STATISTICAL MODELLING OF SOFTWARE RELIABILITY

Semi-Annual Status Report No. 2

1 October 1991 through 31 December 1992

National Aeronautics and Space Administration
Grant NAG 1-1241

Douglas R. Miller
Principal Investigator

NAG 1-1241 was originally funded for a one-year period, 1 April 1991 through 31 March 1992. The first semi-annual report for this grant covered the 6-month period: 1 April 1991 through 30 September 1991. The grant was given a 9 month no-cost extension through 31 December 1992. Therefore this report for the second half of the funding period covers the period, 1 October 1991 through 31 December 1992. The work consisted of developing ideas for the design and analysis of controlled software experiments.

Department of Operations Research and Engineering
School of Information Technology and Engineering
George Mason University
Fairfax, VA 20030-4444

Working Paper April 7, 1993

Design of Monte Carlo Simulation Experiments to Investigate Reliability and Failure Characteristics of Guidance and Control Software (GCS)

Douglas R. Miller

Abstract

This working paper discusses the statistical simulation part of a controlled software development experiment being conducted under the direction of the System Validation Methods Branch, Information Systems Division, NASA Langley Research Center. The experiment uses guidance and control software (GCS) aboard a fictitious planetary landing spacecraft: real-time control software operating on a transient mission. Software execution is simulated to study the statistical aspects of reliability and other failure characteristics of the software during development, testing, and random usage. Quantification of software reliability is a major goal.

Various reliability concepts are discussed. Experiments are described for performing simulations and collecting appropriate simulated software performance and failure data. This data is then used to make statistical inferences about the quality of the software development and verification processes as well as inferences about the reliability of software versions and reliability growth under random testing and debugging.

The discussion is not complete. Comments, criticisms, suggestions, additional topics, and corrections are welcome and encouraged. Hopefully, a second draft of this working paper will result which will be a useful document for the statistical simulation part of the GCS experiment.

Contents Page

1. Introduction and Overview

The guidance and control software (GCS) experiment is an investigation of development, testing and reliability of real-time control software. The statistical simulation part of the experiment generates software performance data that can be used to make inferences about software reliability. We are particularly interested in relationships that might be discovered between software development/testing/verification methods and the reliability of the software versions produced.

Experiments investigating software development, testing and failures often just count the number of bugs observed, and perhaps classify them according to functionality and/or severity. From a practical point of view, however, the important measure is reliability, not the number of bugs in the software. We are more interested in the reliability improvement resulting from a particular testing activity than in the number of bugs discovered and removed. A piece of software containing 100 small bugs with an overall average failure rate of 1 failure per 10,000 hours of execution is of higher quality than a similar implementation of software containing 10 bugs with an overall average failure rate of 1 failure per 1000 hours of execution: the operative measure for reliability is the failure rate or the failure probability, not the number of bugs hidden in the software. With this in mind, we want to attach estimated reliability numbers to the software versions developed in the GCS experiment. This is done by simulating replicated executions of the software for a distribution of random inputs representative of real-world usage.

A significant aspect of the GCS experiment is the focus on real-time control software with feedback operating on a transient mission. Previous experiments looked at batch-processed application software. The statistical description of failures for control software is much more complicated than for batch-processed software. It is necessary to define additional failure and reliability concepts for control software. There are two levels of detail in the reliability analysis:

i. mission reliability; and
ii. detailed failure behavior within trajectories.

If we restrict ourselves to mission reliability, then we can treat a sequence of missions as a "batch-processed" application with reliability defined in terms of the dichotomy of mission success and mission failure. In this case the statistical analysis and reliability modelling is similar to previous experiments, e.g. Phyllis Nagel's Launch-Interceptor-Condition (LIC) software experiments. If we want to investigate internal failure behavior within a trajectory, then we are breaking new ground. There are various types of events whose probabilities we might want to estimate. This working paper asks readers to identify such events.

This working paper addresses the issue of the design of software simulation experiments for investigating the failure behavior of real-time control software in the GCS experiment. The simulation will involve two sets of versions for each implementation of the GCS. The first set is the sequence of versions coming out of the software development, testing, and verification process. The second set are the versions created after additional bugs are discovered in the simulated random testing phase and removed.

Simulated execution of the first (verification stage) set of versions will give estimates of the reliability improvement achieved in each stage of non-random testing during the verification stage. This might give insight into relationships between different types of testing and reliability

improvement. This possibility is limited however because under the current GCS experimental plan the verification process is not replicated for each implementation and a common black-box test suite is used for all implementations.

Simulated execution of the second set of versions (those arising from random test and debug) can accomplish (in terms of mission reliability) the same thing as previous replicated-run reliability experiments: failure rate estimation of individual bugs, reliability growth estimation and modelling, and possible observation of bug interactions such as masking and compensation. It will be interesting to examine the bugs that reveal themselves in the second stage (the random run and debug stage). These bugs have remained hidden during a rigorous verification process that is based on DO178A guidelines. It is of considerable interest to estimate the mission reliability of the version coming out of this verification process.

The other aspect of the replicated simulated execution of the various versions of each software implementation (in addition to looking at mission reliability) is a deeper, more detailed look at the failure behavior within trajectories at the frame and sub-frame level. This can be done for all versions. This is a difficult problem from a statistical point of view because of the feedback in control software. Various statistics can be collected such as the frequency of multiple failures in a single trajectory, the duration of a failure burst, the proportion of successful landings in trajectories experiencing failures during some frames. The probabilities of these events can be estimated: these probabilities correspond to the proportion of missions that experience such events. It will be interesting to observe these statistics, but difficult to interpret them in terms of reliability relative to the criterion of the GCS experiment: The criterion for success is correct implementation of the specification, not a successful landing. To assign a probability to successful implementation of the specification after an error has changed the trajectory to a path that has zero probability under error-free execution is impossible. Nevertheless, the detailed failure behavior within a trajectory should be observed and statistical summaries of behavior calculated.

With the above general situation in mind, this working paper suggests how to set up some Monte Carlo simulation experiments to randomly execute versions of guidance and control software (GCS). An important issue is the efficient use of computer resources. Many software versions will be available and additional versions can be produced during random testing. Furthermore, we expect to be dealing with fairly high reliability, so a single version will experience failures on a small fraction of simulation replicates. Deciding which version to simulate and how many replicates for each is an important experimental design consideration. Importance sampling many increase the statistical efficiency of the simulation experiment.

There is a difficulty in identifying incorrect output. The approach advocated in the GCS experiment is to use back-to-back testing: having versions of three different implementations execute in parallel. This is very close to n-version programming except that there is no voter, instead the output of a designated primary version is used. It is easier to make reliability interpretations about some aspects of failure behavior within a trajectory in the n-version scenario than in the proposed back-to-back scheme. So, this working paper raises the issue of running some simulations in n-version implementation.

It is hoped that the failure behavior observed in this experiment can be described by

mathematical reliability models. The statistics of mission reliability may be compared to reliability growth models. The statistics of failure behavior within trajectories may suggest more sophisticated reliability models. It would be nice if relationships between some metrics of the development process and reliability of the software could be mathematically modelled. Also the prediction of reliability from non-random testing failure data should be investigated; this experiment might provide some light on that important subject.

Finally, this working paper ends with suggestions for an initial experimental plan for simulation experiments. The best approach is to proceed sequentially: perform a sequence of experiments, modifying later experiments as a function of the results of the earlier experiments.

2. Experimental Test System

The experimental test system is comprised of guidance and control software (GCS) for a planetary lander, a Monte Carlo test case generator, and a software simulator to run the control software. There are several salient features of this test system (and control SW in general) that should be discussed and clarified.

2.1 Uncertainty in usage environment

The usage environment is the neighborhood and atmosphere of a planet about which we have imperfect knowledge. The uncertain or random context in which the software operates has four aspects: i. Physical parameters of the planet are uncertain (These include gravitation, atmospheric density, optimal drop height, etc.); ii. Variability of planetary conditions during the flight (for example, wind conditions); iii. Uncertainty in the positioning and velocity of the lander at the beginning of the flight; iv. Noise or turbulence during the flight (for example, there could be sensor noise; these effects will lead to different trajectories from the identical initial and background conditions). As a simplifying assumption, the fourth source of noise is ignored by the simulator. RIGHT? This leaves only randomness in the initial and background conditions. So the specification of the input for a particular trajectory requires sampling from several distributions. Note that the uncertainty of type (i) is subjective, while type (ii) and (iii) are objective (frequentist) probabilities. The question arises whether this distinction should be kept in presenting the statistical summaries of failure behavior. How much unreliability is due to subjective uncertainty and how much is due to physical randomness?

2.2 Transient Behavior

This control software performs in four distinct "phases." Each phase is split into "frames." There are approximately 2000 frames in the entire trajectory. Depending where a frame occurs during a trajectory, it gets different input and may be required to perform different functions. Thus, it is not natural in this transient situation to think in terms of generic frame behavior; the events and their probabilities change along the trajectory.

2.3 Control software

One complicating factor in control software is "feedback." This makes things much more complicated than batch processing of independent test cases. Feedback changes the usage environment; successive inputs (at frame level) are dependent. Errors in control software can lead to later inputs (at the frame) level that a perfect piece of SW would never see. These two features (dependence and error-modified inputs) make it tricky to talk about reliability at the frame level. So, if detailed failure behavior of the SW within a trajectory is investigated, the concept of reliability and failure probability must be carefully defined or ignored all together.

Another important aspect of control SW is "inertia." If the control laws are good, the vehicle should be able to fly-through a bad frame (for example, caused by bad sensor input; or, an isolated zero-divide) without causing a crash. This would seem to be an important consideration in collecting and classifying failure data.

