NASA Contractor Report 181667

ICASE REPORT NO. 88-31

ICASE

AUTOMATICALLY GENERATED ACCEPTANCE TEST:
A SOFTWARE RELIABILITY EXPERIMENT

Peter W. Protzel

Contract No. NAS1-18107
May 1988

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Automatically Generated Acceptance Test: A Software Reliability Experiment

Peter W. Protzel *

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA 23665

Abstract

This study presents results of a software reliability experiment that investigates the feasibility of a new error detection method. The method can be used as an acceptance test and is solely based on empirical data about the behavior of internal states of a program. The experimental design uses the existing environment of a multi-version experiment previously conducted at the NASA Langley Research Center, in which the 'launch interceptor' problem is used as a model problem. This allows the controlled experimental investigation of versions with well-known single and multiple faults, and the availability of an oracle permits the determination of the error detection performance of the test. Fault-interaction phenomena are observed that have an amplifying effect on the number of error occurrences. Preliminary results indicate that all faults examined so far are detected by the acceptance test. This shows promise for further investigations, and for the employment of this test method in other applications.

*This research was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-18107 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

1. Introduction

The problem of dealing with software faults has become one of the most critical issues in the design of systems that have to meet very high reliability requirements, especially in life-critical applications. This is also reflected by the increasing cost of software testing and maintenance, which has been estimated to consume 70% of the overall software development effort [1]. Thus, the development of effective error detection techniques plays a major role in the effort to reduce the cost and to increase the reliability of software. A comprehensive overview about research activities and results in the area of validation, verification, and testing strategies can be found in [2] and in a more recent survey by Morell [3], which includes an extensive annotated bibliography.

Although many different approaches to software testing and error detection have been proposed in the last years, the basic problem is far from being solved. There exists no unified methodology yet, and each strategy has its own advantages. A recent empirical study by Basili and Selby [4] even indicates that code reading by stepwise abstraction is at least as effective as on-line functional and structural testing in terms of number and cost of faults observed; however, it is also stated that the direct extrapolation of these findings to other testing environments is not implied. But in general it is expected that the production and testing of complex and highly reliable software will require a high degree of automation of the testing and verification strategies in order to eliminate as much subjectiveness and human intervention as is practical [5,6].

This paper addresses the problem of error detection by studying the feasibility of an acceptance test solely based on empirical data about the behavior of internal states of a program. The following section describes the basic idea of the proposed method and discusses implementation and application issues. In order to study the effectiveness and error detection capability of this approach, an experiment was originated by using the existing environment of a previously performed multi-version experiment at the NASA Langley Research Center. Section 3 describes the experimental design, which uses the 'launch interceptor' problem as a model problem, and section 4 shows the empirical results obtained so far. Another aspect of the experiment is the investigation and observation of fault interaction phenomena, and how it might affect the error detection performance. These results are also given in section 4, followed by a general discussion in section 5. In section 6, the results are summarized and conclusions drawn.

2. Acceptance Test Based on Empirical Data

The starting point of this approach is the definition of several test-bits within the application software; for example, these test-bits might be derived from certain intermediate results or flags. The test-bits are combined into a pattern which reflects the internal state of the program during its execution. Empirical data about this pattern is collected from the execution of an oracle by storing the values and the frequency of the test-bit patterns in the form of a distribution. This distribution contains information about the correct program behavior and can be used as an acceptance test for the application program, in which for each run the observed pattern is compared to the data stored in the distribution. If the observed pattern has never occurred before, i.e. is not listed in the distribution, then this corresponds either to a software error or to a correct, but very unusual, new event. In either case, a 'warning' message should be produced as an indicator of a dubious output or appropriate error handling measures should be taken.

But there is also the possibility that a software error produces a 'known' pattern that is listed in the distribution and has occurred under different input conditions as a 'correct' pattern during the test run of the oracle. This is possible because the test-bit pattern can be seen as the product of mapping the (usually infinite) input space onto a finite set of numbers. Therefore, the input space contains an infinite number of points that produce the same bit pattern. Since such an error would not have been detected by comparing the pattern with the distribution, the frequency of these events determine the error detection performance. Unfortunately, there is no general analytical way to predict this performance because of the complexity of the problem and the dependency on the application, but this is the case for almost all error detection techniques.

One problem associated with this approach is the availability of an 'oracle', that is an external mechanism that produces the correct results for a given set of inputs. In this context, it is important to distinguish between a *true oracle* and a *pseudo-oracle* [7]. If a true oracle is available, which produces all required outputs always correctly ***and*** within the required time, then there is clearly no need for any testing, and such an oracle can simply be used as the application program; but this is hardly ever the case. However, there is a certain class of applications where a pseudo-oracle is available, which might be less efficient than required for the final application, or less complete and able to produce only certain parts of the required output.

Examples for the implementation of a pseudo-oracle are the use of a very high level language [7] or executable specification languages [6], which are less likely to be incorrect than the application program, but cannot be used for an on-line comparison to the output of the application program in real time due to the inefficiency of the compiled object code. Another important class of applications is the area of parallel programming, in which it is quite common that a sequential program version is written before implementing the parallel version in order to check and verify the used algorithms. Then the sequential program can be used as a pseudo-oracle for parallel debugging, but with the same difficulty of an on-line comparison in real time. An *incomplete* pseudo-oracle might be available when analytical methods or formal proof techniques allow one to predetermine only partial aspects of the software behavior, for example, the values or the range of values of certain internal states of the program. Wild suggests e.g. in [6] the use of algebraic specification of abstract data types to generate assertions about the general behavior of the system.

The additional effort to develop a pseudo-oracle is especially justified or even required in life-critical applications and other areas where high reliability requirements have to be met. If a pseudo-oracle is available, it can, of course, be used to generate specific test cases for an off-line testing of the application program, and this should always be done to test, for example, the handling of exceptional cases by employing special or extremal values. But in order to develop an acceptance test that gives an (on-line) error indication after each run, it is necessary to make some assertions about the correct software behavior, and to use this information for judging whether the current behavior is acceptable.

