Reducing Uncertainty About Common-Mode Failures

Jeffrey Voas, Anup Ghosh, Frank Charron
{jmvoas,aghosh,fhchar}@rstcorp.com
Reliable Software Technologies
21515 Ridgetop Circle #250
Sterling, VA 20166
http://www.rstcorp.com

Lora Kassab[i]
kassab@itd.nrl.navy.mil
Naval Research Laboratory
Center for High Assurance Computer Systems
4555 Overlook Avenue, SW, Code 5542
Washington D.C., 20375

[i] L. Kassab performed this work while a graduate student at the College of William and Mary.

Abstract

Multi-version programming is employed in fault-tolerant computer systems in order to provide protection against common-mode failure in software. Multi-version programming involves building diverse software implementations of critical functions. The premise of building diverse versions is that the likelihood of a programming error in one version causing a failure in an identical manner as an error in another version is reduced. Skeptics of multi-version programming have correctly pointed out that common-mode failures between redundant diverse versions can reduce the return on investment in creating diverse versions.

To date, other than using historical data from other projects, there has been no way to estimate the potential for a given multi-version programming system to suffer a common-mode failure. This paper presents an algorithm and software analysis prototype to reduce the uncertainty of whether software flaws in diverse versions can result in common-mode failure. The analysis uses software fault-injection techniques to subject one or more versions to anomalous behavior. From this, we can predict how the software will behave if real faults exist in the multiple versions.

1. Diversity and Various Perspectives

Although software systems made of redundant software programs are fairly uncommon in the United States, they are looked upon more favorably elsewhere. Airbus Industrie, the European consortium which competes directly with Boeing Co., uses diverse software programs for the A320/A330/A340 electrical flight control systems. For example, Airbus uses two (2) different types of computers for flight control in the A320. The two computers, whose monikers are SEC and ELAC, are designed and manufactured by different equipment manufacturers using different microprocessors, different computer architectures, and different functional specifications. Each flight control computer uses one channel for control and another channel for monitoring. Since a different software program is used for each of these channels on each redundant computer, a total of four (4) different software packages are used in the control and monitoring of the A320 flight control system [15][4]. By achieving diversity in hardware and software, Airbus hopes to mitigate the common-mode failure problem in redundant computer systems.

Redundancy is also prevalent in nuclear power systems. Digital instrumentation and control systems in nuclear power plants employ independent protection systems to detect system failures in order to isolate and shut-down failed subsystems. The U.S. Nuclear Regulatory Commission (NRC) has developed a position with respect to diversity, as stated in the technical position document “Digital Instrumentation and Control Systems in Advanced Plants” [13]. Two excerpts from this document are particularly relevant for requiring the assessment of common-mode failures:

1. *The applicant shall assess the defense-in-depth and diversity of the proposed instrumentation and control system to demonstrate that vulnerabilities to common-mode failures have been adequately addressed. The staff considers software design errors to be credible common-mode failures that must be specifically included in the evaluation.*

2. *In performing the assessment, the vendor or applicant shall analyze each postulated common-mode failure for each event that is evaluated in the analysis section of the safety analysis report (SAR) using best-estimate methods. The vendor or applicant shall demonstrate*

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **1997**	2. REPORT TYPE	3. DATES COVERED **00-00-1997 to 00-00-1997**

4. TITLE AND SUBTITLE **Reducing Uncertainty About Common-Mode Failures**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**12**	

> *adequate diversity within the design for each of these events.*

The Canadian Atomic Energy Control Board (AECB) recognizes the danger of common-mode failures in nuclear control applications as well. The AECB requirements for achieving software diversity are succinctly stated in draft guide C-138, “Software in Protection and Control Systems” as [1]:

> *To achieve the required levels of safety and reliability, the system may need to be designed to use multiple, diverse components performing the same or similar functions. For example, AECB Regulatory Documents R-8 and R-10 require two independent and diverse protective shutdown systems in Canadian nuclear power reactors. It should be recognized that when multiple components use software to provide similar functionality, there is a danger that design diversity may be compromised. The design should address this danger by enforcing other types of diversity such as functional diversity, independent and diverse sensors, and timing diversity.*

Clearly these two nuclear regulatory bodies have recognized common-mode failures as a critical weakness in redundant component implementations of nuclear control systems. It is interesting to note that they recommend preventative design measures as well as evaluative measures to address the common-mode failure problem.

From industry recommendations, the U.S. Federal Aviation Administration (FAA) has formulated a different perspective on redundancy. Their position is that since the degree of protection afforded by design diversity is not quantifiable, employing diversity will only be counted as additional protection beyond the already required levels of assurance [6]:

> *The degree of dissimilarity and hence the degree of protection is not usually measurable. Probability of loss of system function will increase to the extent that the safety monitoring associated with dissimilar software versions detects actual errors or experiences transients that exceed comparator threshold limits. Multiple software versions are usually used, therefore, as a means of providing additional protection after the software verification process objectives for the software level have been satisfied.*

The U.S. Office of Device Evaluation of the Center for Devices and Radiological Health of the U.S. Food and Drug Administration (FDA) has issued a report that applies to the software aspects of pre-market notification submissions for medical devices [7]. The FDA does not dictate any particular approach to safety, nor does it dictate specific software quality assurance and development procedures. Because there is no specification on how safety is to be achieved nor demonstrated, the FDA provides no guidance on redundancy and diversity.