2.4 Definition of success and failure

In seems obvious that in the real-world application, a "success" is a safe landing and a "failure" is a crash or abort. Failures would analyzed and it would be determined whether it was a software implementation failure, a control law failure, a SW specification failure, an interface failure, etc. Additionally, in this application the failure might be due to physical parameters whose values were not known to sufficient precision. A reasonable definition of "software failure" is failure to correctly implement the specification. However, if the vehicle lands safely and the software had some minor errors, we may not consider it a software failure. The point is that it seems necessary to have a broader perspective on "success" and "failure" than whether or not the software perfectly implements the specification. Note: if all we care about is perfection, it rules out consideration of recovery from an error within a trajectory.

2.5 Performance measures

There are other performance measures in addition to whether the software perfectly follows the specification. An obvious performance measure is whether the vehicle lands safely. Another concerns recovery from errors during a trajectory. A third is the duration of the failure during a trajectory. There are others. In performing the random-run software execution experiment all relevant data should be collected whenever an anomaly occurs. It is worth thinking ahead, trying to imagine all the possible performance measures to collect data for, and setting up the experiments appropriately. We want to collect more information than whether the software is perfect or not.

2.6 Interpretation of reliability

For control software operating on a transient, terminating mission, we can observe whether the mission is a success. Over many independent replicates, the proportion of successful missions is "mission reliability."

We can also talk about "phase 1 reliability," "phase 2 reliability," etc. We must be careful when assigning reliabilities to parts of trajectories, however. Perhaps, the correct definition of "phase 2 reliability" is the proportion of successful 2nd phases conditional on correct initial conditions at the start of phase 2; if there was a error in phase 1, phase 2 might be correctly executed according to the specification, but from the wrong initial values. The point is that we can talk about various failure probabilities or reliabilities concerning parts of the trajectory but on closer inspect see that the definition is ambiguous for control software in general and this application in particular.

2.7 Single-version and N-version implementations

The current statement of the GCS experiment assumes multiple implementations of single-version software; it also assumes that software correctness is defined as agreement with the specification. But there is an implication that, as part of the GCS experiment, there is an interest in some failure phenomena that cannot be clearly defined in the above context. In particular, statistics of error bursts and statistics of error crystals are hard to define for single-version control software because feedback changes the trajectory. In contrast, for n-version implementations, if there is a failure in one version of an n-version implementation, the voter

will keep the vehicle on the correct trajectory, and the time until recovery will be observed; in this case we know it is a meaningful recovery because the vehicle stayed on the correct trajectory. If a single-version implementation experiences failures, and then later recovers (starts correctly processing frame inputs according to the specification) there is no way of knowing without detailed analysis whether it really is a meaningful recovery in the physical sense; and it is not very exciting to try to collect and analyze such data. But the opposite is true for n-version software: There are some additional phenomena that arise in n-version implementations of control software that are important and can be studied via GCS. Because back-to-back execution of single-version implementations is quite close to execution of an n-version implementation, it is worth considering broadening the experiment in this direction.

2.8 Bug interactions

The phenomena of bug interactions during a single trajectory can be complex for control software. Furthermore, the phenomena differs for single-version and n-version implementations because of the effect of feedback changing the trajectory. A single-version implementation might have multiple bugs that it will encounter on a given trajectory (if it stays on this trajectory), but encountering the first bug might change the trajectory so that different bugs are encountered later. This plays havoc with concepts like the "probability of occurrence of a bug"; which will depend on interacting bugs. An n-version implementation is more likely to follow the "correct" trajectory, so this kind of bug interaction is less likely.

2.9 Ramifications due to novel application and inexperience

The GCS is novel software. DO178a guidelines are being followed. These guidelines seem to be successful in creating very high reliability avionics software. Reliabilities of .999999 and higher are obtained; furthermore, there is a high degree of confidence before the fact. Does this mean we should expect such high reliabilities for GCS? Perhaps not, because the GCS situation is different from the usual DO178a situation. But does anyone have any idea how reliable the software will be? If the software is incredibly reliable there will be no interesting data from random testing of the final versions. So the experimental design must be flexible in order to take the unknown level of software quality into account.

3. Goals of the Monte Carlo Simulation Experimentation

There are multiple goals pursued in performing the replicated, simulated execution of the software. It is probably necessary to set priorities among these goals because there is a limitation on the number of replications that can be run. Many versions are being created during the verification process of each implementation. Furthermore we hope to be working in the realm of high reliability. These two facts suggest a requirement for a huge number of simulated missions.

3.1 Evaluation of DO178A Development Guidelines

DO178A gives guidelines for the development of avionics software. The GCS experiment is following DO178A guidelines in the development of the Pluto, Earth and Mercury implementations. Each implementation will pass through a sequence of verification activities. When bugs are discovered they are removed; this results in a sequence of versions for each of the implementations. These software versions can be used as part of an evaluation of the DO178A guidelines. In particular, we want to see what reliability is achieved.

The versions of each implementation can be subjected to replicated random runs on the simulator. From these runs we wish to collect data and make inferences about the effectiveness of DO178A guidelines. We can do the following:

i. Estimate the reliability of the final version of each implementation.

ii. Estimate the reliability growth during verification and see how much reliability improvement occurred at each step of the verification process.

iii. Estimate the variation of reliability between the different implementations. Low variability suggests that the DO178A process is consistent.

iv. Estimate parameters of individual faults (failure rate and average duration) removed at each step of verification. This is dependent on how the fixes were made (one-at-a-time or in groups) and what versions are available for replicated random testing.

v. Compare characteristics of bugs removed by the DO178A process with bugs that escaped detection and were discovered during the later replicated random testing stage.

vi. OTHER?

3.2 Comparison of Software Testing Methods

The verification process consists of different testing and verification activities in sequence. If the verification process is replicated with different replicates executing test suites in different orders and/or using different test methods, it should be possible to compare different testing methods. The reliability improvement from a testing method depends on the prior reliability of the version being tested. Running the test methods in different orders would allow unbiased comparison between them. We would estimate comparable reliability improvements for each method, for each pair of methods, etc. We could try to estimate variation across implementations. Unfortunately, it seems that replication of the verification process is needed to do much of anything as far as comparing test methods. The verification process of the GCS experiment is replicated only once and furthermore one of the test suites is used for all three implementations. Nevertheless, one the goals is to do whatever we can as far as comparing the efficacy of different testing methods.

3.3 Estimation of Software Reliability

We would like to get estimates of the mission reliability of all the versions available. We can get straightforward estimates from the replicated-run failure statistics for the version.

3.4 Characterization and Quantification of Failure Behavior

Identify and estimate failure characteristics of control software. In effect, we want to go beyond simple mission failure probabilities. Complicated phenomena of bug interaction should be characterized.

3.5 Study of Reliability Growth

We would like to see if reliability growth models can describe the growth of mission reliability during random testing and debugging. We would like to extend the reliability growth concept beyond mission reliability, so that it might be possible to predict other aspects of future failure behavior. One aspect of interest is the failure duration of new bugs causing future failures.

3.6 Investigation of Failure Behavior of N-Version Programs

This experiment can shed light on failure behavior of n-version programs if an n-version implementation is run.

3.7 Development of Statistically Efficient Simulation Methods

Because of the large number of replicates that we anticipate needing, we would like to develop efficient sampling approaches to the replicated run experimentation.

3.8 Posing, Fitting and Validation of Mathematical Models

We would like to fit mathematical models to the data collected and then use them to make predictions. For example, are there relationships between measurements that can be made during the verification process and the resulting reliability?

3.9 Questions

It might be worthwhile to list many questions we might ask in connection with the GCS experiment that we hope will be answered by simulated random execution of the software developed and verified. Here are some of them:

i. Are there any differences between bugs that are detected by systematic non-random testing and bugs that are detected by random testing?

ii. What are the most appropriate measures of reliability in real-time control software? Does the size of a bug equal its probability of occurrence, or should we also appreciate the failure duration for each failure.

iii. What is the reliability importance of the robustness of control laws to small blips? Can we quantify it? Can we exploit it?

iv. Is there any relationship between severity of bugs and their rate of occurrence and their ease of detection?

v. What quantitative information can non-random testing give about reliability?

vi. If bugs with higher failure rates have longer failure durations, does that mean that the system is more likely to be able to fly-through the undiscovered bugs (which probably have shorter durations).

vii. What features of control software require changes in the reliability concepts based on independent batch processing?

4. Test Case Generation

The guidance and control software must execute correctly for many different values of input and environmental variables. This is due to uncertainty and randomness in the application. The Monte Carlo test case generator provides a sample of the possible situations in which the software must run.

4.1 Underlying Randomness and Uncertainty

There are several aspects to the randomness of the operating environment for the software:

i. Physical parameters of the planet are uncertain. These include gravitation, atmospheric density, and various related operating characteristics such as optimal drop height and drop speed. The GCS specification gives ranges for these values. The GCS is expected to operate for cases in which the parameter values are within these ranges. Presumably, there are distributions that reflect the uncertainty of the values of these parameters. NOTE: Are these distributions documented? Or have preliminary simulation runs been using nominal deterministic values.

ii. Planetary conditions during the flight (for example, wind conditions) are uncertain and even variable during the flight. The GCS experiment documentation indicates that the initial values of these environment conditions are randomly generated and then assumes that they remain fixed throughout the remainder of the flight.

iii. There is uncertainty in the positioning and velocity of the lander at the beginning of the flight. According to GCS experiment documentation the initial condition of the lander is described by 10 variables; it is assumed that the values of these variables are drawn from independent (a simplifying assumption) distributions. All of these variables are given ranges in the GCS specification, so the distributions should be over these ranges (or subsets of these ranges). The documentation on usage distribution data seems to contradict this boundedness requirement by suggesting Normal distributions; these Normal distributions would have to be truncated to lie within the ranges given in the GCS specification.

iv. There seem to be additional variables (some control variables, e.g., "gains") that can take a range of values according to the GCS specification. How are the values of these variables determined? Are they part of a random usage distribution?

