTS TRACEABILITY MATRIX

Process	Requirement	Design	Code	Sub-Frame Test	Frame Test	System Test
TDSP	Use correct data elements					
	Determine if touch-down has been sensed					
TSP	Use correct data elements					
	Conversion of solid state temperature					
	Conversion of thermocouple pair temperature					
	Select most accurate estimate					
	Set status to healthy					
CP	Use correct data elements					
	Construct packet					
	Send packet					
	Set communicator status to healthy					
GP	Use correct data elements					
	Set up the GP_ROTATION matrix					
	Calculate new values of velocity, altitude, and attitude					
	Determine if engines should be on or off					
	Determine VELOCITY_ERROR					
	Determine if contour has been crossed					
	Determine guidance phase					
AECLP	Use correct data elements					
	Determine if axial engines are switched on					
	Determine engine temperature					
	Compute limiting errors					
	Compute pitch, yaw, and thrust errors					
	Compute axial engine valve settings					
	Set axial engine status to healthy					

2 of 3

GCS REQUIREMENTS TRACEABILITY MATRIX

Process	Requirement	Design	Code	Sub-Frame Test	Frame Test	System Test
CP	Use correct data elements					
	Construct packet					
	Send packet					
	Set communicator status to healthy					
CRCP	Use correct data elements					
	Determine CHUTE_RELEASED value					
RECLP	Use correct data elements					
	Determine if engines are on					
	Determine pulse intestity and direction					
	Determine roll engine command					
	Command engines					
	Set roll engine status to healthy					
GENERAL	Process executed in manner compatible with table 4.1					
	Rendezvous routine called first					
	Rendezvous routine called between all sub-frames					
	Timing requirements in Table 6.1 are met					
	Memory space requirements in Table 6.2 are met					

3 of 3

B.2 GCS Problem Report Form

GCS PROBLEM REPORT FORM

PR No.:	Planet:	Discovery Date & Time:

Activity at Fault/Error Detection Time:

- ☐ Reading Specification Document
- ☐ Reading Code
- ☐ Design Review
- ☐ Code Review
- ☐ Test Readiness/Completion Review

Problem Discovered By:

- ☐ Module Testing
- ☐ Sub-frame Testing
- ☐ Frame Testing
- ☐ System Testing
- ☐ Multiple Version Comparisons

- ☐ White Box
- ☐ Black Box

Explanation of Fault/Error Detection: (Cite test case number, input, expected output, and actual output where appropriate)

Tester Approval:

General Version Number:	Effort Hours for Fix:	Fix Date & Time:

Description of Problem and Fix: (First state module affected and CMS Generation Number)

Error Type

- [ ] Computational Error
- [ ] Interface Error
- [ ] Other ______________________
- [ ] Control Flow Error
- [ ] Operation Error
- [ ] Data Error
- [ ] Inconsistency

Error Severity

- [ ] Critical
- [ ] Serious
- [ ] Nonessential
- [ ] Not Applicable

Was this error related to a previous change?

- [ ] Yes (PR No. ____________)
- [ ] No
- [ ] Can't Tell

When did the error enter the system?

- [ ] Specification
- [ ] Design
- [ ] Coding and Testing
- [ ] Other
- [ ] Can't Tell
- [ ] Not Applicable

SQA Approval:

GCS PROBLEM REPORT FORM

Continuation of Section: ______________________

B.3 GCS Design Review Checklist

planet: ______________________
date: ______________________
start time:________finish time: ________

GCS DESIGN REVIEW CHECKLIST

A. DESIGN DOCUMENT

1. Does the format of the Design Document match that described in Programmer Instruction #6? *yes no*
2. Does the top level description in the Design Document conflict with anything in the specification? *no yes*
3. Do any subroutine names start with GCS_? *no yes*
4. Is the chosen method(s) of integration described? *yes no*
5. Were team*work* and the Hatley method used as described in Programmer Instruction #7? *yes no*