2. Software Composition

Fault-tolerant computer systems that use redundancy are employed in a wide-range of safety-critical and ultra-reliable applications, including nuclear control, flight control, and medical devices. In systems where software is replicated on redundant platforms, a failure resulting from a flaw in software on one platform is certain to result in other redundant platforms for the same input, since the replicated software has replicated flaws. In these types of fault-tolerant architectures, redundant hardware platforms only protect the system from anomalous or transient errors resulting from hardware faults or external corruptions. The reliability afforded such a system by the software can be modeled as a series reliability block diagram, because the reliability of the multiple software versions is only as good as the reliability of a single version.

In order to provide some level of protection against redundant software programs failing identically, diverse multi-programming, also known as *N*-version programming, has been advocated [2][3]. Other references that more fully discuss *N*-version programming are [11] and [12]. In *N*-version programming, different software versions, written to the same specification but developed independently, are run in parallel. If there is no correlation between version failures, then the system’s dependability is essentially the product of version dependabilities—which has the potential for construction of nearly perfect systems from imperfect programs. Diversity in program versions attempts to prevent redundant programs from failing identically, or from failing simultaneously. To protect against common design errors, diversity in design is employed. Functional diversity involves specifying that different programs have different functional requirements. For example, one program might do a linear search, while another performs a binary search. The goal of finding the element in a list might be the same, but the algorithm specified will be different. The techniques specified in this paper are well-suited to both types of software diversity.

Despite the diverse software implementations, it is believed that common errors compromise the independence between multiple versions that is needed to make diversity worthwhile [9][5]. By definition, a *common-mode failure* (CMF) occurs when two or more software versions fail in exactly the same way for the same input. Common-mode

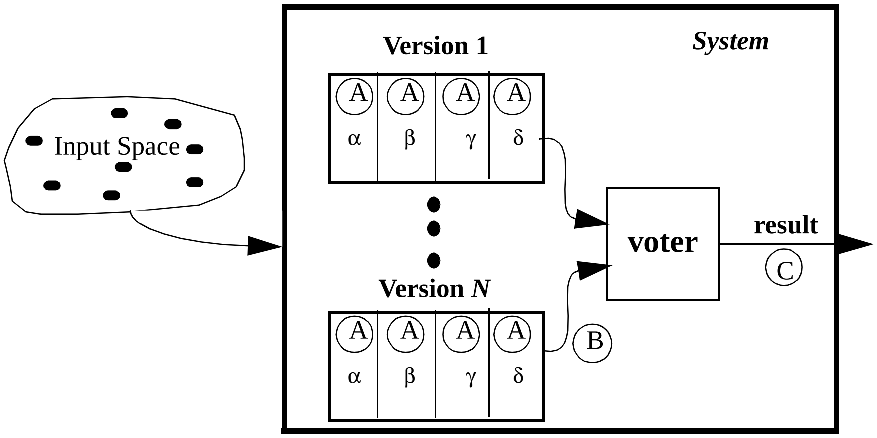


Figure 1: *N*-version system architecture

failures are said to occur when there exists at least one input combination for which the outputs of two or more versions are erroneous, and the outputs are identical for all possible input combinations [8]. Thus, if two or more versions respond to all inputs in the same way, and there is at least one input combination that causes them both to respond incorrectly, then a common-mode failure has occurred. Since it is a practical impossibility to test that all versions of a program will respond identically to all inputs, we loosen the definition for common-mode failure by defining a "single input" common-mode failure. A single input CMF occurs when there exists at least one input for which the outputs of two or more versions of a program are identically incorrect. This definition loosens the restriction that the outputs for *all* inputs must also be identical. Note that CMFs do not have to occur only from identical faults, however, that class of problem is the predominant cause of CMFs. In this paper, we develop techniques that allow prediction of CMFs that result not only from identical faults, but also from uncorrelated anomalies in different combinations of versions.

Knight and Leveson demonstrated that different programmers can make the same logical error [9]. An additional result involved cases where different logical errors yielded common-mode failures in completely distinct algorithms or in different parts of similar algorithms. The technique presented here is concerned with these "different logical errors". The technique can determine if a flaw in one function in one version can result in a common mode failure with a flaw in a different function in another version. The technique can be applied equally to faults in similar functions, as well, though this outcome has less value since the likelihood of CMFs is greater for faults in similar functions. The goal of this analysis is to provide both an indication of the potential for common-mode failures to occur and the statements of where faults could hide that cause CMFs. This information, in turn, can be used by developers of multi-version fault-tolerant systems to make them more robust against common-mode failures.

3. Assessing the Likelihood of Common-mode Failures

Recognizing the importance of predicting the potential for common-mode failures, we have developed an algorithm and a prototype software analysis tool to observe common-mode failures produced by combinations of simulated programmer faults. Consider the simple *N*-version system depicted in Figure 1. This figure illustrates an architecture of an *N*-version system that is composed of *N* independent programs executing identical inputs sampled from the input space. Each program consists of four functions labeled α through δ that were coded from a common specification. To achieve fault tolerance through replication, each version in Figure 1 would be an identical replica. In this case, software flaws in one version will be present in each of the other $N-1$ versions. An input that triggers a flaw in one version that ultimately causes a faulty output to result will guarantee a common-mode failure between the identical versions. Fault tolerance through *N* replicated versions can only be achieved when faults that affect one version do not affect another, *e.g.,* a hardware fault or external corruption.