In contrast to the classical error detection and program instrumentation techniques [8,9,10], the proposed method derives these assertions solely from empirical observations about the behavior of a pseudo-oracle. Thus, it is not necessary to have any a priori knowledge about the correct behavior, which is the major difficulty in implementing standard instrumentation techniques. In order to get sufficient information about the program behavior, a representative set of test cases has to be provided for the run of the pseudo-oracle. One way to automate this process is the use of randomly generated input data according to the expected usage distribution. Although random testing is a controversial issue, a comparative study by Duran and Ntafos [11] shows its cost effectiveness for many programs with the application of only 25 to 120 random test

cases. The use of random testing seems reasonable in the context of the presented approach, since it is based on statistical data about the program behavior, and the availability of a pseudo-oracle allows the automatic generation of a large number of test cases in the order of 10^4 to 10^6.

The next section describes the 'launch interceptor' program, which is used as a model problem for the implementation of the described test method, and the design of the experiment which is performed to study the feasibility of this approach and to get statistical data about the error detection performance.

3. Experimental Design

The 'launch interceptor' (LIC) problem was chosen as an application because it is a well-known model problem, which has been used in various other experiments for software reliability estimation and modeling [12,13,14], and for evaluation of the independence assumption in multi-version programming [15]. This experiment uses four different program versions from a previous software reliability study carried out by the Research Triangle Institute (RTI) for the NASA Langley Research Center [13,14]. This allows a controlled experimental investigation since there is considerable confidence that all the original faults in the different versions have been identified during extensive testing by the previous study, and these versions can now be instrumented with known single or multiple faults. Furthermore, there is another empirical study that has used the LIC problem to evaluate the error detection performance of standard self-check techniques [16], and it is interesting to compare these results to those of the present experiment.

The launch interceptor program can be seen as part of a hypothetical antiballistic missile system that reads simulated radar tracking information and has to decide whether the radar reflections represent the trajectory of a missile. This is done by using 15 different geometrical conditions, called launch interceptor conditions (LICs). If a missile is detected, a signal to launch an interceptor has to be generated. Figure 1 shows a schematic representation of the launch interceptor program.

For each run, the program reads a random number of up to 100 two-dimensional points as its input, which are randomly generated according to a specified usage distribution. The program determines which of the 15 LICs is met for the given set of input points. For example, LIC 1 is met if there exists at least one set of two consecutive

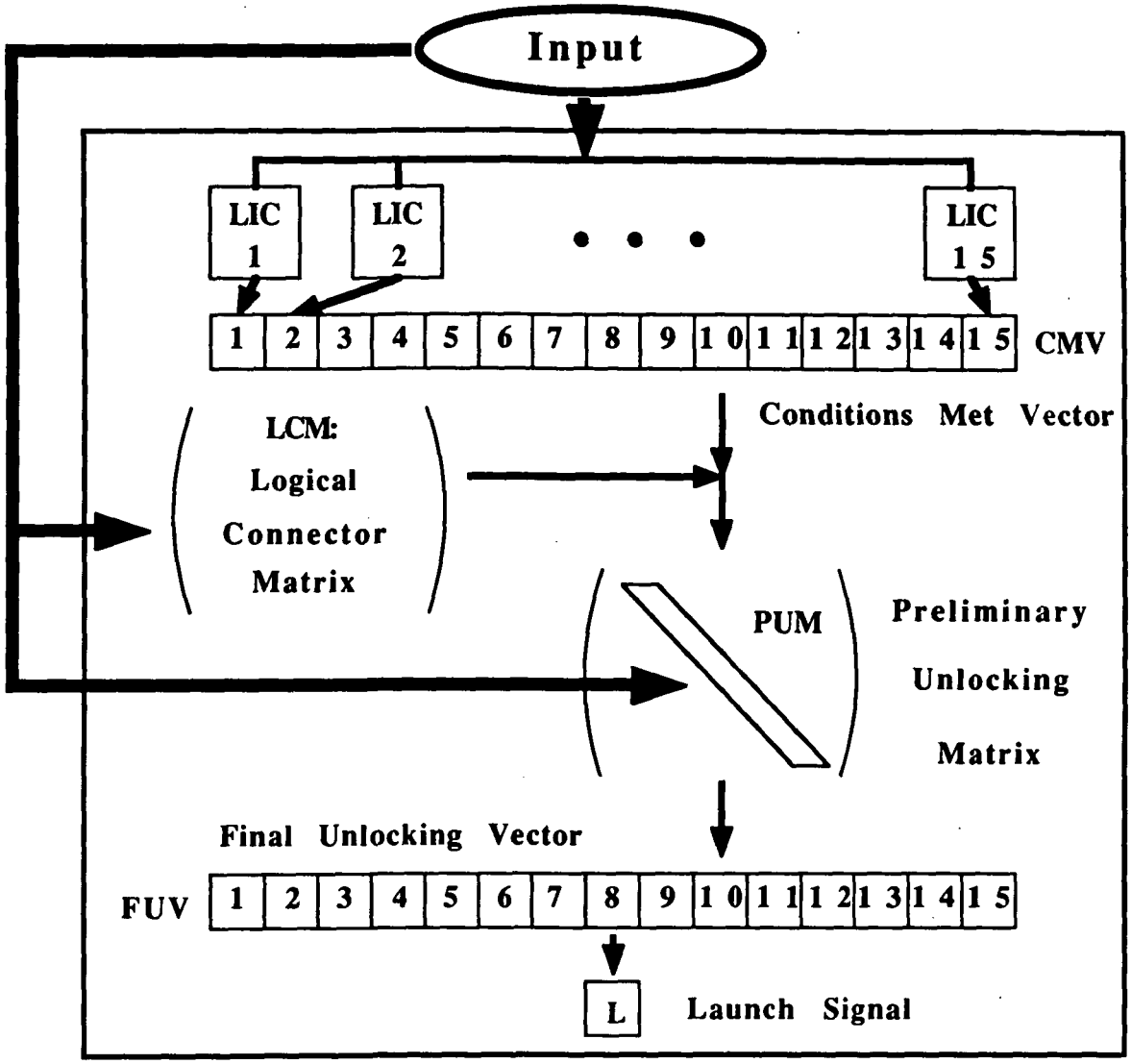

Figure 1: Schematic representation of the launch interceptor program.