B. STRUCTURE

1. Did the team*work* model pass the balancing check? *yes no*
2. Are any processes for separate sub-frames grouped together? *no yes*
3. Are all the processes which are described in the specification traceable to the design? *yes no*
4. Is the purpose of each module stated clearly? *yes no*
5. Are the inputs and outputs for each module stated clearly? Do they have their own flow and use the correct units? *yes no*
6. Are the inputs and outputs for the processes described in the specification traceable to the module inputs and outputs? *yes no*
7. Does each module have high cohesion, i.e. strong functional association? *yes no*
8. Do the modules have low coupling between them, i.e. little dependance on each other? *yes no*
9. Are any processes performed which were not described in the specification? *no yes*
10. Is any team*work* process specification longer than two pages? *no yes*

C. PROCESSING SEQUENCE

1. Is the rendezvous routine called before any processing takes place? *yes no*
2. Is the rendezvous routine called between sub-frames? *yes no*
3. Are the processes performed in a manner compatible with table 4.1 in the specification? *yes no*
4. Are all local variables, which depend on data stores, recalculated after the rendezvous routine is called? *yes no*
5. Are all guidance phases handled as described in table 5.9 of the specification? *yes no*
6. Are the position and velocity of the vehicle relative to the altitude contour checked every frame? *yes no*

D. DETAIL

1. Can each team*work* process specification be easily coded into a module? *yes no*
2. Do the team*work* process specifications use pseudo code which resembles the level of detail for program code? *no yes*
3. Does the pseudo code need or use comments to explain it? *no yes*
4. Are all of the specified local variables necessary and are they only used locally? *yes no*
5. Does it appear that the designer intends for any VMS Run-Time Library Routines to be used? *no yes*
6. Does it appear that the designer intends for any VMS System Services to be used? *no yes*
7. Are all data stores used as specified in the Data Dictionary? *yes no*

E. OTHER

1. Are all algorithms understandable? If a reviewer cannot understand a part of the design without repeated explanation, then the design should be redone to make it more clear. *yes no*
2. Are all assumptions documented? *yes no*

3. Is the decision logic clear and does it handle all conditions properly? *yes no*
4. If error handlers are called for, are they used wisely? *yes no*

approval:______________________________

B.4 GCS Code Review Checklist

planet: ______________________
modules reviewed: ______________
:_________________________
date: _______________________
start time:________finish time: ________

GCS CODE REVIEW CHECKLIST

A. FUNCTIONALITY

1. Do the code modules map to a well-defined section in the design? *yes no*
2. Do the code modules account for all requirements specified in the above well-defined section in the design? *yes no*
3. Does each unit have a single purpose? *yes no*

B. FORMAT

1. Has the programmer used available template files for the modules and headers? *yes no*
2. Are comments explanatory and correctly done? I.e., block code comments, section code comments, and declarations section comments. *yes no*
3. Are all assumptions documented in comments? *yes no*
4. Is the formatting of the code correct? *yes no*
   (a) Are labels in column 1? *yes no*
   (b) Are continuation markers in column 6? *yes no*
   (c) Does code begin in column 8 or 16? *yes no*
   (d) Are all indentations 4 spaces? *yes no*
5. Are there more than one hundred lines (or two pages) of code in a module? If yes, what is the justification for this? *no yes*
6. Is there only one argument per line in an OPEN statement? *yes no*
7. Are there blanks around equal ("=") signs in OPEN statements? *yes no*

8. Is all code in lower case, with references to program constants and external modules in upper case? *yes no*
9. Has the programmer used:
   (a) one file per module? *yes no*
   (b) bundled modules for only one application? If so, are module headers and page breaks used appropriately? *yes no*
10. Does the file name match the module name? *yes no*
11. Is all code written only in FORTRAN? *yes no*
12. Are any VMS Run Time Libraries or System services? *no yes*
13. Are any data hidden? (see restrictions on P. 110 of the specifications) *no yes*