To achieve fault tolerance through diversity, each version implements the individual functions in a diverse manner. The rationale behind programming diverse versions is to prevent programming errors in different versions from resulting in common failures. For example, system fault

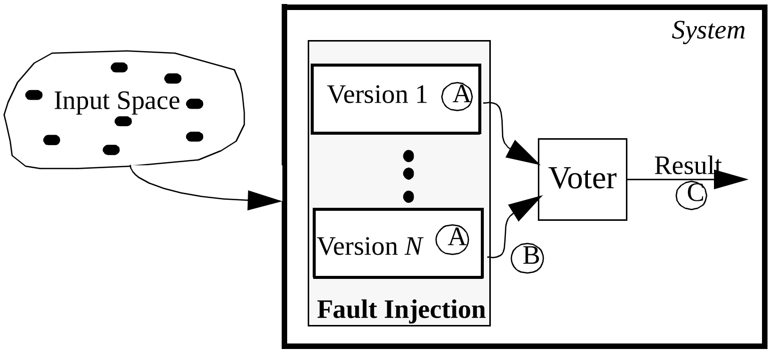


Figure 2: *N*-version system with simulated faulty versions.

tolerance is sought from flaws in Version 1 (in Figure 1) that do not exist in a majority of the other versions. A voter decides which output to release from the *N* programs based on a pre-specified algorithm. The reliability of an *N*-version system can be defeated by problems in two areas: (1) common-mode failures between a multiple versions, and (2) flaws in the voter that result in incorrect or unreliable outputs. The approach presented in this paper addresses one class of anomaly that can contribute to (1). An approach to address concern (2) is developed in [17].

In our approach, a fault-injection--based method is used to simulate uncorrelated programmer flaws in diverse versions. The effect of the simultaneous faults is observed and a count of the resulting common-mode failures is tallied. If the frequency of common-mode failure observations is unacceptably high, then either different versions of the program can be swapped in and re-analyzed, or additional fault-tolerant mechanisms can be added to reduce the likelihood of common-mode failure, *e.g.*, by diversifying versions further or by adding additional versions. It is important to realize that not all common-mode failures will defeat an *N*-version system. For example, a 5-version system that employs a majority voter will be able to tolerate a common-mode failure between at most two versions. Therefore, common-mode failures can be ranked by *severity* depending on the system configuration and the voter algorithm. In a majority voting system, a common-mode failure involving the majority of versions will be classified as severe, while a common-mode failure in a minority of versions may be tolerable. If the analysis reveals that common-mode failures are unlikely or that the voter is survivable to multiple component failures, then it can be argued that placing redundant software versions in parallel can increase the reliability of the system, even if we are unsure about or dissatisfied with the quality of the components.

3.1 Anomalous events

Software that does not experience any problems in its program state during execution cannot behave dangerously. Only when software encounters problems that corrupt its program state can things go awry. Anomalous events (or anomalies) are problematic events that occur during software execution in any state of the software that has the potential to alter the output of the software such that the output is different than if the events had not occurred. Anomalies can be caused by internal code defects or external failures that behave as input to the program. The number of different anomalies that software may be forced to experience during its lifetime is effectively infinite, and the members of this set are unknown. Software fault-injection simulates anomalous events in the execution of software under specific input conditions.

For *N*-version systems, the key failure class to be prevented against is common-mode failure. The key anomaly classes to simulate using fault injection are: (1) programmer faults and (2) simulated specification errors and ambiguities. It is easy to understand how errors in specification for an *N*-version system can result in common-mode failures between diverse versions. For example, if the threshold temperature at which a fail-safe nuclear rod cooling system trips is specified in error, then it is easy to imagine how three diverse, redundant fail-safe systems will fail identically because of the incorrectly specified threshold. Section 3.3, addresses the issue of errors in specification and introduces a technique for assessing the effect of specification errors and ambiguities on the resulting system output. What is non-intuitive to understand and difficult to defend against is how unrelated flaws in programming can result in identically incorrect outputs (common-mode failures). Recall that the goal of programming diverse versions was to tolerate uncorrelated flaws in individual versions. There is no assumption that every version is correct, but rather the hope is that the flaws in different versions are unique and will be tolerated by the voting algorithm.

It can be postulated that because of the complexity of software and the number of different failure modes of software, flaws in diverse versions in possibly different functions may combine in unexpected ways to result in common-mode failure. For the example illustrated in Figure 1, consider a software flaw in some function α in Version 1. Next, consider a flaw in a different function δ in Version *N*. It is unknown if the flaw in function α can combine with the flaw in function δ to result in a common-mode failure. To our knowledge, there has been no technique for testing this hypothesis for a given multi-version system, until now.

The new analysis technique presented in this paper tests this hypothesis. This technique simulates flaws in multi-versions and observes the resulting outputs. If the simulated flaws result in identically incorrect outputs (for the same input), then we postulate that there is potential for an actual common-mode failure if actual flaws exist in the statements that were perturbed.

In order to detect the potential for common-mode failures between multiple diverse versions, corrupted states that mimic the effects of programmer faults are injected within executing versions in statements corresponding to **A** in Figure 1. The effect of the corrupted states in each version is observed at the output of each version (location **B** in Figure 1). For a given input to the multiple versions, if two or more versions produce an output that is both identical *and* different from the output that would be produced without fault injection, then a “single input” common-mode failure is said to have occurred due to the corrupted states. The analysis does not reveal whether a common-mode failure would occur in practice; rather the analysis only reveals the potential for a common-mode failure occurring in the future.

It is important to realize that the common-mode failure likelihood assessed through analysis will vary from implementation to implementation as well as with the inputs used and the corruptions injected. No warranties are given with respect to the representativeness of the corruptions.