THE UNDERLYING DISTRIBUTIONS OF RANDOMNESS AND UNCERTAINTY SHOULD BE DECIDED UPON AND CLEARLY INDICATED. SIMULATED RANDOM RUNS CANNOT BE MADE UNTIL THIS IS DONE. The distributions must be kept fixed over the course of the experiment. If it is determined later that some distribution should be changed, this could be done. Depending on the change, some data might have to be discarded and the inferences made from the remaining previous data adjusted to reflect the change; this is similar to importance sampling.

4.2 The Mission Input Distribution

Once all the distributions of underlying randomness and uncertainty are determined and fixed, we in effect have the distributions needed for the initial values and background values for a mission. It appears to be the assumption of the GCS documentation that the only randomness is in the initial values; there is no additional variability or noise in the trajectory. The "mission input distribution" is then the sole source of randomness; this distribution determines all the

variability in the experiment. Other distributions such as phase input distributions and frame input distributions are completely determined by the mission input distribution (as far as randomness is concerned).

4.3 The Random Number Generator

Random samples are drawn from the mission input distribution by generating random numbers and then transforming them into random variates with the desired distributions.

The random number generator that is used should be clearly indicated and documented. There is some controversy over the quality of random number generators. Any widely accepted generator is probably OK for this experiment.

Typically, random number generators provide multiple independent streams of random numbers. These streams usually give 100,000 random numbers before they start overlapping with neighboring streams. If multiple streams are used, care must be taken to avoid this overlap.

4.4 Random Variate Generation

Random variate generation will be a minor fraction of the computation in the random run experimentation. Nevertheless, efficient and accurate algorithms should be used. Since most of the random variables seem to be on bounded domains, the Beta distribution is a possible candidate for some of the variables. So, a good Beta generator should be found and used.

4.5 Phase input distributions

Each phase has an input distribution. The distributions are different for each of the four phases. The input distribution to phase 1 is identical to the mission input distribution. The input distribution to phase 2 is determined by the mission input distribution and "correct" trajectories from the start of the mission to the start of phase 2. A mission input vector and a correct software implementation (a gold version?) creates a trajectory; the values of all the necessary variables at the start of phase 2 correspond to the input for phase 2. The problem is that there is no single "correct" trajectory starting from a single mission input; different implementations may be within specification and still give different states at the beginning of phase 2. Thus, it is difficult, if not impossible, to define input distributions to phases 2, 3, and 4. We are tempted to want such distributions because they could be used to focus Monte Carlo simulations experimentation on particular phases and be used for the definition of phase reliabilities. This approach does not appear to be worth pursuing. Probabilities of events involving a single phase can be determined from entire missions.

4.6 Frame input distributions

Frame input distributions suffer from the same difficulties as phase input distributions. But, it would be nice to have rough idea of the frame input distribution for evaluation of the non-random testing that is done at the frame level; it would be nice to see where the test cases fell in the true distribution.

4.7 Importance Sampling

It is not necessary to sample exactly according to the usage distribution. We might think that some values of input variables are more likely to be interesting (i.e. result in mission failures), so we can bias the sampling to give these values more weight in the sampling scheme. If this is done right, the bias can be removed in the statistical analysis of the failure data. This technique is called "importance sampling." Importance sampling can improve the efficiency of the simulation.

Other types of importance sampling might involve perturbing a trajectory which has shown failure behavior. The perturbation might be centered about the mission input values for that trajectory, or it might be centered about the state values within the trajectory close to where the failure occurred. Also, if implementations are tested independently, we probably would want to run all the implementations on the particular inputs that caused failures in other implementations; this is non-random sampling. These approaches might be useful in uncovering more failures, but it is not clear that the bias in these particular sampling schemes can be removed.

5. Design of Experiments

To design a statistical sampling experiment, one should first specify the inference being attempted: e.g. estimation of some parameter, or test of some hypothesis. Then, details of the sampling must be specified: what cases are to be run, how many replicates are to be taken, what data is to be collected, etc.

There are two or three general types of Monte Carlo simulation experimentation involving simulated execution of the guidance and control software (GCS):

i. The first experimental situation involves the sequence of versions of each implementation produced under the DO178A development guidelines. For the Pluto, Earth and Mercury implementations we will receive a sequence of versions:

P(1),P(2), ... , P(VP)
E(1),E(2), ... , E(VE)
M(1),M(2), ... , M(VM)

where the final version is the version released from the DO178A process, and there many be a different number of versions of each implementation, and successive versions correspond to the correction of one or more bugs. (Furthermore, we would like to assume that no bugs are introduced during the verification process, so the sequence of versions has strictly improving reliability.) In this situation, we are generally interested in making quantitative inferences about the reliability and failure behavior of the actual versions delivered from the verification process.

ii. The second experimental situation involves the final versions of each implementation:

P(VP) = P*
E(VE) = E*
M(VM) = M*

We perform random simulated-execution testing and debugging on these versions, creating new versions as bugs are removed. We have the choice of making one long run or several shorter replications of the debugging process. Also we could make replicated runs with no debugging of the original versions or their descendants. The experimental goal is make inferences about the reliability of the software and failure characteristics of individual bugs present.

iii. There is a third possibility. We could subject pre-release versions to replicated random testing and debugging. The goal would be to get more accurate information about failure characteristics of bugs present in the versions but removed before the implementation is released from the DO178A process.

In all experimental cases there are common design issues. There are special design issues for reliability estimation of a sequence of versions. There are special design issues for replicated random testing and debugging experiments. There might be some special experimental design issues for performance measures that involve detailed failure behavior within a trajectory, but that is mainly involved with data collection. But the approach in this working paper is to assume that such behavior is observed by simulating entire trajectories so the same design issues arise in both cases.

5.1 Error Detection

Defining and detecting errors is non-trivial.

5.1.1 Definition of Error

Due to the nature of the test application and the fact that the execution of the software is being simulated, the best definition of error or failure for this experiment seems to be that the output calculated from valid inputs does not agree with the specification.

5.1.2 Back-to-back testing

A priori, it is hard to always tell for sure whether the software output the correct value. So back-to-back testing is used to look for discrepancies. This is not guaranteed to catch all errors, so the reliabilities estimated in this experiment are conditional on the fact that undetected failures may have occurred.

5.1.3 Triads or Diads

In back-to-back testing it might be more efficient to run diads (two versions) instead of triads (three versions). If a non-compare is observed in a diad, then it could be resolved by re-running the test case with a third alternate version added to back it a triad. If the software has high reliability, only a small fraction of the cases would be re-run. For the cases with no disagreement, there will be a reduction of between 25% to 30% in execution time depending on how much time the simulator takes. The disadvantage is that the two versions in a diad could agree and both be wrong, while this is much less likely for a triad; so a diad might miss some failures that a triad would catch. This is typical of trade-offs in this experiment: how much effort should be spent trying to see the extremely rare failures in contrast to seeing more occurrences of the less rare failures. It depends in the general level of reliability of the versions being tested.

5.1.4 Choice of secondary versions

The role of secondary versions in a back-to-back triad or diad is purely error detection for the failures of the primary member. So, the most reliable version of different implementations available should be used.

The gold version might be considered as a member of a triad. The disadvantage of using the gold version is that we may have no interest in estimating its reliability or discovering hidden bugs in it, so anecdotal failures observed while it is a secondary member are of less value than observing anecdotal failures of the best version of Earth, Pluto or Mercury.

5.1.5 Investigation of Tolerance and Drift

Control software deals with real-valued variables. Different algorithms may give different output, both within specification. The specification describes the correct result at the frame level. Small discrepancies at the frame level can accumulate into major discrepancies over the length of a trajectory: this is "drift." Is there any indication of how large this effect might

be and how it might effect attempts to define things like frame input distributions?

An experiment to look into this could be based on different implementations (as primary versions in separate triads) starting from the same initial point. In particular, we could run 3 implementations separately and look at the maximum deviation between all pairs of frame-level ouputs, plus deviations of all terminal values. Then evaluate the amount of deviation due to drift; from this decide if drift is a problem. The deviation values at the end of the mission might be useful for initial screening in error detection in back-to-back testing.

If it turned out that different implementations always behaved identically, then it might be advisable to change some experimental design decisions to take advantage of this and increase the statistical efficiency of the experimentation.

5.1.6 Identification of Bad Frames

Will it always be obvious in which exact frame any given failure occurs?

5.1.7 Secondary failure data

When there is a non-compare in a triad, it may be due to failure in one of the secondary members of the triad. Since we are using the best versions for the secondary members, this will be a new failure. This information is useful because it is additional failure information and we can fix the version and improve its reliability. However, this failure was not detected by random testing of the secondary version, therefore this failure observation appears to have no statistical value for reliability estimation. In fact, if this bug never manifests itself under random testing we will only be able to get an upper bound on its failure probability based on zero observation in so many random test cases.

5.2 Choice of test cases and runs

Each test case is determined by the values of 12 or more input variables. These values are generated by Monte Carlo simulation. Therefore we have more control that in physical sampling and we might want to take advantage of this control to improve statistical efficiency.

5.2.1 Common vs. Independent Input Cases

In testing different implementations it seems that independent test cases should be used for the different implementations. If the different implementations have dependent failure behavior then we might get a more accurate estimate of differences in reliability if common test cases are used by all implementations. The advantage of using independent test cases is that we will be looking at three times as many test cases and thus expect to have three times as many difficult test cases. We can take the test cases that one implementation fails on and run the other two implementations on these test cases: We expect to see a high failure rate and might discover new bugs. The additional failure data is hard to use in statistical inferences, but we will have seen additional failures and perhaps discovered new bugs. So on balance, it seems that it is better to use independent test cases for the different implementations.