points that are a distance greater than a given length l apart. If this is the case, the first bit of the Conditions Met Vector (CMV) is set to '1', otherwise it remains at '0'. The length l is a parameter for LIC 1, and there are other parameters for the various LICs that are given as additional input. The Logical Connector Matrix (LCM) and the 15 diagonal elements of the Preliminary Unlocking Matrix (PUM) are further input information that dictate how to combine the outputs of the 15 LICs and which of them

are relevant for the current set of input points. By using this information and the results of the CMV, the elements of the Final Unlocking Vector (FUV) are computed. The final launch bit is set to '1' to signal an interceptor launch if and only if all 15 bits of the FUV are set to '1'.

Thus, the launch bit is the only true output of the program, while the CMV, the FUV, and the non-diagonal elements of the PUM can be seen as intermediate results of the computation. The values of the parameters used for the determination of the 15 LICs are the original values of the previous experiment [13], and are not changed during the test runs. Some of the LCM-elements are also constants and some are randomly generated for each run with a given probability distribution. All of the 15 binary diagonal elements of the PUM are randomly generated for each run. More detailed information about the values and the generation of the parameters can be found in [13], which also contains the complete program specification.

The binary form of the intermediate results in the LIC program makes it easier to select appropriate test-bits for the proposed acceptance test, but similar situations can be found in many programs, for example in the area of digital control problems. For this application, the 15 bits of the CMV and additionally the final launch bit have been selected as test-bits. They are combined to a pattern in the order of the CMV enumeration with the first bit of the CMV (corresponding to LIC 1) as the most significant bit and the launch bit as the 16th and least significant bit. The order of the test-bits is arbitrary in principle, but important to define because the pattern will be referred to in its octal or decimal representation. These 16 bits are selected as test-bits because they represent important intermediate results as well as the final output. The general problem of selecting test-bits will be discussed in section 5.

The definition of some terms used in the following is essential to the interpretation of the results. In this context, it is especially important to distinguish between the terms *fault* and *error*. According to [14], a *fault* is a conceptual flaw in the program. An *error* is seen as the manifestation of a fault under certain input conditions, and results in an undesired state of the program. Thus, a fault is the cause of an error. A fault is *latent* as long as it has not caused any errors, but exists in the program as a potential cause [1]. An error is latent as long as it is not detected by a detection mechanism. In the following, the term *bug* is also used to characterize a software fault. Since the meaning of *bug* and *fault* is identical, both terms are used interchangeably.

Furthermore, a *test case* is denoted as a single program run, in which the program is provided with a complete set of input data and generates the corresponding outputs.

Figure 2 shows the configuration of the program versions for this experiment. After identifying and removing all the bugs, three of the four available versions are employed as oracles and their outputs, that is the launch signal and all intermediate results, are compared by a voter for each test case. The three oracles together with the voter constitute the so called "gold-version". This configuration is chosen to increase the confidence of getting a correct result. Two of the versions are instrumented with known bugs, and all the outputs of all versions are compared to the gold-version. The names of the bugs refer back to the original numbers assigned in the previous experiment, in which, for example, bug 7 and 8 were identified in one version called AT1, and another version (AT3) originally contained bug 2 and 13 [13,14]. These previous publications also give a detailed description of those bugs. In Figure 2 for example, three copies of version AT1 are used, each instrumented with bug 7 only, bug 8 only, or bug 7 and 8, respectively. The run of a given number of test cases is automatically performed by a software package called N-version controller developed at NASA Langley Research Center that generates the input data, executes the versions, and examines the outputs. Detailed information about certain events, like e.g. output disagreements, are stored according to commands specified by the experimenter.

The gold-version of this experimental multi-version environment actually represents a *true oracle.* As discussed in the previous section, this is not necessarily required for the application of the proposed test method, but in the context of the performed experiment, it is required to have a true oracle for the evaluation of the effectiveness of the test method. Thus, it is necessary to detect and to list all error occurrences in order to see if, and how many of, these errors would have been found by applying the acceptance test. But since the test is based on empirical data about the correct program behavior, the first step of the experiment is to gather this information by executing the gold-version for a certain number of test cases. These experimental results are described in the next section.