The fault-injection process involves injecting anomalies into a piece of software using *instrumentation*. Instrumentation typically involves inserting code into the code that is being analyzed, and then compiling and executing the modified (or *instrumented*) software. In this paper, we will simulate anomalies using instrumentation that is customized for code defects, which are caused by programming errors or design defects.

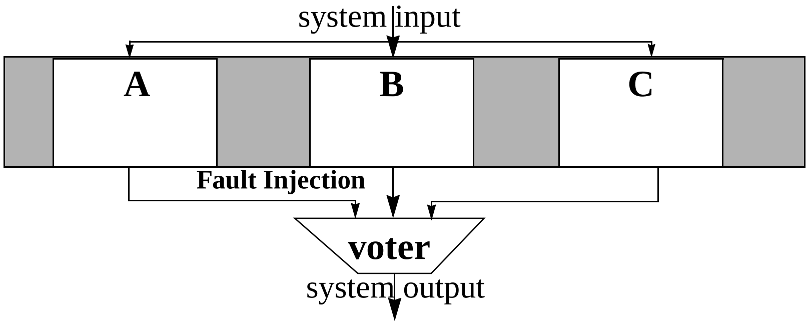
3.2 Simulating faulty versions

In Figure 2, the highlighted portion represents the versions, which are where instrumentation will be inserted to simulate coding defects. The goal is to predict whether problems that manifest themselves in the versions (at locations **A** in Figure 2) can propagate out of the versions to location **B**—the interfaces between the versions and the voter. To simulate faulty software, instrumentation is traditionally employed in one of two methods: *state perturbation* and *mutation*. State perturbation usually adds code to modify the program states created by the original code. Mutation changes the syntax of existing code statements. Explanations for how to implement these two approaches can be found in [19] and [14]. The work described in this paper only uses state perturbation, though the algorithms to follow do not preclude using mutation instrumentation.

In a multiple-version system, an interesting combinatorics problem arises during fault injection. The problem is which combination of versions will fault-injection be applied to. From mathematics, we know the number of combinations of *n* distinct objects taken *r* at a time is given by: $n!/((n-r)! \cdot r!)$. For an *N*-version system, there are $\sum_{r=1}^{N} N!/((N-r)! \cdot r!)$ total combinations. This total should not be too large since *N* is usually a small odd integer, such as three, five or seven.

An algorithm for detecting single input common-mode failures is presented in Figure 3. The algorithm is shown for three versions for clarity, without loss of generality in *N* versions. This algorithm provides warnings each time the output satisfies the definition for a single input common-mode failure. As defined earlier, a “single-input” common-mode failure is a common-mode failure for *one* input case, as opposed to identical outputs for *all* inputs.

The algorithm calls for the execution of a set of inputs on all versions in two different phases: the original run without state perturbations and the perturbed run that employs fault injection. In the original run, the outputs from each version for each input is temporarily stored. Each version is then executed again on the same input, but is subjected to fault injection. This execution is called the perturbed run [19]. If the output from the perturbed run is different from the original run, then the perturbed output is stored for a later comparison (in Step 6.0 of the algorithm). This process is repeated for all instrumented statements in all versions for each input used. A comparison of all perturbed outputs for a given input is performed to determine if a single input common-mode failure resulted. A single input common-mode failure is counted when the perturbed outputs are identical and different from the original outputs for a single input.



```
1.0 Create a set of input vectors, $\mathbf{I}$, to be run on versions A, B, & C.
2.0 Instrument all analyzeable statements in A, B, & C for fault perturbation.
3.0 Execute A, B, & C unperturbed on input $i \in \mathbf{I}$.
4.0 Temporarily store the resulting outputs, $o \in \mathbf{O}$, for A, B, & C.
    4.1 Designate the output $o$ resulting from the execution of A on $i$, $A_{o_i}$. Similarly, designate $B_{o_i}$ and $C_{o_i}$.
5.0 Execute A, B, & C *perturbed* on input $i \in \mathbf{I}$.
    5.1 For each $i$, save the set of perturbed outputs $A'_{o_i}$, $B'_{o_i}$, and $C'_{o_i}$ where $A'_{o_i} \neq A_{o_i}$, $B'_{o_i} \neq B_{o_i}$, and $C'_{o_i} \neq C_{o_i}$.
    5.2 Designate the sets resulting from 5.1 as $\{P_{A_i}, P_{B_i}, P_{C_i}\}$.
6.0 For each input i, compare $\{P_{A_i}, P_{B_i}, P_{C_i}\}$ .
    6.1 Record all non-empty instances of $\{P_{A_i} \cap P_{B_i}\}$, $\{P_{A_i} \cap P_{C_i}\}$, $\{P_{B_i} \cap P_{C_i}\}$, and $\{P_{A_i} \cap P_{B_i} \cap P_{C_i}\}$.
    6.2 Increment CMF counter for each instance.
    6.3 Record each statement that resulted in the CMF: $\{F_{A_i}, F_{B_i}, F_{C_i}\}$.
```

Figure 3: Algorithm 1: Detecting the presence of common-mode failures in multiple version programming.

Recall that *N*-version programming is a method to build a more reliable system from potentially unreliable parts. The premise of *N*-version diverse programming recognizes that not every version will be correct. While one or more versions may have flaws, a flaw in one version can be compensated by a majority of other versions that do not contain that same flaw. The assumption is that as long as the majority of versions do not fail identically on the same input, then common-mode failures may be avoided. This assumption can be tested for a given *N*-version system by the analytical technique presented in Figure 3.