5.2.2 Fractional Designs

A fractional design would use common test cases for a fraction of the implementations. This may not be worth pursuing because it seems that total independence of test cases across implementations is desirable.

5.2.3 Conditional Sampling of Test Cases

It seems that the occasion might arise where we would want to fix or limit the range of some mission input values while sampling randomly from the remaining variables. For example, we might want to estimate the reliability given that wind velocity equals zero.

5.3 Experiments for Mission Reliability

The single most interesting reliability values estimated in this entire experiment is probably that of the versions released from the DO178A development process. The estimation of these reliabilities is straight-forward. Independent random test cases are generated, the versions are executed as the primary member of triads (or diads), and success or failure observed.

5.3.1 Assigning Blame to Input Values

If the reliability of the DO178A final versions is disappointingly low, we might try to blame it on the input distributions. Presumably the version has worked correctly for nominal values of the mission input variables. Perhaps the distributions of one or a set of mission input variables give high probability to inputs that the specification treats lightly.

5.4 Efficient Sampling for Multiple Versions

We are given a sequence of versions A(1), A(2), ... , A(k) of implementation A. We wish to estimate the reliabilities of the individual versions and also the differences in reliability between pairs of versions.

The crude way to design this experiment would involve taking independent test cases: n(i) test cases for version A(i), i=1,2,...,k. The reliabilities and the differences in reliability can be estimated using sample failure proportions; also, estimation errors can be estimated. This is the standard statistical approach. A variation of this approach would use the same n test cases for all versions; this requires k*n replications.

If we make the assumption of strict reliability improvement and assume that no faults are introduced or un-masked during the verification process that gave birth to this sequence of versions (regression testing might encourage this assumption), then a much more efficient design takes advantage of this. Test the first version A(1) with n independent test cases and observe f(1) failed cases; the estimate of the reliability of A(1) is f(1)/n. By assumption, A(2) will never fail unless A(1) fails; therefore, test A(2) with the f(1) test cases that caused failures in A(1) and observe f(2) failures; the estimate of the reliability of A(2) is f(2)/n. Continue this scheme for later versions in the sequence. For reliable software the total number of replications will be slightly more than n, a great savings over k*n. This sampling procedure will miss bugs introduced during the verification process; the trade-off with statistical efficiency must be

considered.

An alternative design might be based on a mixture of the above two sampling procedures. Version A(i) is tested with the test cases for which previous versions failed plus a additional set of n(i) independent test cases. The previous versions are all tested on the test cases among the n(i) for which A(i) fails; this will partially address the problem of fault injection during the verification process. Optimal choices for the n(i)'s and the statistical analysis would have to be developed for this design: we would like to minimize the variance of our reliability estimates.

5.5 Replicated Debugging Experiments

The first major reason for replicating the debugging process is to see more than one failure caused by any given bug; this will result in much more accurate estimates of the failure rates of individual bugs. A second major reason for replication is to allow for the discovery of bugs in different orders of occurrence. This gives a partially order set of versions rather than a linearly ordered sequence of versions (as results from one replication of the DO178A verification process). The advantage of all these versions (as many as 2^n if there are n bugs) is that bug failure interactions (masking, compensation, etc.) can be observed.

5.5.1 Number and Length of Replications

The design decision must be made concerning the length (number of test cases) of a debugging run. The number of runs must also be decided. Should we make a few long runs or many short runs?

5.5.2 Design with Partial Debugging

If you do not believe there is any significant bug interaction or you are not interested in studying it, then there is no value to replicated debugging. It is best to make one debugging run, get a linearly ordered sequence of versions. Then these individual versions can be subjected to replicated execution without debugging. Enough replicates can be taken to achieve a desired level of accuracy of the reliability estimates.

5.5.3 Phyllis Nagel's Statistics for Replicated Run

In her LIC experiments, Phyllis Nagel did replicated debugging runs. In each replication she identified bugs according to the order of occurrence. She focused on the failure rate between discoveries (test stages) rather than the failure rates of particular bugs. She was able to see variability in the reliability growth process this way. She aggregated statistics: she calculated the average failure rate between the ith and the i+1st failure over many replicates of the debugging process. This statistic is hard to interpret.

5.6 Experiments to Compare Test Methods

The current GCS plan does not replicate the verification process. Therefore the design of experiments to investigate various aspects of non-random testing need not be addressed.

5.7 Experiments with N-Version Implementations

The design of experiments for n-version implementations is pretty straight-forward here because we only have 3 single-version implementations. There is only one n-version implementation possible: a 3-version implementation consisting of Pluto, Earth and Mercury. The best version of each implementation could be used. Random independent executions of the 3-version implementation could be run and failure behavior observed. The mission reliability of the of the 3-version implementation could be estimated and compared with reliabilities of the individual versions. Furthermore, statistics on error recovery by individual versions and duration of failures of individual versions could be collected. Finally statistics related to the independence of individual versions failing could be collected and tested.

5.8 Experiments on Complicated Behavior within Trajectories

This working paper recommends the approach of simulating whole entire missions based on the mission input distribution. The trajectories can then be examined for the occurrence of various events involving complicated failure behavior within the trajectory. A list of such events is given later in this paper. This approach is preferred to trying to simulate part of a trajectory starting with some distribution such as a phase input distribution or frame input distribution.

5.8.1 Observing Perfect Recovery

Perfect recovery means that a failure occurred for one or more frames and then the software resumed correct execution for the rest of the trajectory and the remainder of the trajectory was the correct trajectory. The only way perfect recovery can be verified is by running the same input case with a version that has the bug removed and compare trajectories. This event cannot be recognized by simply looking at the single individual trajectory.

5.8.2 Experiments on Partial Trajectories

Not recommended because of the problem of statistical definition and interpretation of input distributions other than the mission input distribution.

5.9 Perturbations of initial conditions and trajectories

One of the goals of the random testing part of the experiment is to investigate the geometry of the set of inputs that cause failure due to a particular bug. Aspect of this investigation could involve perturbing the inputs. Sampling additional inputs in the neighborhood of a previous input that was randomly drawn from the usage distribution and caused the version to fail: the actual sampling distribution might be a 12 dimensional Normal with mean equal to the failed input. There is a possibility the statistics can be developed so that failure data so collected can be used in failure probability estimates for this bug that are more accurate than estimates obtained from purely random sampling.

Even trickier closer looks at failure behavior of bugs might be obtained by the experimenter replicating a interesting trajectory up to a point (where it started to get interesting) and then perturbing it in one frame to get a slightly different trajectory for the rest of the mission. Observations would be made to see if the same failure behavior occurred in the

perturbed trajectory. This would give some idea of the size of the error crystal in the trajectory set.

5.10 Data Collection

The general philosophy of data collection is to ignore the missions that are correct and to save as much information as possible about the missions that failed. We try to anticipate the data we wish to save from each replication. If we save the seeds and the versions in the triad and whether any anomaly was observed, then we can rerun the interesting cases and collect more data if necessary.

5.11 Non-random Sampling

One of the goals in the random testing part of the GCS experiment is to discover as many bugs in the final DO178A versions as possible. Additional bugs might discovered by running the version on all input cases that causes some other version or implementation to fail. This is an example of non-random testing.

5.12 Tactics to Improve Statistical Efficiency

There is a good chance that we will want to run more random replications than there is time or computer resources available. The desired amount may be an order of magnitude more than is possible. Thus statistical efficiency is important. We should be prepared to accept lower accuracy in reliability estimates. We should also be prepared to make simplifying assumptions that ignore some second-order software phenomena in order to get the pafoff of smaller sample sizes.

5.13 Sequential Designs

The general experiment should be done in stages. The design of the next stage depends on results of the previous stage.

5.13.1 Sequential design decisions

In the first stage of the sampling experiment we should estimate the reliability of the final DO178A versions of all implementations. Depending on the level of the reliability, we will plan to do different things in the next stage. If there are no bugs discovered in these versions, we might devote a lot of effort in the next stage to a long run looking for at least one failure, or perhaps importance sampling. If the reliability is low, we might look more closely at the bugs that were caught during verification and see how they differ from remaining bugs, or we might investigate the possibility of bug insertion during the verification process.

5.13.2 Sequential sampling

There is usually a question about sample size. A sequential sampling scheme stops when a desired accuracy is reached. The precise statistical theory for sequential methods is much more complicated than fixed-sample size procedures. But it is within the level of rigor and

accuracy desired for this experiment to use fixed-sample size procedures sequentially. So sequential sampling is possible. It will contribute to efficiency.

5.14 Miscellaneous Design Issues

5.14.1 Role of Gold Version

Bernice Becker created an implementation: Venus. There should be some use for it in the experimentation.

5.14.2 Miscellaneous Other Software Implementations

In addition to the three implementations developed in the controlled DO178A environment, there are various other implementations. Implementations were created at the College of William & Mary, at Old Dominion University, and at Syracuse University. The question arises whether these implementations can and should be used in some capacity within the simulated random run experiments. Basing an experiment on just three replicates (Pluto, Earth and Mercury) limits any inferences that can be made about the variability of failure behavior.

6. Statistical Inferences about Mission Reliability

We shall use various statistical estimation procedures to make inferences about mission reliabilities.

6.1 Reliability Estimation of a Version

Suppose that p is the unknown probability of mission failure of a software version for inputs drawn from the mission input distribution. We wish to estimate p.

A random sample of k independent mission observations is simulated; f of the k result in mission failure. The estimate of mission failure is the sample proportion f/k. The variance of the estimate is known to have the form p(1-p)/k; an estimate of this sample variance is (f/k)(1-(f/k))/k. For high reliabilities the sample variance is approximately equal to p/k and the standard deviation is approximately equal to (p/k)^(1/2). We can use the variance (or standard deviation) of the estimator as a measure of estimation error. For example if p=.0001, and k=1000000, then the standard deviation of the estimator is .00001, or 10% of the value of p.