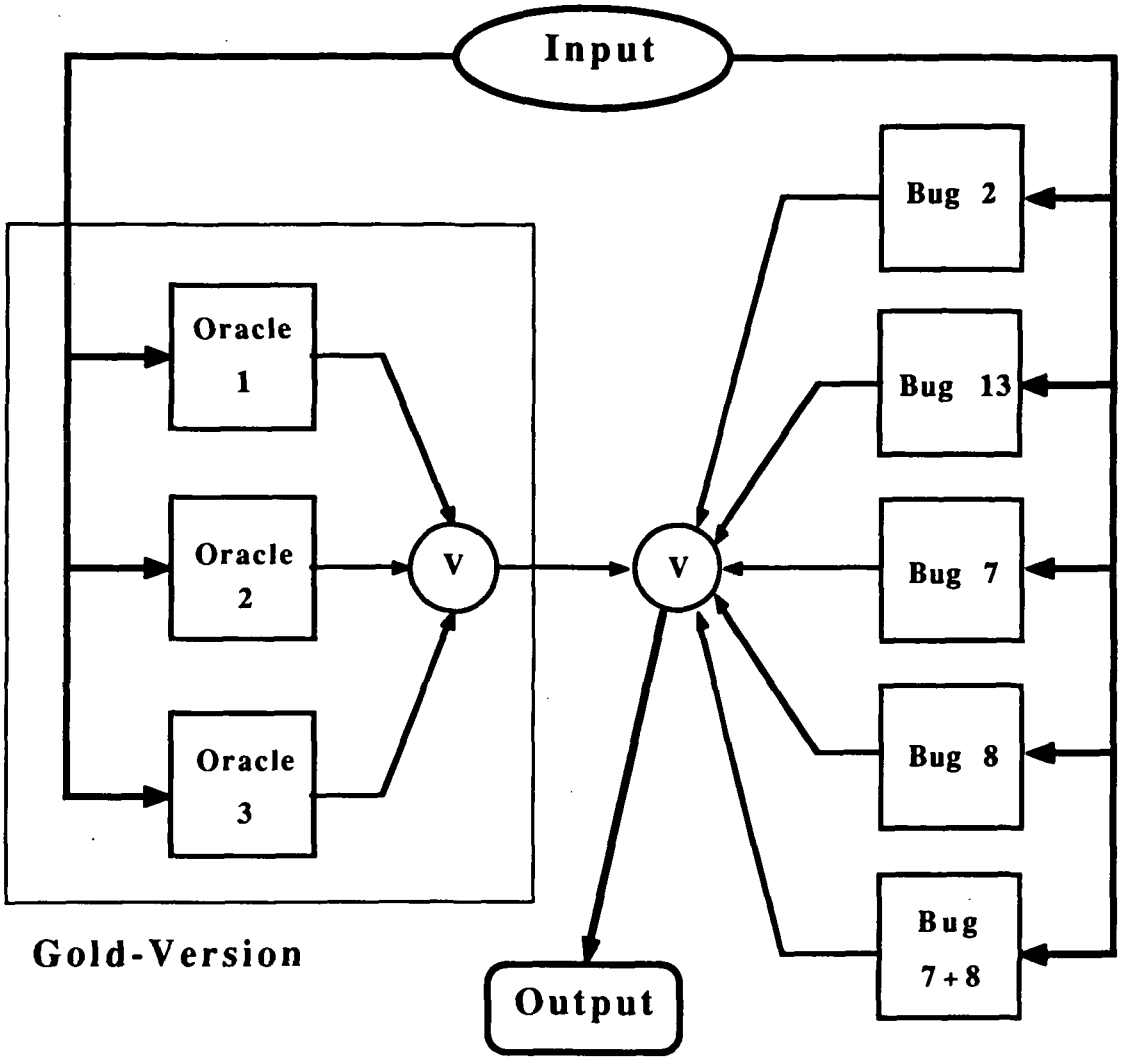

Figure 2: Configuration of the multi-version environment.

4. Results of the Experiment

Since 16 bits are selected as test-bits and combined to a pattern, there are $2^{16} = 65535$ different patterns that can theoretically occur. To obtain the desired information about the program behavior with respect to the test-bit pattern, data about the values and the frequency of the patterns were accumulated by executing the gold-version for various numbers of test cases. Surprisingly, only 208 different patterns out of the 65535 possible patterns occurred after one million test cases. Furthermore, one out of the 10 most

frequent patterns occurred in about 95% of the test cases and there is one special case that occurred in 71% of all test cases. This special pattern with an octal value of '177777' represents the event that all of the 15 launch interceptor conditions were met (all CMV elements set to '1'), and that the launch signal was given (test-bit 16 also set to '1'). Table 1 shows a partial listing of the distribution of patterns, which is called "gold-distribution" because it reflects the correct program behavior and was obtained by executing the gold-version. Figure 3 shows a plot of the gold-distribution with the number of occurrences related to the total number of test cases to get a relative frequency representation.

The shape of the distribution in Figure 3 depends on the (arbitrary) definition of the order of test-bits, but it is interesting to observe the considerable differences in the frequency of different patterns. The frequent occurrence of the special case with all bits set to '1' also means that a launch signal is given in more than 71% of the test cases under random input conditions. Since this is hardly the expected behavior of a real anti-ballistic missile system, the LIC program has to be regarded as a hypothetical model problem. The occurrence of only 208 different patterns in one million test cases raises the question how the number of different patterns depends on the number of test cases. Figure 4 shows the observed statistical convergence by executing the gold-version for different numbers of test cases. The statistical convergence can be used to determine the number of test cases necessary to get representative information about the program behavior. Thus, the number of different patterns can be seen as a special kind of 'coverage measure', and provides useful supplementary information about the testing process.

One reason for the occurrence of the small number of different patterns is possible dependencies between test-bits. In fact, there are three obvious dependencies due to the specification of the underlying launch interceptor conditions. For example, the output of LIC 8 (CMV 8) is set to '1' if a certain condition A is met, and CMV 13 is set to '1' if the same condition A *and* another condition B is met [13,15]. Therefore, LIC 13 can never be met if LIC 8 is not met, and this dependency inhibits all pattern combinations with CMV 8='0' and CMV 13='1'. Similar dependencies exist for two other LIC pairs and it can be estimated that these three dependencies inhibit 58% of the possible pattern combinations. But this does not yet explain the observed small number of different patterns. It would be interesting to apply formal proof or analytical

test-bit pattern		number of	test-bit pattern		number of
decimal	octal	occurrences	decimal	octal	occurrences
0	0	4962	.	.	.
64	100	1	65518	177756	181
1024	2000	4706	65519	177757	798
1088	2100	1	65520	177760	1
8192	20000	1359	65522	177762	1
8256	20100	348	65528	177770	9
9216	22000	7434	65529	177771	9
9217	22001	1	65530	177772	6
9280	22100	23517	65531	177773	41
9281	22101	1	65533	177775	2
.	.	.	65535	177777	705731

Table 1: The "Gold-Distribution" after 1 million test cases (partial listing).