3.3 Simulating Errors in Specification

To this point, we have focused on simulating uncorrelated programmer faults. One of the greatest concerns when deploying an *N*-version system, however, is the potential for specification errors to cause correlated programmer faults that, in turn, produce common-mode failures. Errors in specification certainly have the potential to result in incorrect software outputs. In *N*-version programming systems, presumably each independent programming team works from an identical specification of the application problem. An error in the specification will map into *N* diverse implementations of the same error. If all *N* versions were to correctly implement the erred specification, then the potential for common-mode failures is quite high. *N*-version programming was designed not necessarily to provide protection against specification errors, but rather to protect the system against programming errors. Even so, ambiguous or ill-defined specifications may also lead to diverse implementations that are indifferent to specification errors.

Specification errors are simply specifications that are either incomplete, ambiguous, or incorrect. Given the uncertainty in the effect of specification errors on *N*-version systems, we have also developed a method that observes the impact of simulated specification errors or specification ambiguities on the voter. This observation is made possible by a family of fault injection algorithms customized for

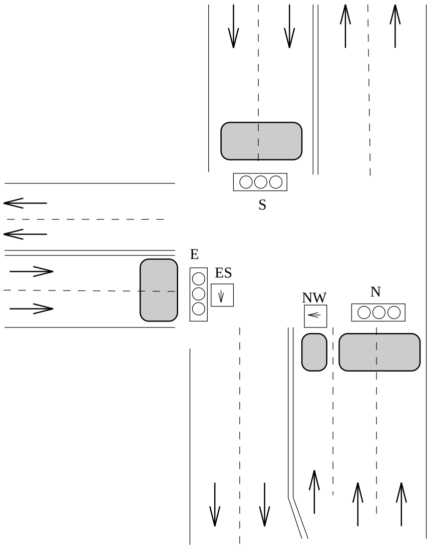


Figure 4: Traffic controller application

specification anomalies. If the impact on the voter is negligible from this class of simulated anomalies, that suggests that the likelihood of common-mode failures from real specification anomalies might be less severe. In the report [17], this specification-based analysis provides information concerning which portions of the specification, if even slightly wrong or misinterpreted, will lead to system failures. Results from using this specification-based analysis are published in [18]. This provides insight as to which directives in the specification have the most impact on the system's functionality. This technique provides a *sensitivity analysis* of the effects of errors or ambiguities in the specification on the voter of an *N*-version system. For those persons charged with verifying requirements, this information could prove to be useful.

4. Experiment

In order to study the usefulness of the algorithms that simulate uncorrelated programmer faults, a prototype *N*-version analysis tool was developed and applied to a set of diverse student programs. The prototype tool currently only implements the analysis of common-mode failures due to programming errors in multiple versions. The prototype tool implements algorithm shown in Figure 3. The experimentation used three independent versions of a controller for managing the traffic lights and turn arrows for a particular intersection. These versions were written by students of Prof. Adam Porter at the University of Maryland, and are based on a specification developed by Porter. The specification is not formal, and since these were student versions, it is likely that the groups communicated. Further, these programs were not designed according to *N*-version programming principles. The point of this experimentation is not to assess the quality of a fault-tolerant traffic controller, but rather to demonstrate the types of results produced by an implementation of the common-mode failure analysis algorithm. While the traffic intersection is real, the programs are fictitious in that they are not deployed in an actual traffic controller. The objective of the experiment, however, is to gauge the applicability of the developed algorithms and prototype tool for assessing the common-mode failure likelihood. As the wisdom of this type of analysis becomes apparent, it is expected that the analysis will be applied to "real-world" fault-tolerant systems from industry as they become available.

Figure 4 depicts the intersection that is controlled by the traffic lights. Traffic can move along this road going north-bound (N), south-bound (S), east-bound (east to north (E), east-to-south (ES)), and north-to-west (NW). There is a traffic light controlling all north-bound, south-bound, and east-bound lanes. There are also two turn arrows, one for the north-to-west turn lane and for the east-to-south lane.

Version Combination	Number of Potential Common-mode Failures Detected
$A \cap B$	156
$A \cap C$	631
$B \cap C$	143
$A \cap B \cap C$	101

Table 1 : Results from analyzing traffic control system for common-mode failures

Each software version requires inputs from the user, at which point the software will generate the appropriate traffic light controls (outputs) for all lanes at the intersection. This cycle may iterate indefinitely. User inputs represent physical sensors under the roadway. There are four (4) sensors. One for all east-bound (E) lanes, one for all south-bound (S) lanes, one for the north-bound (N) lanes and one for the north-to-west (NW) turnout lane. A sensor emits an input signal only if at least one car is in the corresponding lane. The rate at which sensors emit signals is arbitrary. At every state change the system provides several outputs. These outputs signify the color (GREEN, YELLOW, or RED) of every traffic light, and indicate whether or not every traffic light is illuminated.