It is more efficient to estimate the failure probability p sequentially rather than using the above fixed-sample size procedure. We would like to estimate p with some relative precision, i.e., an estimation error (standard deviation) equal to some percentage of its value. A sequential sampling and estimation procedure can do this efficiently. An acceptable approximate procedure is to use the above fixed-sample estimator in a sequential mode. If the desired relative error is 100*e%, then sample sequentially until the estimate of the relative standard deviation of the sample proportion is less that e: ((f/k)/k)^(1/2) < e*(f/k). The estimate is then f/k with approximately the desired precision.

It can be shown that the above sequential procedure has the following approximate properties. The expected sample size is 1/(p*e^2) and the expected number of failures observed is 1/e^2. For example, if p =.001 and e=.2, the expected sample size is 25000 and the expected number of failures observed is 25.

The sequential approach will use sample size more efficiently. Also the above numbers give an indication of the sample sizes required to achieve a certain level of precision for failure probabilities in various ranges. A further look at the sequential statistical literature is recommended for fine tuning this procedure.

6.2 Reliability Estimation for a Sequence of Versions

If we assume that the sequence of versions produced by the DO178A process has strictly improving reliability (no bugs are introduced or unmasked in the process), then it is possible to efficiently estimate the individual reliabilities of the versions. This assumption means that a later version never fails on an input that an earlier version that has already successfully executed; this means that earlier successful test data can be used on later versions without rerunning the test cases.

Here is a sequential procedure for estimating the reliability of versions A(1), A(2),..., A(k). Set a desired level of relative error for the failure probabilities: 100*e%. Sequentially estimate the reliability of A(1); suppose f1 failures were observed in n1 test cases, then the estimate of the mission reliability of A(1) is 1-f1/n1. Now proceed to version A(2): using A(2), rerun the f1 cases for which the previous version failed; and observe f20 failures of A(2).

For the purposes of sequential estimation treat this data as f20 failures in n1 test cases, and continue with the usual sequential sampling until the relative error condition is met for the failure probability of A(2), an additional n2 test cases with f2 failures giving an estimate of the A(2) mission reliability as 1-(f20+f2)/(n1+n2). However, it was not necessary to run A(2) on all n1+n2 test cases. The estimation scheme can be continued: the f20+f2 failure cases from version A(2) are executed using version A(3) and f30 failures are observed, and additional n3 test cases are run for A(3) based on sequential stopping rule and f3 more failures are observed. The reliability estimate for A(3) is then (f30+f3)/(n1+n2+n3). And so on, estimating the reliabilities of all the versions.

6.3 Reliability Improvement Estimation for Verification Versions

In addition to the estimates of reliability of the individual versions produced in the verification process, we also want to estimate the difference of reliabilities of the successive versions. The point estimate is simply the difference of the reliability estimates of the individual version obtained above. But we would also like to have an estimate of the estimation error. Furthermore we might like to specify in advance a level of relative precision and use a sequential sampling scheme to achieve it.

If we take the approximate approach of using fixed-sample statistical properties, we can certainly estimate the variance of the difference of the above sequential reliability estimates; this will be a larger relative error than the individual estimates. If we want to sequentially estimate the difference to a desired level of precision, a more complicated sampling scheme than described above will have to be developed. This can probably be done, if we are content with an approximate sequential procedure based on fixed-sample size properties.

6.4 Comparing Implementations

We can compare implementations by estimating the difference of the reliabilities between pairs of implementations. There is no short-cut proposed here. Each implementation is tested on n test cases. The test cases for different versions can be different or they can be the same. The differences in reliabilities can be estimated. Sequential procedures could be used for efficient sample size.

If the three versions have unknown mission failure probabilities p1, p2 and p3, then it could be assumed that the DO178A process creates implementations with random probabilities of mission failure drawn from a distribution with mean pp and a variance vp. The variance vp gives a measure of how variable the DO178A process is in regards to the reliability of the implementations produced. This variance could be estimated, based on the assumption that p1, p2 and p3 are independent samples from the underlying distribution. We might be able to get error estimates of this variance estimate. We should carefully consider the fact that common test suites were used in the DO178A process; the assumptions about the underlying distribution of p must be carefully stated.

6.5 Reliability Growth of a Single Debug Run

We should look at the reliability growth of the single replicate of the verification process. This is not based on random testing but there might still be a pattern that can be modelled. The successive versions may correspond to the removal of more than one bug, but that should not

matter as far as reliability growth is concerned. The important thing is the number of executions and the number of failures between successive versions.

6.6 Reliability Growth of Replicated Random Debugging Runs

The usual valid reliability growth scenario is a single debugging run based on random testing. The goal is to predict the times and number of future failures. In replicated runs, the data is quite different: we see individual bugs more then once. At the end of the multiple replications, we have a version with all observed bugs removed. We would like to use the replicated run failure data to predict future failures of this version as well as its reliability growth under further debugging. This requires extending existing reliability growth modeling techniques. A comparison should be made between the reliability predictions based on one long run verses the reliability predictions based on several shorter runs.

6.7 Effect of Bug Interactions on Mission Reliabilities

If there are two possible bugs (B1 and B2) in an implementation, then the effect of any interaction between them can be described in terms of the mission reliabilities of three different versions: V12, V1, V2, which are the versions with both bugs, the version with just B1, and the version with just B2, respectively. If the reliability of V12 is higher than the reliability of V1 or higher than the reliability of V2, then interaction exists. So determination of interaction requires estimating the reliabilities of V12, V1, and V2 and the differences. This is done by executing all three on a common set of test cases; it is important to use common test cases to get a small estimation error.

Thus, the versions resulting from the replicated random testing and debugging should be tested on common input cases if bug interactions effecting mission reliability are of interest. If bug interactions are ignored and bug insertion during debugging is ignored, then it might be more efficient to not execute all versions on common test cases.

6.8 Estimation of Reliabilities of Partially Ordered Versions

The multiple new versions created during replicated random testing and debugging will not be an ordered sequence of versions like the those produced during the single replication of the DO178A process. Instead there will be a partial order on the versions. Under the above assumptions of no bug interaction and not bug insertion during debugging, these versions can be tested efficiently using a modification of the above sequential procedure to estimate version reliabilities and differences in reliabilities.

6.9 Investigation of Failure Regions (Error Crystals)

For each version, we have a subset of the 12-dimensional input space for which the version fails. The Statistical Computing Group at George Mason University claim to have data visualization techniques that are sophisticated enough to see patterns in 12-dimensional space. It would be interesting to give them some realizations of these failure subsets of the input space for different versions and see if they can see anything. Also the error crystals in the trajectory space could be examined.

7. Detailed Failure Behavior within Trajectories

There are many performance measures in addition to mission reliability. Mission reliability is simply based on the events of mission success and mission failure. Additional measures are based on more complicated behavior during the trajectory than whether all frames are executed correctly according to the specification. We would like to estimate the probabilities of these various more complicated performance measures.

7.1 General Structural Decomposition

One way of looking more closely at the behavior within a trajectory is to break the trajectory down and look at frame behavior. What is the probability of successful execution of a single randomly chosen frame? This is the "frame reliability." It is hard to compute this probability because the feedback in control software makes it hard to define what is meant by frame input distribution. Because of this difficulty, this working paper advocates avoiding analyses that encounter concepts like phase input distributions or frame input distributions. Instead, detailed trajectory behavior should be approached by identifying the appropriate events for the entire mission trajectory. The probabilities of these events can then be clearly defined in terms of the mission input distribution. The probabilities of these events can be estimated by the sample proportion of sample missions for which the event occurs.

7.2 Specific Trajectory Events for Single-Version Programs

Complex behavior within a trajectory is defined in terms of appropriate mission events. For this experiment it would be useful to try to list in advance all the events that might be of interest. Whenever a trajectory is sampled, the occurrence or nonoccurrence of all events in this list could be noted. The estimate of probability of occurrence for a particular software version will be the sample proportion of missions for which the event occurs.

7.2.1 Events involving single versions

Some events of complex failure behavior within a trajectory for single versions are:

a. Vehicle crashes (may not be defined as failure)
b. Vehicle lands safely (may not be defined as a success)
c. An error occurs for several frames but SW recovers
d. An error occurs, SW recovers, and a second error occurs
e. An error occurs in phase one
f. An error occurs at interface between phase 1 and phase 2
g. Etc.

7.2.2 Statistical values observed in trajectories

Instead of whether particular events occur in a trajectory, it may be more efficient to observe the value of some statistics. For example:

a. The number of bad frames.
b. The number of consecutive bad frames (duration of failure).
c. The number of bugs encountered

d. Etc.

7.2.2 Events involving multiple versions

There are events of complex failure behavior within the trajectories of multiple versions starting from the same initial point. Typically we compare the behavior of two versions A1 and A2, where A2 might be a fix of A1. Some events are:

a. Bug interaction: A1 hits bug1 and recovers and A2 hits bug2 and recovers (where A2 is the same as A1 but bug1 is corrected)

b. Two implementations foloow the same trjectory, i.e., they land within epsilon of each other.

c. Etc.

7.3 Observing Perfect Recovery

We define "perfect recovery" from a fault if the version executes incorrectly for some frames, then resumes correct execution and the resumed trajectory is identical to the trajectory that the version would have followed if the fault was not present. This is a special case of events involving two versions. The versions with and without the fault are executed on identical inputs. The versions should follow the identically same trajectory if no failure occurs. If the fault causes a failure and the version recovers, we observe whether or not it continues on the identical trajectory of the correct version. To accomplish this, it certainly suffices to observe whether the two versions terminate trajectories in exactly the same way and with the same values of state variables.