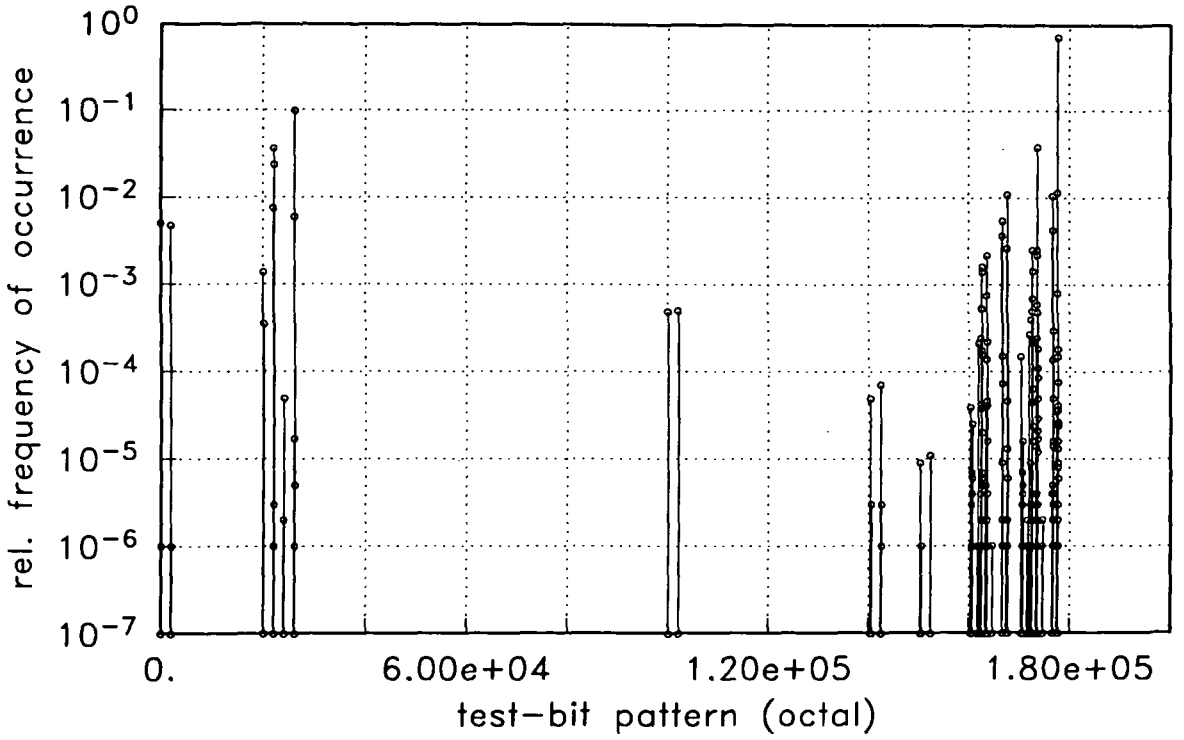

Figure 3: The "Gold-Distribution" after 1 million test cases.

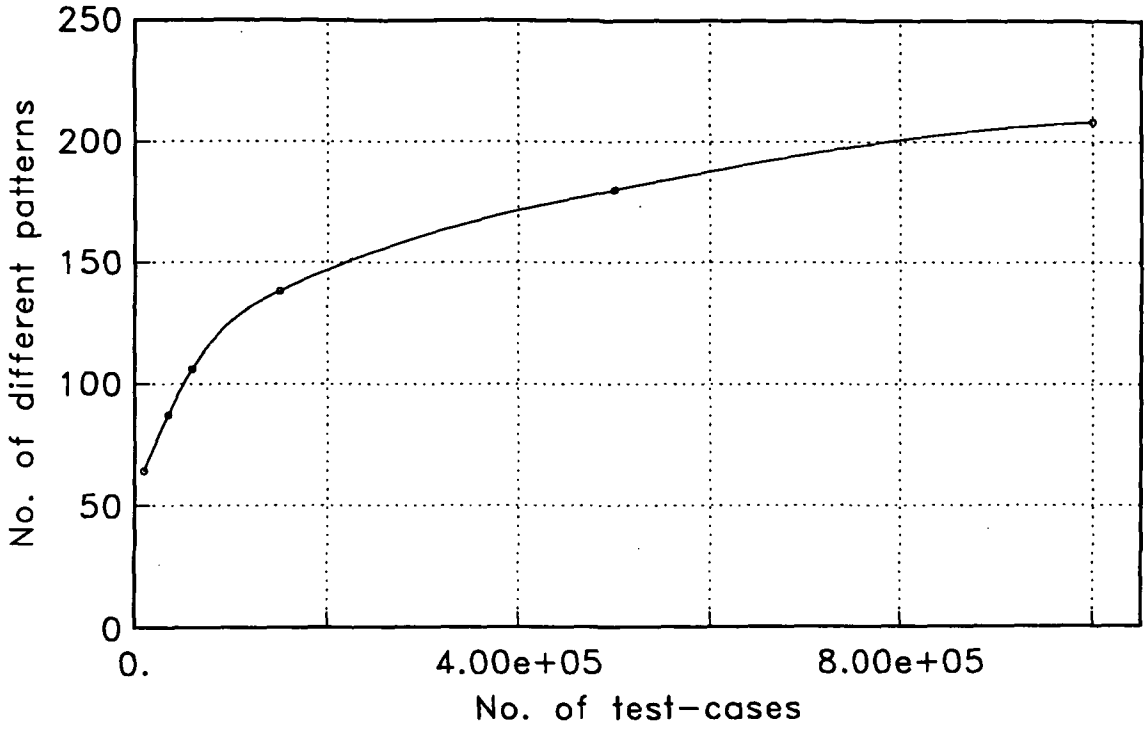

Figure 4: Statistical convergence of the number of different test-bit patterns.

techniques to detect other, possibly more subtle dependencies.