C. DATA USAGE

a. Declarations

1. Is all storage declared? *yes no*
2. Is IMPLICIT NONE the only use of the IMPLICIT statement? *no yes*
3. Do declarations show both type and size? (i.e., integer*4) *yes no*
4. Do declarations match the specifications? *yes no*
5. Are arrays explicitly dimensioned? *yes no*

b. Common/Equivalence

1. Are there local variables which are misspellings of a COMMON element? *no yes*
2. Are the elements in the COMMON in the right sequence? *yes no*
3. Are COMMON blocks used in multiple units defined by a common INCLUDE file? *yes no*
4. Do EQUIVALENCE statements force any unintended shared data storage? *no yes*
5. Is each EQUIVALENCE commented? *yes no*

c. Variables

1. Are the variables initialized in DATA statements, BLOCK DATA, or previously defined by assignments or COMMON usage? *yes no*

2. If variables are initialized in DATA statements, should they instead have been initialized by an assignment statement? *no yes*
3. Are variables used for one purpose? *yes no*
4. Are variables which are GCS globals initialized in the program? *no yes*

D. ERROR HANDLING Error handling is not required, but if implemented, it needs to follow certain standards.

1. If there is no error handling, are IOSTAT and ERR used? *no yes*
2. Are errors only handled in a preventative manner? (Error handlers should not intercept system fatal errors.) *yes no*

E. LINKAGE

1. Do all arguments in a subroutine call or function reference agree in number, order, dimension, data type and passed limits with the dummy arguments in the subroutine or function? *yes no*
2. Does the function return data of the correct type? *yes no*
3. Does the calling module deal with any modifications to global data areas, errors on open I/O units, or completion of I/O to certain units which occur in the subprogram? *yes no*
4. Does a subroutine modify any input parameter? If so, is this fact clearly stated? *yes no*
5. Does a FUNCTION routine have only one output data item and no side effects? *yes no*

F. LOGIC

a. Loops

1. Do the initial, terminal, and increment values appear correct? *yes no*
2. Is the correct condition tested? *yes no*
3. In REAL-valued conditions, does testing include testing for a narrow range around the desired value and does it use .GE. and .LE. rather than .GT. and .LT., especially if the test is used for iteration control? *yes no*

4. For REAL-valued index DO loops, is the loop variable subject to floating point errors which can impact other code? *no yes*
5. Are there changes to the index value within a DO loop? *no yes*
6. Is the index variable only used within the DO loop? *yes no*
7. Are there any loops used for delay or timing purposes? *no yes*
8. Is the use of early returns absolutely necessary? *yes no*
9. Are there branches into the body of a loop or out of the body of the loop (other than to the close of the loop or to the statement following the loop)? *no yes*
10. Do targets exist for all branches? *yes no*
11. For compound conditions in DO WHILE loops, are parentheses used correctly? *yes no*
12. Is loop nesting in the correct order? *yes no*

b. Branches

1. Does the use of unconditional GOTOs follow the standard on p.11? *yes no*
2. Is the correct condition tested? *yes no*
3. In REAL-valued IF statements, does testing for single-ended tests include testing for a narrow range around the desired value and does it use .GE. and .LE. rather than .GT. and .LT., especially if the test is used for iteration control? *yes no*
4. Are the correct variables used in the test? *yes no*
5. For compound IF statements, are the parentheses used to control ordering and precedence coded correctly? *yes no*
6. Is the nesting of IF-THEN-ELSE constructs correct? *yes no*
7. Can the code exercise all branches of IF statements? *yes no*
8. Is there a default branch for all IF statements? *yes no*
9. Do the target branches exist? *yes no*

c. Lexical Rules

1. Do integer comparisons account for truncation? *yes no*
2. Are parentheses used correctly for precedence and ordering? *yes no*

3. Are potential floating-point problems considered in the choice of numerical operations? *yes no*

4. Are array subscript references within the bounds of the array? *yes no*

approval:__________________________

B.5 GCS Module Test Log

GCS MODULE TEST LOG

page ____ of ____

planet: ____________________

sub-frame: ____________________

module: ____________________

module input: ____________________

module output: ____________________

total # of test cases: ____________

Notes:

GCS MODULE TEST LOG

page ____ of ____

planet: ____________________
sub-frame: __________________
module: ____________________

PR #	test case #	module input	expected results	actual results	expected accuracy	actual accuracy	date test executed

C McCabe's Structured Test Technique

C.1 Introduction

McCabe's testing technique will be used in white-box sub-frame testing for GCS. The method satisfies the multiple-decision coverage criteria for white-box testing. The technique is applied at the code level. It can be partially automated and provides consistent and objective results.