7.4 Specific Trajectory Events for N-Version Programs

There are many trajectory events on interest for n-version programs. Some of them are:

a. Failures occur in two versions but do not overlap frames, so the n-version implementation does not fail.

b. A version fails for exactly 1 frame

c. Etc.

8. Importance Sampling

One way to gain statistical efficiency in the simulation experiment is to exploit the Monte Carlo sampling of test cases. We are not restricted to simple random sampling from the mission input distribution to generate test cases. We can bias the Monte Carlo sampling to favor more interesting test cases; this may lead to reduced variance of the estimates based on the same sample size. Biased sampling to favor "important" values is called "importance sampling." In order for an importance sampling scheme to work, the bias in the sampling must be removed in calculating the estimates. There are several possibilities for the GCS simulation.

8.1 Variance Reduction from Importance Sampling

Here is a batch processing example of importance sampling that reduces the variance of failure probability estimates.

Suppose that there are 10^10 possible inputs for a software program; all of these inputs are equally likely. Suppose that of these inputs there are 10^6 bad points on which the SW fails; so, the probability of failure is p=10^-4. Suppose we test this software by running it on 10^5 randomly chosen inputs; we expect to see 10 failures. We estimate the failure probability with the sample proportion of observed failures; if the software fails nf times in ns random executions, then sp=nf/ns is the estimate of the failure probability p and the variance of the estimate is p(1-p)/ns. For this particular example the variance of the estimate of the probability of failure is 10^-9 and the standard deviation is 10^-4.5.

Suppose that we can recognize that there are essentially two types of input: easy and hard. The easy inputs account for 3/4 of all inputs; the remaining 1/4 of the inputs are hard inputs. It turns out that 3/4 of the bad inputs are among the hard inputs, while the remaining 1/4 of bad inputs are among the easy inputs. Let pe be the probability of failure for a random easy input; then, it turns that pe=.0000333. Let ph be the probability of failure for a random hard input; then, it turns that ph=.0003000. So, ph is 9 times grater than pe. Note that pe*3/4+ph*1/4=0.0001=p, agreeing with the above calculation. Instead of drawing a simple random sample from the mission input distribution, we do importance sampling. We take a sample of size 10^5, but decide that we will randomly draw equal numbers of easy and hard inputs: the number of easy drawn is nse and the number of hard drawn is nsh (nse=nsh=5*10^4). In this sample, a hard input is three times more likely to be drawn than it would be in a simple random sample. The easy inputs are weighted by we=.25 and the hard inputs are weighted by wh=.75. We observe the number of failures among easy inputs and among hard inputs: nfe and nfh, respectively. We expect that nfe will equal 5/3 and that nfh will equal 15, so we expect to see more failures. An unbiased estimate of the failure probability p is (3/4)*nfe/nse+(1/4)*nfh/nsh. The variance of this estimator is (3/4)^2*pe(1-pe)/nfe+(1/4)^2*ph(1-ph)/nfh = (27/32)*10^-9, for this numerical example. We see a 5/32 = 16% variance reduction. In other words, we could get the same statistical precision with a sample that is 16% smaller than the estimation based on purely random sampling.

This example illustrates that, if we can identify more failure prone inputs and sample from them more heavily, we can achieve higher precision estimates of failure probabilities.

8.2 Importance Sampling from Mission Input Distribution

The simulation of GCS is driven by the mission input distribution. So, we would like

to importance sample from this distribution. To do this, we must have some knowledge or some intuition about what input points are more likely to fail and then arrange the sampling so that these points are favored. This is difficult because of the definition of mission success: the favored points must be ones for which the specification is more likely to be violated, not necessarily inputs for which the vehicle is more likely to crash. (It would be interesting to check the intuition of the software developers, programmers and testers.)

8.2.1 Sampling from Marginal Input Distributions

The 12 or so input variables are assumed to take values independently (a questionable assumption), so the values can be sampled independently of one another. The specification gives restricted ranges to all these variables. The distributions are either uniform over the range or some type of bell-shaped distribution (Beta or truncated-Normal). A good candidate for importance sampling distributions is the U-shaped distribution. U-shaped distributions give more mass to the ends of the ranges of the variables than to the middle values. We could sample independently from 12 U-shaped distributions; we would get independent coordinates to our input points.

8.2.2 Sampling from Joint Input Distribution

A more general way to importance sample would be to use a general distribution for which the separate variables are dependent.

8.2.3 Rejection Method

One way of sampling is to define a weight function over the input space, $w(x1,x2,x3,\ldots,x12)$ takes values between 0 and 1. The higher the weight, the more interesting the input point. We sample randomly using the mission input distribution; we accept a sample point with probability equal to its weight and reject a sample point with probability equal to 1-w. We sample from the original mission input distribution and accept or reject each point until we get the desired sample size of accepted points; we use these points as our importance sample.

An example of a potentially good weight function would be a normalized measure of the distance of an input point from the nominal input.

8.2.4 Unbiased Estimation

If we know the original underlying mission input distribution and the importance sampling distribution, we can remove the bias in the importance sampling by weighting the observations by the ratio of their likelihoods under the original distribution and the importance sampling distribution. If we guessed right in picking an importance sampling distribution, we will get a variance reduction.

8.3 Importance Sampling from Frame Input Distributions

If we can get a valid frame input distribution, we could perform importance sampling at the frame level. Because the specification is given at the frame level, it might be easier to pick an importance sampling distribution at this level.

8.4 Other Biased Sampling Methods

There are other instances where we might want to bias the sampling. Investigations of error crystals could be done by sampling in the neighborhood of inputs on which failure has occurred. This could be accomplished by perturbing initial conditions. Or we might want to make multiple perturbations of a trajectory during the mission. The problem with such sampling methods is that it may be impossible to remove bias from estimates based on these samples.

8.5 Cluster Sampling

A related sampling method is cluster sampling. Instead of sampling independent observations from the mission input distribution, we sample clusters. In the first step a sub-sample of independent observations is drawn. In the second step a cluster of observations is drawn from the vicinity of each of the original observations. This can be done is such a way that sample proportions are unbiased. This type of sampling might be useful in looking for failures on multiple trajectories due to the same bug. Presumably, a given bug is likely to hit more than one case within a cluster.

9. N-Version Programming

It is recommended to include n-version programming in this experiment. The failure behavior of n-version programming is of considerable interest. DO178A may not give explicit credit for redundant software, but redundant software is being used in flight critical commercial aeronautical applications. Avionics and airframe software engineers have been heard to claim that redundant software gives them an order of magnitude improvement in software reliability.

The three implementations (Earth, Mercury and Pluto) could be run in a 3-version configuration. Failure data could be collected and performance evaluated. Some important issues to be investigated are:

i. quantification of reliability improvement from a 3-version implementation over single-version implementations.

ii. estimation of the duration of failures

iii. relationships between the probability of occurrence and the distribution of duration of occurrences.

iv. probabilities of over-lapping failures in two versions in the 3-version implementation.

v. Estimation of the degree of independence of failure behavior before and after the common black-box testing.

10. Models and Metrics

There are several opportunities for mathematical modelling in connection with the GCS experiment.

10.1 Reliability Growth Modeling with Replicated Debugging

The usual reliability growth scenario is a single debugging run based on random testing. In this experiment, we will have replicated debugging runs. Reliabiity growth modelling should be extended to this situations.

10.2 Module-Level Reliability Modelling

The usual reliability model treats the program as a black box. In this experiment the software has 11 modules. Failures will be caused by one or more modules. Data on the source of failures can be collected in the GCS simulation. Detailed reliability modeling incorporating the failure behavior and sources should be attempted.

10.3 Reliability Models of Occurrence and Duration of Failures

Previous simple reliabiity models dealt with the occurrence of failures. Control software is more complicated and the length of failure duration is very important. An attempt should be made to extend reliability models from just failure rates of bugs to the joint distribution of failure rate and failure duration. It might turn out that there is a positive correlation between rate and duration. This might be a good result because it could imply that the rarely occurring failures (the undetected ones) have short durations and therefore are less likely to have catastrophic consequences when they do occur. A model should be developed and the correlation or some other measures of dependence estimated from experimental failure data. In the reliability growth scenario, we would like to predict the duration of failures as well as their incidence rate.

10.4 Models of dependent failure behavior in trajectories

Just because two versions fail on the same input, they do not necessarily fail at the same point of the trajectory. This is one of the ways that the failure behavior of control software is more complicated than that of batch processed software. So, the concept of simultaneous failures should be extended for control software and models developed to describe it.

10.5 Modelling the effect of common testing

The three implementations will be subjected to the same black-box test suite. It will be of interest to try to compare the degree of independence of failure behavior among the three implementations before and after the common black-box testing. Models should be developed to describe the phenomenon.

10.6 Predicting Reliability from Metrics and Prior Information

It is always a hope that reliability can be predicted from prior information before any random testing is done.

10.7 Exploiting Physical Continuity

Control software has a certain continuous physical aspect to it. The question arises as to whether this continuity can be factored into any reliability analysis of the control software. Parnas makes a big point that software testing is much more difficult than testing a physical structure because software is discrete instead of continuous. In this experiment with control software can we incorporate any of the continuity of the underlying physical system into the reliability analysis?

11. Estimating Reliability from Non-Random Testing

The relationship between non-random testing and reliability estimation is probably the most important missing link in software engineering. Virtually all testing methods are non-random. (The methods that incorporate randomness into test selection do not use a real world usage distribution from which direct reliability estimation is possible.) The most important measure of software quality for critical software is certainly reliability. Testing improves the reliability, but unless a quantitative link is found we do not know how reliable the software is and thus whether it is acceptable. Random testing can give estimates of moderate reliability but it is not viable for high reliability software. So we use engineering judgement instead of quantification and measurement. So, even if it is a long shot, we should keep our eyes open for any relationships between non-random testing and the resulting reliability of the software.