The question in using this gold-distribution as an acceptance test is whether software errors produce one out of the 208 known patterns or a different pattern. This information was obtained by executing the versions that contain different bugs together with the gold-version. The bugs were not removed after an error occurred, but each discrepancy between the outputs was registered and the values of the corresponding test-bit patterns were stored. Table 2 shows examples of these listings for three different bugs after executing 234,988 test cases. By looking at the erroneous pattern (bug-pattern), it can be determined if this pattern is listed in the gold-distribution. If it is not listed, the error would have been detected by the proposed acceptance test.

Additional information about the bug-effects are also listed in Table 2. The comparison of the bug-pattern and the corresponding correct pattern of the gold-version (gold-pattern) shows that bug 2 produces errors by changing the output of LIC 13 from '1' to '0', while bug 13 has the opposite effect. The errors caused by bug 7 affect

bug name	bug-pattern	gold-pattern	number of occurrences	bug-effect	0:detected 1:undetected
bug2	140600	140610	1	LIC13 1 → 0	0
bug2	142700	142710	1	LIC13 1 → 0	0
bug2	160600	160610	2	LIC13 1 → 0	1
bug2	160640	160650	2	LIC13 1 → 0	0
bug2	160700	160710	2	LIC13 1 → 0	0
.	.	.	.	.	.
bug2	177724	177734	16	LIC13 1 → 0	0
bug2	177767	177777	338	LIC13 1 → 0	0
bug13	10	0	1114	LIC13 0 → 1	0
bug13	2010	2000	1124	LIC13 0 → 1	0
bug13	100010	100000	126	LIC13 0 → 1	0
bug13	102010	102000	108	LIC13 0 → 1	0
bug7	0	20000	73	LIC3 1 → 0	1
bug7	0	20100	1	LIC3+10 1 → 0	1
bug7	100	20100	2	LIC3 1 → 0	1
bug7	2000	22000	259	LIC3 1 → 0	1
bug7	2000	22100	5	LIC3+10 1 → 0	1
bug7	2100	22100	22	LIC3 1 → 0	1
.	.	.	.	.	.
bug7	177651	177751	1	LIC10 1 → 0	0
bug7	177657	177757	2	LIC10 1 → 0	0

Table 2: Exemplary listing of error occurrences for several bugs.

the output of two different LICs and there are also cases with a two-bit disagreement, that is, both LIC outputs are changed simultaneously. This analysis of the bug-effects provides valuable information for locating the faults.

As a measure for the error detection performance of the acceptance test, an *error detection rate* is defined as the number of errors detected relative to the total number of errors occurred. Table 3 shows the values of the error detection rate together with a summary of the error statistics. It can be seen that a remarkably high error detection

bug name	number of test cases	number of diff. error patterns	rel. number of error occurrences	error detection rate
bug2	234,988	94	1.4 %	82.0 %
bug13	234,988	4	1.1 %	100.0 %
bug2+13	180,000	92	5.2 %	98.8 %
bug7	234,988	84	0.5 %	8.3 %
bug 8	234,988	67 abort cases		
bug7+8	234,988	like bug7, but additionally 90 abort cases		

Table 3: Summary of the error statistics.

rate was observed in most cases, which is even 100% for errors caused by bug 13. But the relatively small error detection rate for errors caused by bug 7 also demonstrates the different sensitivity of the test for different bugs. For the interpretation of these results, it is essential to distinguish between the *error detection rate* and the *fault detection rate*, which can be defined as the number of detected faults relative to the total number of faults. The error detection rate is an estimator for the conditional probability of detecting an error under the condition that the error occurs. Only if this rate is *zero* for a given number of test cases, that is none of the error occurrences is detected, is the fault not detected. This shows clearly that all the faults investigated so far would have been detected by using this test method. These results show promise for further investigations of different faults, since it seems highly unlikely that errors caused by a specific fault produce *always* one out of the known patterns of a gold-distribution.

Another aspect of this experiment was the investigation of fault-interaction phenomena, and how this might affect the error detection rate. For example, Table 2 shows that bug 2 and bug 13 affect the output of LIC 13 in an opposite way, and one could suspect that this leads to a compensational effect if both bugs are present. In order to study this phenomenon, a version with both bugs (bug2+13) was executed on 180,000 test cases. The result in Table 3 indicates that the presence of both faults has an amplifying effect, because the relative number of error occurrences is considerably higher than the sum of these numbers for the single faults. But despite the high rate of error occurrences, a very high error detection rate of 98.8% was observed, which means that 98.8% of the errors produced patterns not listed in the gold-distribution. The same experiment was made by executing a version with bug 7 and 8, but since

bug 8 produced only abort cases, the results are difficult to quantify. Nevertheless, a similar amplifying effect can be observed because 23 additional abort cases occurred in the presence of both faults.

The test-bit pattern can be regarded as the product of mapping the N-dimensional input space onto a linear array, and it can be used to visualize this fault interaction phenomenon. Figure 5 shows plots of the produced error patterns after executing the version with bug 2 only, bug 13 only, and bug 2 and 13 for 60,000 test cases. It can be seen that the errors of bug 2 and bug 13 produce patterns in different 'regions'. The version with both faults produces similar patterns in those regions, but also additional patterns with a high frequency in other regions. Since the effect shown in Figure 5 represents only one possible fault-interaction phenomenon, further empirical studies are necessary to classify different effects. The use of the test-bit patterns might be helpful for visualizing these effects and to gain more insight in the behavior of programs with multiple faults.

5. Discussion

It is interesting to look at the results of another empirical study that has also used the LIC program to evaluate the fault detection performance of standard self-check techniques [16]. In this study, 8 programs were selected out of 27 different versions from a previous multi-version experiment [15]. Three copies were made from each of the 8 programs and assigned to 24 students who instrumented the programs with self-checking code. The programs were executed using the test cases on which they had failed in the previous experiment along with 20,000 new randomly generated test cases. The students applied overall 865 self-checks in the 24 programs, but only 33 checks were effective in detecting a fault. These effective checks detected only 11 out of 20 previously known faults, but also 6 new faults that were not found by over one million test runs of the previous experiment. Furthermore, 22 new faults were introduced by the students during the instrumentation of the programs.