The technique relies on McCabe's complexity metric, which is based on the cyclomatic number, V(G), from graph theory. After graphing the code in an appropriate form (control flow graph), the complexity metric is determined from the graph. The metric essentially counts the number of decisions in the graph, giving the minimum number of independent test paths necessary for 100% multiple-decision coverage for the code. McCabe then uses a baseline method to find the test paths.

Each test path consists of a set of decisions from the graph. The baseline method involves choosing a baseline path through the graph for the first test path. Successive test paths are found by deviating from the baseline path in a prescribed fashion. The size of the set of test paths will be equivalent to the complexity metric, and the set will in fact be a basis set of test paths. That is, any path through the graph can be found from a linear combination of the test paths in the basis set. Finally, input sets need to be created to satisfy each test path.

C.2 Procedures

These procedures are a basic listing of the steps involved in using McCabe's method. For a more detailed explanation, see [14].

1. On a printout of the code, mark the branches (if-then, case, and loop statements) in the code.
2. Create a control flow graph from the code.
   (a) Nodes in the graph consist of blocks of sequential logic. Any branches in the code (if-then or loop statements) will be noted either as separate nodes or at the end of a block of sequential logic.

   (b) Arcs are transfers of control.

3. Determine cyclomatic complexity. This measure will be the number of test paths necessary to achieve 100% multiple-decision coverage. Use any of the following three formulas.

   (a) Number of decisions in flow graph + 1

   (b) Edges - nodes + 2

   (c) Number of enclosed regions in flow graph (where an enclosed region is a visual region on a graph which is enclosed by edges)

4. Find the baseline path. Any path through the graph will work, but it is recommended that the baseline path be the longest path through the graph without loops.

5. Write out the baseline path by listing the nodes.

6. Find the next test path.

   (a) Follow the baseline path until the first branch is reached.

   (b) Choose another branch than the one the baseline path took.

   (c) As soon as possible, return to the baseline path.

7. Find the remaining test paths. The total number of test paths should equal the cyclomatic complexity (see above).

   (a) Follow the **baseline path** until the next branch is reached.

   (b) Choose another branch than the ones previously chosen.

   (c) As soon as possible, return to the baseline path.