It's a dumb idea, but we might as well try fitting reliability growth models to the versions produced in the DO178A verification stage. Some insight might be gained. Furthermore, much of the reliabiity growth modelling done in industry is performed on failure data derived from non-random testing.

Also, the test cases in the DO178A verification stage should be examined relative to the frame input distribution. There might be a relation to observe between the quality of the test suite and how representative it is of the frame input distribution.

12. Experimental Plan for the Simulations

The beginning of an experimental plan can be laid out.

12.1 Preliminary Reconnoiter

It is recommended to get an idea of the potential size of the simulation experiment. Knowing the level of effort required will help plan the specific experimental activities that can and should be undertaken.

The amount of simulation time required per trajectory should be estimated to obtain a rough idea of how many trajectories could be generated in a month (say) of simulation time. Also, a rough estimate of the reliabilty of the final DO178A versions should be obtained; this estimate can give a rough idea of the number of replicates required for failure rate estimation. (Recall, that to estimate a failure probability p with a relative error of 10%, we require 100/p replicates.) These two pieces of information can give a rough prediction of how much simulation time is required to perform various statistical experiments.

12.2 Determine Mission Input Distribution

The exact mission input distribution is not described in the GCS documentation. It must be pinned down.

12.3 Experimental Investigation of Drift

An experiment should be run to see how big a problem drift is. Use the best versions of the three implementations (plus Venus?) running separately from the same mission input test cases. See if the implementations drift away from each other. For a given input case, this can be determined roughly by looking at the final positions of the vehicle under each implementation. For cases where there is a discrepancy, determine whether it is caused by drift or by an error in one of the implementations; it may be necessary to resort to back-to-back testing to make this distinction. For many test cases, look at the dispersion of the final positions that is attributed to drift. If this dispersion is small, then drift can be declared a non-problem. Furthermore, this measure of the dispersion of outputs due to drift might be used as a failure indicator later in the experimental study. If drift is not a problem, then abandonment of the triad design should be considered for the sake of reduced simulation execution time.

12.4 Reliability Estimation of Final DO178A Versions

The most important quantity to estimate is the reliabiity of the final versions in the DO178A development process. The reliability of these versions might effect the decision of what to do next: focus on the estimation of reliabiity growth during the DO178A development process, focus on replicated random debugging runs starting with the final versions from the DO178A process, or both.

12.5 Reliabiity Growth During the DO178A Verification Process

Estimate the reliability growth during the verification process. Estimate the size of the bugs removed by different verification methods.

12.6 Replicated Random Testing of Final DO178A Versions

Look for bugs that were missed by DO178A. Estimate their failure probabilities. Try to determine why they were missed by the DO178A process.

12.7 Sequential Strategy for the Remainder of the Experiment

The rest of the simulation experiment must be planned sequentially depending on what is seen in the first steps of the simulation experiment.

Appendix. Concepts and Definitions

Previous controlled software failure experimentation was done with application software that was executed in batch mode where successive inputs were independent. The current experiment investigates real-time control software for a transient application. This is a much more complicated experimental situation. Various reliability and failure concepts and terminology that were clear and unambiguous in the earlier (independent batch processing) experiments require additional clarification in the context of transient real-time control software. Concepts may be undefined, have multiple definitions, or may need new definitions. So it is worthwhile to list, define and discuss the concepts encountered in this controlled experimentation on transient real-time guidance and control software (GCS).

Anomaly:

Software output that deviates from the specification.

Back-to-back testing:

Running two or more software implementations of a specification on the same input and checking the outputs for agreement. If the outputs do not agree, one of the implementations has failed on this input. If the outputs being compared are real-valued, determining agreement may be difficult or even non-defined.

Batch processing:

Execution of software on a sequence of independent inputs. Processing of successive inputs is done independently. The prior state of the machine is irrelevant, in stark contrast to real-time control software in which successive inputs are dependent and output depends on previous states of the machine. In addition feedback usually exists in control software. Previous controlled replicated-run software experiments used batch processing.

Crash Landing:

Not necessarily an incorrect result according to the specification.

Correct output:

For the GCS experiment the output is consistent with the software specification.

Correct trajectory:

A trajectory (starting from a given initial condition) calculated within the specification. The GCS specification is board enough in terms of numerical integration algorithms that it is conceivable that different software implementations might correctly calculate trajectories from the same initial conditions that drift apart quite a bit. So the concept of "correct trajectory" is not precise. Presumably there is a theoretically correct trajectory from a given set of initial conditions that could be calculated using an algorithm of infinite precision, but that is not a useful definition.

Diad:

Two software implementations running back-to-back with one in the primary driving role and the second in the secondary redundant role for comparison and error detection.

DO178A:

RTCA document giving guidelines for the development of avionics software.

Drift:

Over several frames, two single-version implementations executing independently but from the identical initial condition could correctly calculate trajectories that gradually move apart. This separation can be called "drift" due to imprecise in real-valued calculations. Both trajectories would be correct according to the specification, but it is conceivable that one trajectory could end with a crash or abort and the other with a successful landing.

Duration of error:

If a failure occurs in real-time control software, the software may be able to continue executing and the error may persist for several frames. The number of frames during which the fault caused incorrect output is the "duration of the error." While the error manifests itself, the trajectory may be so perturbed that the remainder of the trajectory is statistically meaningless as far as reliability calculations are concerned. It is not clear that "duration of error" has a useful statistical meaning for single-version software because feedback has caused the system to go off on the wrong trajectory. For n-version software, the system may stay on the correct trajectory even though one of the components has an error; in this case the statistics of "duration of error" are well-defined and of importance for system reliability. This is connected to the phenomenon of "error bursts."

Error bursts:

Clusters of errors on a single trajectory caused by one or more faults. An important phenomenon for n-version software.

Error crystal:

A subset of input space or a subset of trajectory space in which a software version fails. The implication is that the subset is localized or connected.

Error severity:

Classification of effects of different errors. Instead of just talking about success or failure and their probabilities, it may be useful give probabilities for these different classes.

Experimental design:

A description of a plan to collect data.

Failure:

The software fails to perform according to specification.

Failure detection:

A method or act of finding instances where the software does not execute according to specification. Back-to-back testing is a convenient, if imperfect, method.

Failure rate of fault:

The "failure rate of a fault" cannot be defined exactly because it generally depends on the software version. The same faulty code could fail at different rates in different versions

because of fault interaction. It might be a reasonable to ignore this problem and to assume the failure rate just depends on the fault (and the input distribution).

Failure rate of version (general):

The probability that the given version of a particular implementation fails for a random input, chosen from the appropriate input distribution. The failure may be for a mission, phase, frame or sub-frame, in which cases the random input is chosen from the corresponding distribution.

Fault:

Code in the software whose execution gives results that do not agree with the specification.

Fault category:

Faults can be classified according to categories and then reliability statistics calculated respectively.

Fault interaction:

Faults can mask or compensate one another. This means that the failure rate of a fault is changed by the presence or absence of other faults.

Feedback:

The output of one frame effects the input to the next frame. The presence of feedback in control software greatly complicates some basic reliability concepts such as "input distribution."

Fly-through:

A trajectory passes through a set of frame inputs for which it fails and then resumes correct (according to specification) execution.

Frame:

The "frame" is the basic time-interval that the control software iterates on. The software does not necessarily perform an iteration of all the control functions during each frame, only the major ones. Other control functions may be executed during a fraction of the frames.

Frame inputs:

Values of planetary parameters and initial positions and velocities at the beginning of a frame. These values are unknown exactly and therefore assumed to be random with a certain distribution.

Frame input distribution:

Probability distribution of frame inputs. Strictly speaking this is a family of distributions; the distribution depends on which phase of the trajectory the frame is in. It also depends on where in the phase because of the transient nature of the application. A single frame input distribution can be obtained by aggregating over all the frames in a phase.

Frame reliability:

Probability that the code performs according to specification over a given frame of the mission. Reliability of different frames will be different because of the transient nature of the application. But we could aggregate over all frames of a given phase. The reliability will depend on the probability distribution of system state at the beginning of the frame. There is some ambiguity in defining "frame reliability" in control software with feedback: it should be defined as the proportion of "valid" frames that are correctly executed. However, a failure could send the trajectory into frames that are otherwise impossible; such frames should not enter into the definition of frame reliability. "Frame reliability" may not be a fruitful concept.

Functionality:

A piece of software performs multiple functions. Its performance or reliability might be broken down by function.

Gold implementation:

An implementation in which there is a high degree of confidence of high reliability.

Imperfect fix:

When a failure occurs, the fault identified and a correction made, it may be imperfect. There are two main ways this can happen: The extent of the fault may not be appreciated and only part of it is removed. The second type of imperfection is that the fix may actually introduce new faults. This second type of imperfect fix will play havoc with replicated-run experiments.

Implementation:

An "implementation" is computer software which attempts to fulfill the requirements of the GCS specification. Separate teams of software programmers develop separate implementations. Three implementations developed under DO178A requirements are named: Pluto, Earth, and Mercury. The development of these three versions was not done in total isolation: a common test suite was used during verification. Another implementation was developed by Bernice Becker: it is called Venus.

Importance sampling:

Inputs are sampled from the input distribution in a biased way. The bias is introduced to make it more likely to get inputs that will cause the software to fail, thus giving more failure information. The biasing is done in such a way that it can be removed in the reliability estimation calculations. The goal is to get unbiased reliability estimates that have smaller variation (or estimation error) than straight-forward unbiased sampling from the input distribution.