Although the employment of 24 students does not represent an industrial environment to perform software testing, the results of this study clearly show the difficulties in developing effective self-checks for the considered application. The results also confirm the known problem of introducing new faults by changing the program code.

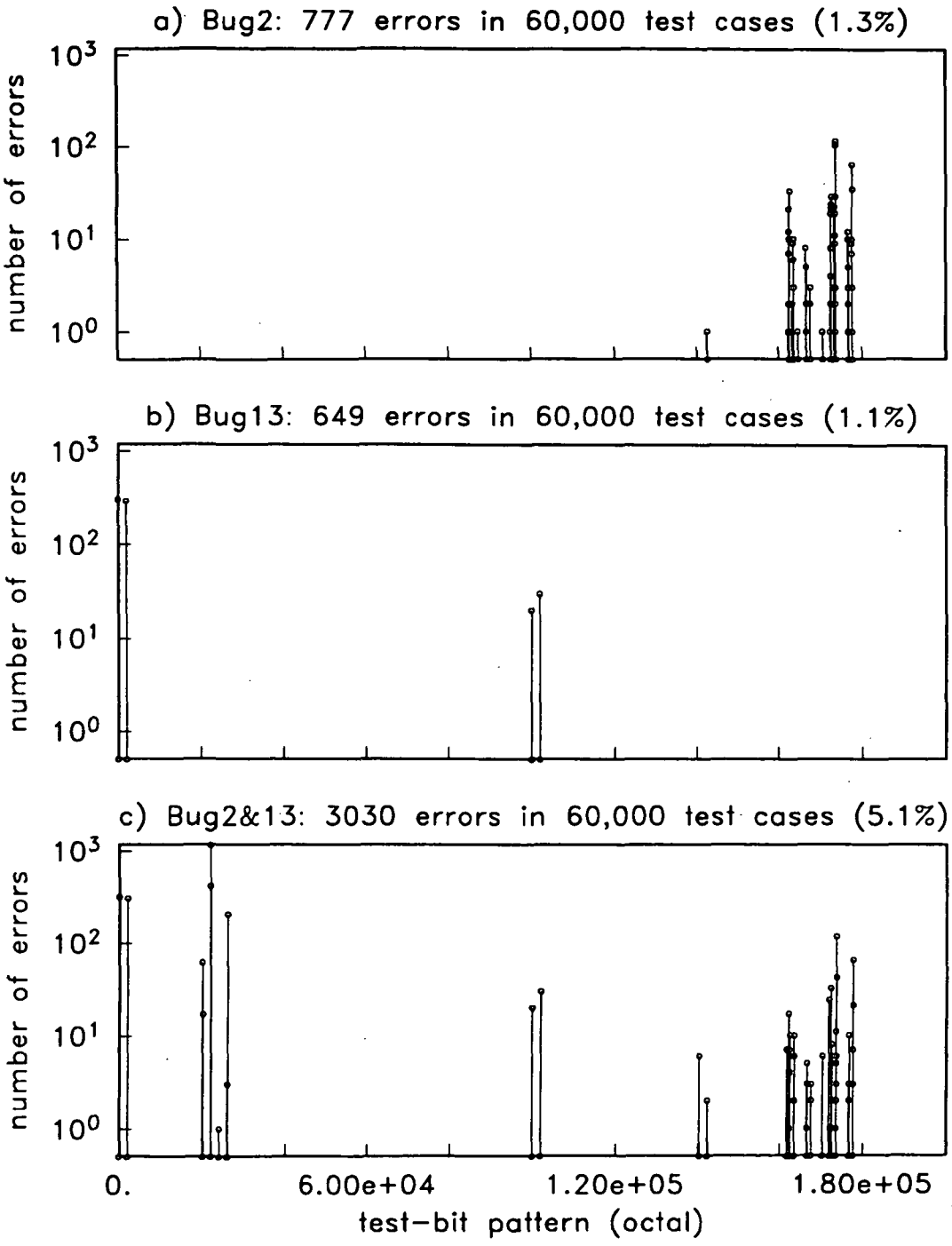

Figure 5: Visualization of fault interaction in form of produced error patterns.

Unfortunately, these results cannot be directly compared to the results of the present experiment, since the program versions and faults are different. But it can be seen that the LIC problem is not a trivial example that allows an easy implementation of standard error detection techniques.

There are several problems associated with the use of the proposed acceptance test that limit the range of possible applications. One problem concerns the availability of a pseudo-oracle, and this has already been discussed in section 2. Another problem concerns the definition or placement of test-bits within the application program. This is basically the same problem that arises in all instrumentation techniques, and the existing methods for locating insertions could also be applied for locating test-bits. If the application mainly uses binary variables, for example in digital control problems, then it is not even necessary to insert additional code, and appropriate test-bits can be identified by using the existing variables. This avoids the problem of introducing new faults by changing the source code. If it is not possible to derive the test-bits from existing variables, the program might be instrumented with assertions that result in an binary output. For example, in [5] executable assertions are used in a different context, but these techniques could be easily combined.

Another aspect is the occurrence of 'false alarms'. Since the gold- distribution does not contain all possible patterns, it is possible that the application program produces a pattern that corresponds to a correct result, but is not listed in the distribution. The frequency of these false alarm cases depends on the number of test cases that are used to produce the gold-distribution, or on the statistical convergence of the number of different patterns, respectively. During the performed test runs, only three false alarm cases were observed, which is not seen as a serious problem compared to events in which an actual error is not detected. Applications that tolerate a relatively high number of false alarms could even use only a subset of the gold-distribution, for example only the 50 most frequent patterns, because this increases the error detection rate.