The traffic controller for the intersection depicted in Figure 4 was implemented by a number of different students. The implementation of the traffic control functions were required to adhere to standard calls from the traffic controller `main` C function. Three different versions were hooked together in `main` and called to calculate the traffic controller state for each input. A voter was coded to execute a majority vote on the state of the traffic controller. The common-mode failure analysis algorithm of Figure 3 was applied to the three implementations to determine the likelihood of common-mode failures to simulated flaws in each version. Each test case consisted of a number of sequential random inputs to the traffic controller corresponding to the (N,S, E, and NW) sensors at the intersection (Step 1.0 in Figure 3). Each version was then executed with a given input in an unperturbed manner and the unperturbed outputs were temporarily stored (Steps 3.0 and 4.0). Next, the prototype tool injected faults into the execution of each version and determined if perturbed outputs were different from the unperturbed outputs for each input (Steps 5.0, 5.1, 5.2). Finally, the perturbed outputs between different versions were compared to determine if any were identically incorrect (Steps 6.0 and 6.1). The common-mode failure counter was incremented for each identically incorrect output (Step 6.2).

The results from the applying the common-mode failure analysis to the fault-tolerant traffic light control system are summarized in Table 1. The results show the number of common-mode failures detected through fault injection in the multiple versions for all test cases. As a result, many common-mode failures were repeatedly observed. Using the frequency of common-mode failure detection as a metric, it can be asserted that the higher the frequency of detecting common-mode failures through repetitive random testing, the higher the likelihood that common-mode failures will result in the future execution of the system. From the table, it is clear that versions *A* and *C* have a higher likelihood of common-mode failure than the other versions in combination. The results showing over 100 common-mode failures detected between all three versions is also significant. Another interesting metric that we hope to capture but have not yet is how many of these potential common mode failures will actually cause system failure.

In order to provide software developers a means for improving the fault tolerance of the system under analysis, the program statements where common-mode failures originated (Step 6.3 in Figure 3) are presented together with their distribution in Figure 5 and Figure 6[1] (at the end of the document). The analysis reveals the potential vulnerable statements in the source code for the three versions where common-mode failures might originate. The program statements where common-mode failures were detected along with the frequency of detection between versions *A* and *B* are shown in Figure 5. By examining this histogram, the problem areas in the code can be quickly assessed. The most frequent common-mode failures in statement pairs (*B*,*A*) were (432,349), (437,355), and (437,352). Figure 6 shows a similar diagram for common-mode failures occurring between versions *A*, *B*, and *C*. The critical statements in versions (*C*,*B*,*A*) for common-mode failures are (832,437,355), (832,437,352), and (964,432,349). Notice that the most common (B,A) pairs appeared in the most common (C,B,A) triples. The same common-mode failures appear most frequently throughout the analysis. Inspection of the programs revealed that the majority of the common-mode failures were found in the different implementations of the traffic light control logic. This result should be somewhat unsurprising since all three versions worked from the same specification of the traffic light control logic.

It is interesting to note that versions *A* and *B* are most similar in implementation, while version *C* diverges significantly from versions *A* and *B* in implementation. Despite the diverse implementations, the frequency of common-mode failures was highest between versions *A* and *C*. Further examination of these two programs revealed that a very large number of common-mode failures detected between versions *A* and *C* were caused by anomalies injected in their respective initialization functions. These results suggest that despite seemingly diverse implementations between versions, common-mode failures are possible and even likely in certain functions implemented similarly in otherwise diverse programs. In contrast, the initialization function coded in version *B* departed from versions *A* and *C* and as a result, few common-mode failures were found involving the initialization functions between version *B* and the two other versions. The type of common-mode failure observed most often between versions *A* and *B* was due to faults injected in nested conditional or `case` statements. The “drop-through” return value in these expressions was often correlated between versions *A* and *B* in spite of diverse implementations of the control expression in the conditional statements. This correlation resulted in common-mode failures at the outputs of the versions. By interpreting these results, system developers can determine if and where potential common-mode failures might originate from flaws in the code and then apply mitigating measures.

5. Conclusions

The premise underlying this paper is that uncorrelated *programmer errors* in diverse versions can result in common-mode failure that will defeat the voter of an *N*-

[1] These results have not been manually validated due to the complexity involved in manual validation.

version programming system. It is intuitive that supposedly equivalent programs that were built by different teams of programmers will possess a reduced likelihood of common-mode failure. To date, there has not been a technique for determining the extent to which diverse versions from a specification can mitigate the effects of common-mode failure. The algorithm and prototype described in this paper provide a new perspective for determining where uncorrelated programmer errors can enable common-mode failures. The results from the analysis show *where* in diverse programs that common-mode failures are more likely to occur. Even if we accept the hypothesis that diverse programs do indeed reduce the likelihood of common-mode failure, then the experimental results afforded by the prototype can aid in deciding how worrisome common-mode failure is for a particular *N*-version system.

The analysis results presented in this paper are a preliminary application of the common-mode failure analysis algorithm. Further research currently underway is assessing the tolerance of the system to simulated specification errors. The potential for decreasing the common-mode failure likelihood by swapping out one version for another may also be explored.

Acknowledgments

The authors thank Adam Porter of the University of Maryland for providing us with the diverse versions used in the analysis and supplying us with Figure 4. This work has been partially supported by DARPA Contract F30602-95-C-0282 and NIST Advanced Technology Program Cooperative Agreement No. 70NANB5H1160.

6. References

[1] Atomic Energy Control Board, *Draft Regulatory Guide* C -138, 1996.

[2] A.A. Avizienis, “The Methodology of N-version Programming”, *Software Fault Tolerance*, edited by M. Lyu, John Wiley & Sons, 1995, pp. 23-46.

[3] A.A. Avizienis and L. Chen, “On the Implementation of N-version Programming for Software Fault Tolerance During Execution”, *Proceedings of the IEEE COMPSAC 77*, November 1977, pp. 149-155.