Independent failure behavior:

The failure events associated with separate faults or implementations are independent events.

Inputs:

"Inputs" are the values of various parameters and variables given to the software. The software bases its calculations on these values. The "inputs" might be for sub-frame, frame,

phase, or the entire trajectory. There are two main types of "inputs." One type of input describes the general environment such as gravitation, atmospheric conditions, or optimal drop height; these inputs are constant throughout a single mission but are treated as random because knowledge of the planet is imperfect. The second type of input describes the state of the vehicle (position, velocity, engine state, etc.) and change throughout a mission; at the start of a given mission, phase, frame or subframe they are considered to be random for the purposes of reliability estimation.

Metrics:

Any quantitative descriptions of the software or its observable characteristics, or its development process or environment.

Mission:

The "mission" consists of taking control of the planetary lander at an initial point during the parachute descent above the planet and guiding it along a trajectory until the vehicle lands on the planet or until some other aborting event occurs.

Mission abort:

The trajectory is terminated with some condition other than crash landing or successful landing.

Mission failure:

The software failed to execute according to specification at some point in the trajectory.

Mission failure rate (of version):

The probability that the given version of a particular implementation fails for a random input, chosen from the mission input distribution. Failure means that the software violates the specification at some point in the trajectory. This definition does not distinguish among severity or consequences of the anomalous behavior.

Mission inputs:

Values of planetary parameters and initial positions and velocities at the beginning of mission. These values are unknown exactly and therefore assumed to be random with a certain distribution.

Mission input distribution:

Probability distribution of mission inputs.

Mission reliability (of version):

Probability of mission success, i.e., the probability that the code performs according to specification for an entire trajectory. The probability depends on the mission input distribution.

Mission success:

The software executed according to specification over the entire trajectory.

N-Version implementation:

Replicated software with a voter.

Non-random testing:

Test cases are chosen by a tester in some way different from random sampling from the usage distribution (or input distribution).

Parameter ranges:

The specification states that most parameters are assumed to fall within certain ranges. For parameters like gravitation constants and drop height, this means that the usage distribution is over this range.

Perfect recovery:

We define "perfect recovery" from a fault if the version executes incorrectly for some frames, then resumes correct execution and the resumed trajectory is identical to the trajectory that the version would have followed if the fault was not present.

Perturbation of trajectory:

For a given initial condition, there is defined one or more correct trajectories (depending on the tolerances in the numerical integration routines). While on this trajectory an error may occur that throws the lander off the correct trajectory. The new trajectory is a "perturbation of the correct trajectory." There appears no way to tell if this new trajectory is part of a correct trajectory for some other initial condition; it seems unlikely that it would be. So evaluation of reliability along this perturbed trajectory may be hard to interpret.

In another context, the experimenter might perturb a trajectory in order to see how the software performs in the neighborhood of the original trajectory. The perturbation could result from perturbing the initial conditions, or it could result from perturbing the frame input values in the middle of the trajectory.

Phase:

The trajectory is divided into a sequence of 4 successive phases: i. initial parachute descent; ii. engine warm-up with parachute attached; iii. controlled flight with parachute released down to drop-height; iv. non-powered drop from drop-height to touch-down. There are different activities and requirements during the different stages. This is part of the transient nature of the application that create a number of various reliability concepts.

Phase reliability:

Probability that the code performs according to specification over a given phase of the mission. Reliability of different phases will be different. The reliability will depend on the probability distribution of system state at the beginning of the phase.

Primary version in triad:

The version in a triad of back-to-back executing versions whose output from one frame is used by all three versions as input to the next frame.

Random testing:

Test cases are chosen randomly from the usage distribution.

Real-time control processing:

Iterative execution of successive inputs with memory and feedback.

Recovery:

A version fails for some frames in a trajectory and then resumes correct execution according to the specifcation. The trajectory may be altered from what it would be if more failure had occurred.

Regression testing:

Re-testing with earlier test cases after later changes are made to software. This catches imperfect fixes that added new faults and may catch bug interactions. It helps assure that successive versions are strict improvements.

Reliability:

The usual definition of "reliability" is the "probability of successful completion of a task." The task might be a single simple application at a single point in time. The task might be a complicated mission with multiple objectives that lasts for a long duration. The concept of reliability is multi-faceted in the GCS experiment because there are many different "tasks" of interest and because "successful completion" may not be clearly defined, or may have more than one definition. Finally "probability" will depend on the random situation in which the software is executing. So we must clearly define the particular reliabilities of interest: e.g., mission reliability, frame reliability, etc.

Reliability estimation:

Calculation of failure probabilities from failure data.

Reliability growth:

Improvement of reliability in successive versions as bugs are removed. There are mathematical models of reliability growth but they strictly valid when the bugs were discovered by random testing. These models apply to the random testing part of the GCS experiment but not to the verification process that is based on non-random testing.

Reliability model:

Mathematical relations between reliability of a system and other variables.

Replicated-run software experiments:

Phyllis Nagel performed the first replicated-run software failure experiments. She had multiple implementations of a single application. She assumed a usage distribution and drew a succession of independent inputs from that distribution. She executed an implementation (using this sequence of inputs) until it failed; at that point the fault causing the failure was removed and execution continued, faults be removed as they occurred. She replicated the debugging process: for an implementation whose original version had n faults, she allowed for the possibility of having 2^n versions. With this many versions, it was possible to estimate failure rates of individual bugs, and to observe that bug failure rates might depend on what other bugs were present in the version. In the current GCS experiment, the version coming out of the verification process can be subjected to replicated-run random testing. It is not clear that it is desirable to generate multiple debugged versions from this random testing arising from the random order of bug appearance in the different replicates. The verification process consists of non-random testing and is replicated only once; so it falls outside the realm of replicated-run random-testing and debugging.

Safe Landing:

Physical event. Not necessarily a correct result according to the specification.

Secondary versions in triad:

Versions in a triad used for output comparisons and error detection.

Sequential estimation procedure:

Taking sample observations one at a time until a desired level of precision is obtained in the statistical inference, and then stopping sampling.

Sequential experimental design:

Determining future experiments after consdiering the results of previous experiments.

Single-version software:

One replicate or implementation running by itself. For control software with feedback this means that errors in the software can change later frame inputs or cause incorrect frame inputs.

State of machine:

All internal values that can effect the output of a software execution.

Subframe:

Each frame is subdivided into three "subframes." This is a way of subdividing the functionality: different functions are performed in the different subframes. It seems to be mainly for the benefit of defining an interface between the application software and the simulator.

Subframe reliability:

Probability that the code performs according to specification over a given subframe of the mission. Reliability of different frames will depend on the type of subframe (there are three) and the type of phase (there are four) and will vary over the trajectory because of the transient nature of the application. But we could aggregate over all subframes frames of a given phase, removing this transient effect. The reliability will depend on the probability distribution of system state at the beginning of the subframe in the phase.

Success:

The software performs according to specification. The GCS experiment adopts this convention. Thus if the planetary lander successfully lands or crashes is not equivalent to "success" and "failure" of the software, respectively.

Triad:

Three versions running back-to-back in parallel. The output during a frame of one version (designated as the primary version) is used as input by all three versions in the next frame.

Tolerance:

It is necessary to raise the concept of "tolerance" when performing calculations with real-valued variables. Different values of the real-valued variables can satisfy the specification. Two

different algorithms might given slightly different answers, both of which are correct. In this case the specification should address the issue of how accurate the output should be or what the correct range is. The GCS gives this "tolerance" implicitly by specifying acceptable numerical integration algorithms in one instance (for example 4th order Runge-Kutta). This creates a verification problem: instead of checking that the output is the correct value within acceptable tolerance for given input values, it is necessary to verify that the correct algorithm is used and properly implemented. This may create a fuzzy region between correct and incorrect output. Even for an entire mission the GCS experiment criterion for correctness is agreement with the specification, not successful landing (an easily verifiable condition.) In determining correctness at the frame level it is even more problematic. If two implementations give different output for a single frame input, we do not have a "tolerance" given in the specification with which to compare this difference.

Trajectory:

The "trajectory" is the sequence of states of the system as it makes its terminal descent onto the planetary surface.

Transient behavior:

System behavior that is not time homogeneous.

Usage distribution (general):

The guidance and control software (GCS) is required to perform for various situations. It must process the "inputs" provided according to the specification. The inputs can take different values; there is uncertainty or randomness regarding the values of the inputs. This uncertainty is described by a probability distribution called the "usage distribution."

Variable ranges:

Acceptable values for variables according to the specification.

Verification process:

During the development of a software implementation, a sequence of activities is performed to remove faults from the software: design review, compile check, code review, unit test (one black-box test suite for all implementations, plus individual supplemental white-box testing to achieve a coverage requirement), subframe test, frame test, top-level simulator integration test, replicated runs using random test case generation based on real-world usage distribution. The verification process is done once for each implementation. It is not replicated. The verification process yields a sequence of versions of the given implementation. Successive versions may differ by the removal of a single fault or by removal of multiple faults. These versions are available for random testing in the statistical simulation part of GCS experiment.

Verified version:

The final version of an implementation that has completed all the verification activity. In the GCS experiment this version is the initial version undergoing random testing.

Version:

A given software implementation goes through many "versions" as faults are removed during verification. A version could be referred to by the implementation name or letter, plus

some alphanumeric identifier to indicate the version. For example, for implementation "A", the version to pass a compile test might be referred to as "(A,0)" and the nth version after that as "(A,n)". Or the verification stage might be explicitly indicated: for example "(A,f,3)" could represent the third version created during "frame testing." The verified version delivered to the random testing stage at the end of the verification process could be denoted as "(A,v)". One aspect of the statistical part of the GCS experiment focuses on estimating the reliabilities of different versions and estimating the differences in reliabilities of successive versions of an implementation.