6. Summary and Conclusion

The goal of this study was to investigate the feasibility of an error detection method that can be used as an acceptance test. The basic idea is to define a test-bit pattern within the application program that represents the internal states of the program. The

values and frequency of this pattern are stored in the form of a distribution during the execution of a pseudo-oracle to get information about the correct program behavior. This information can be used as an acceptance test for the application program, in which after each run the observed pattern is compared to the data stored in the distribution. If the observed pattern has never occurred before, it corresponds either to a software error or to a false alarm, but false alarms were observed in only three cases. On the other hand, it is also possible that an error produces a pattern that is listed in the distribution, and these events determine the error detection performance.

In order to study the error detection performance, an experiment was performed by using the existing environment of a previous multi-version experiment that uses the launch interceptor problem as a model problem. Several versions instrumented with known single and multiple faults were executed together with an oracle to see if the occurred errors would have been detected by the acceptance test. The results obtained so far show error detection rates between 82% and 100%, except for one case with a rate of 8.3%. Since the error detection rate was greater than zero in all cases, all faults were detected by the acceptance test. The presence of two faults in the same version resulted in a fault-interaction phenomena with an amplifying effect on the number of error occurrences.

The experiment succeeded in proving the feasibility of the proposed test method *in principle*. Further investigations of different faults and with other applications are necessary to draw more general conclusions. But the results show promise for a certain class of applications, especially in the area of parallel programming, where a pseudo-oracle is often available in the form of a sequential version. There are many different, application specific variables that influence the error detection performance of the test, and further empirical studies have to identify the conditions under which this test method can be successfully implemented.

Acknowledgement

I am grateful to Earle Migneault and Larry Morell for many stimulating discussions and valuable suggestions. Thanks are also due to Bernice Becher who modified the control software to perform this experiment, and to Bob Voigt for his support and encouragement.

References

[1] A. Avižienis and J. P. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments", *Computer*, Vol. 17, No. 8, pp. 67-80, Aug. 1984.

[2] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky, "Validation, Verification, and Testing of Computer Software", *ACM Computing Surveys*, Vol. 14, No. 2, pp. 159-192, June 1982.

[3] L. J. Morell, "Unit Testing and Analysis", *SEI Curriculum Module SEI-CM-9-1.0*, Software Engineering Institute, Carnegie Mellon University, Oct. 1987.

[4] V. R. Basili and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies", *IEEE Trans. on Software Eng.*, Vol. SE-13, No. 12, pp. 1278-1296, Dec. 1987.

[5] D. M. Andrews and J. P. Benson, "An Automated Program Testing Methodology and its Implementation", *Proc. of the 5th Intern. Conf. on Software Eng.*, San Diego, CA, pp. 254-261, March 1981.

[6] C. Wild, "Automating Software Fault Tolerance", *Journal of Spacecraft and Rockets*, Vol. 24, No. 1, pp. 86-89, Jan.-Feb. 1987.

[7] E. J. Weyuker, "On Testing Non-testable Programs", *Computer Journal*, Vol. 25, No. 4, pp. 465- 470, Nov. 1982.

[8] T. Anderson and P. A. Lee, *Fault Tolerance Principles and Practice*, Englewood Cliffs, NJ, Prentice/Hall Intern., 1981.

[9] J. C. Huang, "Program Instrumentation and Software Testing", *Computer*, Vol. 11, No. 4, pp. 25-32, April 1978.

[10] R. L. Probert, "Optimal Insertion of Software Probes in Well-Delimited Programs", *IEEE Trans. on Software Eng.*, Vol. SE-8, No. 1, pp. 34-42, Jan. 1982.

[11] J. W. Duran and S. C. Ntafos, "An Evaluation of Random Testing", *IEEE Trans. on Software Eng.*, Vol. SE-10, No. 4, pp. 438-444, July 1984.

[12] P. M. Nagel and J. A. Skrivan, "Software Reliability: Repetitive Run Experimentation and Modeling", *NASA Contractor Report 165836*, Feb. 1982.

[13] J. R. Dunham and J. L. Pierce, "An Experiment in Software Reliability", *NASA Contractor Report 172553*, March 1985.

[14] J. R. Dunham, "Experiments in Software Reliability: Life-Critical Applications", *IEEE Trans. on Software Eng.*, Vol. SE-12, No. 1, pp. 110-123, Jan. 1986.

[15] J. C. Knight and N. G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming", *IEEE Trans. on Software Eng.*, Vol. SE-12, No. 1, pp. 96-109, Jan. 1986.

[16] S. D. Cha, N. G. Leveson, T. J. Shimeall, and J. C. Knight, "An Empirical Study of Software Error Detection Using Self-Checks", *Proc. of the 17th Intern. Symp. on Fault-Tolerant Computing (FTCS-17)*, Pittsburgh, PA, pp. 156- 161, July 1987.

NASA National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181667 ICASE Report No. 88-31		

4. Title and Subtitle	5. Report Date
AUTOMATICALLY GENERATED ACCEPTANCE TEST: A SOFTWARE RELIABILITY EXPERIMENT	May 1988
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Peter W. Protzel	88-31
	10. Work Unit No. 505-90-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18107
	13. Type of Report and Period Covered: Contractor Report
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Final Report

Submitted to Proc. of the Second Workshop on Software Testing, Verification, and Analysis, Banff, Alberta, Canada

16. Abstract

This study presents results of a software reliability experiment that investigates the feasibility of a new error detection method. The method can be used as an acceptance test and is solely based on empirical data about the behavior of internal states of a program. The experimental design uses the existing environment of a multi-version experiment previously conducted at the NASA Langley Research Center, in which the "launch interceptor" problem is used as a model problem. This allows the controlled experimental investigation of versions with well-known single and multiple faults, and the availability of an oracle permits the determination of the error detection performance of the test. Fault-interaction phenomena are observed that have an amplifying effect on the number of error occurrences. Preliminary results indicate that all faults examined so far are detected by the acceptance test. This shows promise for further investigations, and for the employment of this test method in other applications.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
software reliability, acceptance test, fault-interaction	38 - Quality Assurance and Reliability 61 - Computer Programming and Software Unclassified - unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	21	A02