[4] D. Briere and P. Traverse, “Airbus A320/A330/A340 Electrical Flight Controls -- A family of fault-tolerant systems”, *Proceedings of the 23rd Fault Tolerant Computing Symposium*, pp. 616-623, Toulouse, FR, June 1994.

[5] S. Brilliant, J. Knight, and N.G. Leveson, “Analysis of Faults in an N-version Software Experiment”, *IEEE Transactions on Software Engineering*, SE-16(2), February, 1990.

[6] Federal Aviation Authority, “Software Considerations in Airborne Systems and Equipment Certification”, *Document No. RTCA/DO-178B*, RTCA, Inc., 1992.

[7] U.S. Food and Drug Administration, *Reviewer Guidance for Computer Controlled Medical Devices Undergoing 510(k) Review*, 1991.

[8] B.W. Johnson, *Design and Analysis of Fault Tolerant Digital Systems*, Addison-Wesley Publishing Company, Reading, MA, 1989.

[9] J. Knight and N.G. Leveson, “An Experimental Evaluation of the Assumption of Independence in Multiversion Programming”, *IEEE Transactions on Software Engineering*, SE-12(1):96-109, January, 1986.

[10] J.L. Lions, “Ariane 5 Flight 501 Failure”, *Report of the Inquiry Board*, Paris, July 19, 1996.

[11] M.R. Lyu and Y. He, “Improving the N-Version Programming Process Through the Evolution of a Design Paradigm”, *IEEE Transactions on Reliability*, 42:2, June, 1993, pp. 179-189.

[12] D.F. McAllister and M.A. Vouk, “Fault-Tolerant Software Reliability Engineering”, in *Handbook of Software Reliability Engineering*, edited by Michael Lyu, McGraw-Hill, New York, NY, 1996, pp. 567-614.

[13] U.S. Nuclear Regulatory Commission, *Draft Branch Technical Position*, 1994.

[14] A.J. Offutt, *Automatic Test Data Generation*, Ph.D. Thesis, Georgia Institute of Technology, Atlanta, GA, 1988. Technical report GIT-ICS 88/28.

[15] P. Traverse, “Dependability of Digital Computers on Board Airplanes”, *Proceedings of the First Dependable Computing for Critical Applications Conference*, Santa Barbara, CA, August, 1989.

[16] J. Voas, A.K. Ghosh, G. McGraw, and K. Miller, “Gluing Together Software Components: How good is your glue?”, *Proceedings of the*

Pacific Northwest Software Quality Conference, Portland, October 1996.

[17] J. Voas, A.K. Ghosh, F. Charron, and L. Kassab, “Reducing Uncertainty About Common-mode Failures” RST Technical Report #RSTR-96-002-1101.

[18] J. Voas and L. Kassab, “Simulating Specification Errors and Ambiguities in Systems Employing Diversity”, In *Proceedings of the 1997 Pacific Northwest Software Quality Conference*, Oct. 1997.

[19] J. Voas and K. Miller, “Software Testability: The New Verification”, *IEEE Software,* 12(3):17-28, May, 1995.