8. Each test path is a set of decisions. At this point the tester can write out the set of decisions for each test path.

9. For each test path, determine input sets such that the decisions given in the test path are invoked.

10. For each input set, determine the expected results.

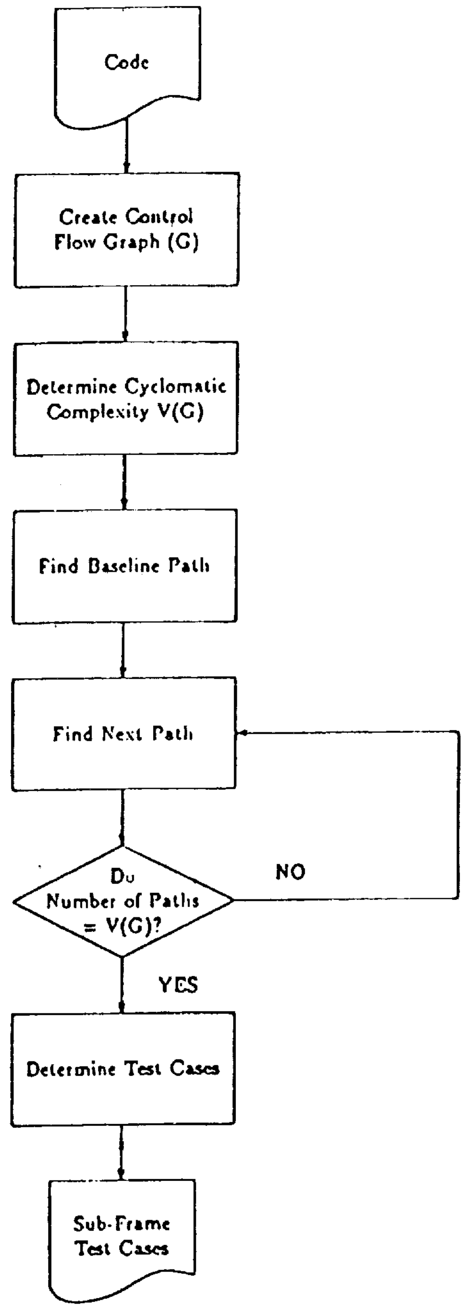

Figure 11: White-Box Sub-Frame Test Case Creation

References

[1] RTCA Special Committee 152. *Software Considerations in Airborne Systems and Equipment Certification.* Technical Report RTCA/DO-178A, Radio Technical Commission for Aeronautics, March 1985.

[2] George B. Finelli. Results of software error-data experiments. In *AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference*, Atlanta, GA, September 1988.

[3] Software Engineering Technical Committee. *IEEE Standard Glossary of Software Engineering Terminology.* Technical Report IEEE Std 729-1983, The Institute of Electrical and Electronic Engineers, New York, New York, February 1983.

[4] Glenford J. Myers. *The Art of Software Testing.* John Wiley & Sons, New York, New York, 1979.

[5] Michael S. Deutsch. *Software Verfication and Validation.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1982.

[6] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold Company, New York, New York, 1983.

[7] *Guide to VAX DEC/Test Manager.* Digital Equipment Corporation, Maynard, Massachusetts, October 1986.

[8] William Hetzel. *The Complete Guide to Software Testing.* QED Information Sciences, Inc., Wellesley, Massachusetts, 1984.

[9] Derek J. Hatley and Imtiaz A. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House Publishing Company, New York, New York, 1987.

[10] Thomas J. McCabe. *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric.* NBS Special Publication 500-99, National Bureau of Standards, December 1982.

National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-101668		

4. Title and Subtitle	5. Report Date
Software Verification Plan for GCS	January 1990
	6. Performing Organization Code
7. Author(s) Leslie A. Dent Anita M. Shagnea Kelly J. Hayhurst	8. Performing Organization Report No.
	10. Work Unit No. 505-66-21-03
9. Performing Organization Name and Address Langley Research Center Hampton, VA 23665-5225	11. Contract or Grant No.
	13. Type of Report and Period Covered Technical Memorandum
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, DC 20546-0001	14. Sponsoring Agency Code

15. Supplementary Notes

Leslie A. Dent, Research Triangle Institute, Research Triangle Park, NC
Anita M. Shagnea, Research Triangle Institute, Research Triangle Park, NC
Kelly J. Hayhurst, Langley Research Center, Hampton, VA

16. Abstract

This verification plan is written as part of an experiment designed to study the fundamental characteristics of the software failure process. The experiment will be conducted using several implementations of software that have been produced according to industry-standard guidelines, namely the Radio Technical Commission for Aeronautics RTCA/DO-178A guidelines "Software Considerations in Airborne Systems and Equipment Certification," for the development of flight software. This plan fulfills the DO-178A requirements for providing instructions on the testing of each implementation of software. The plan details the verification activities to be performed at each phase in the development process, contains a step by step description of the testing procedures, and discusses all of the tools used throughout the verification process.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Software Reliability Verification Plan GCS--Guidance and Control Software Black-box Testing White-box Testing	Unclassified-Unlimited Subject Category 38

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	133	A07

NASA FORM 1626 OCT 86