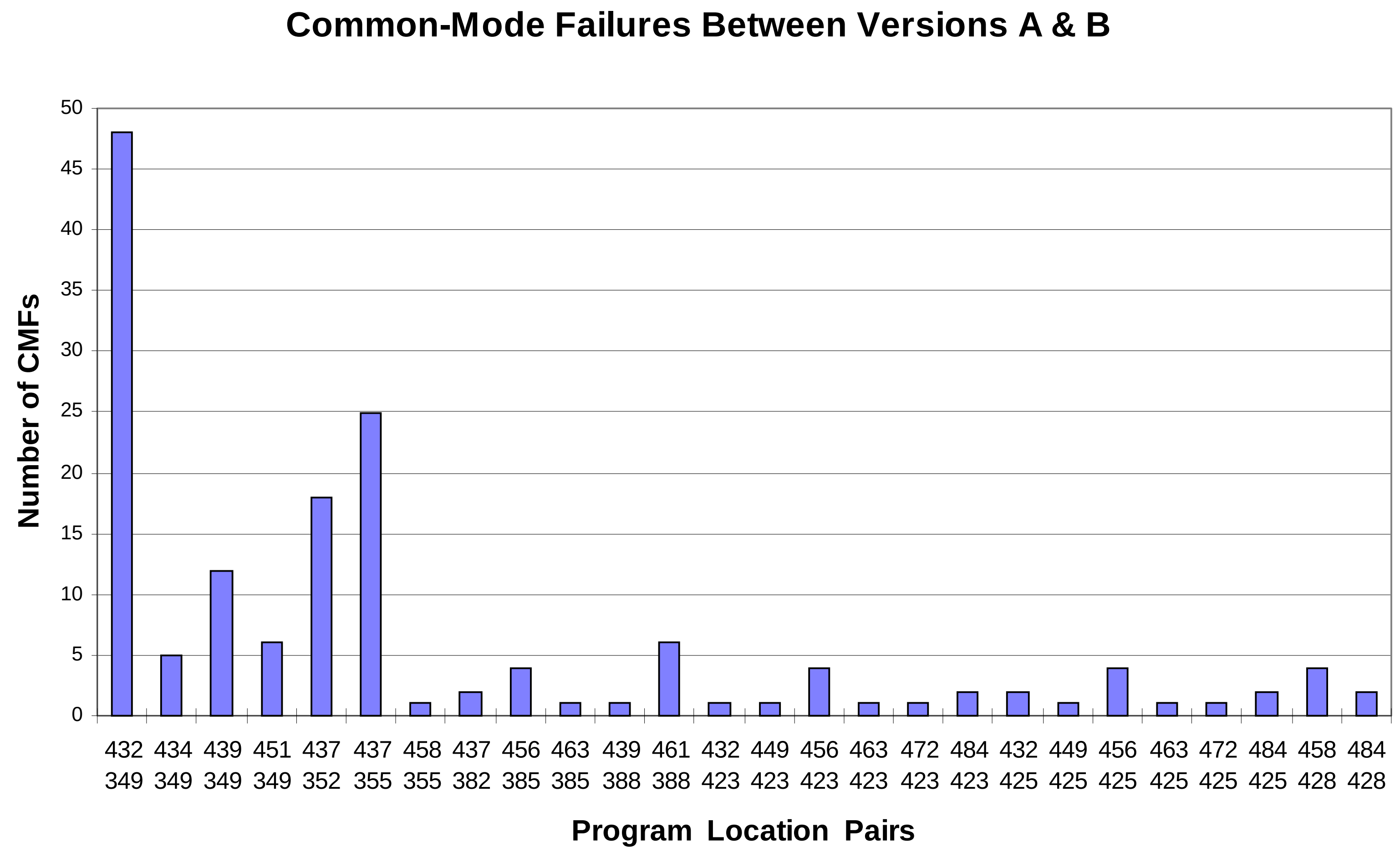


Figure 5: Observed common-mode failures between versions A and B

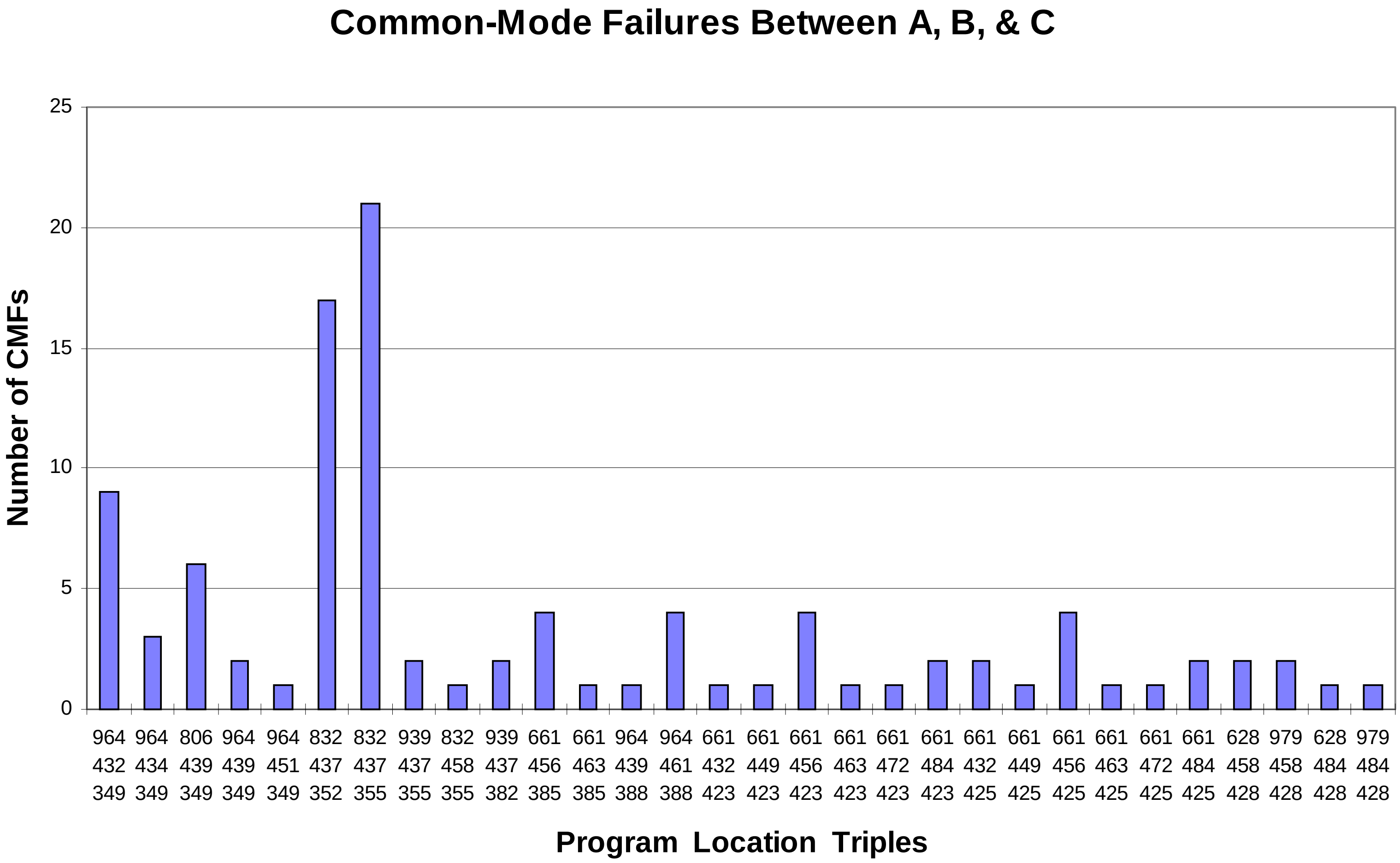


Figure 6: Observed common-mode failures between versions A, B, and C for various program location triples.