Reducing Uncertainty About Common-Mode Failures

J. Voas, A. Ghosh, F. Charron, & L. Kassab

Reliable Software Technologies Corp.
21515 Ridgetop Circle, #250
Sterling, VA 20166

POC: aghosh@rstcorp.com
http://www.rstcorp.com

Abstract

Multi-version programming is employed in fault-tolerant computer systems in order to provide protection against common-mode failures in software design. Multi-version programming uses diverse software implementations of critical functions such that an error in one version will not fail in an identical manner as in another diverse version. 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.

Two critical issues for determining if multi-version programming can increase reliability are: (1) the likelihood of common-mode failures in diverse versions and (2) the reliability of the voter in a multi-version programming system. This paper develops algorithms and software analysis techniques to reduce the uncertainty of these two critical issues occurring in an actual multi-version programming system. The analysis uses software fault-injection techniques to subject redundant diverse versions to anomalous behavior. The effect of the injected faults is observed to determine if common-mode failures have occurred. Our analysis measures the common-mode failure likelihood between multiple versions by observing the frequency of common-mode failures. In addition, the potential weak points in program source code which resulted in common-mode failures through fault injection are identified. This information, in turn, can be used to fortify multi-version fault-tolerant systems against common-mode failures. Results from applying the common-mode failure analysis technique to a traffic controller system that has been implemented in a fault-tolerant diverse software system are presented.

1. Introduction

The world software and services market is currently a U.S. $300 billion industry. The cost of design and development can be significantly reduced by developing a software component industry. Today, most software is sold in the form of an application; it is hoped that someday the subsystems of these applications might perform dual purposes, both within the original application as well as other applications. Studies have indicated that software reuse is capable of a two to seven times increase in software productivity. Given the size of this industry, even more conservative productivity gains will result in billions of dollars of savings. For this to occur, measurement approaches must be endorsed by software consumers and producers that provide knowledge concerning how the quality of one component will affect the rest of the components of the system.

Traditional manufacturing is based on the concept of components. Automobiles, radios, bridges, planes, etc., are made from hundreds or thousands of components. Imagine the difficulty in trying to build a bridge, for example, as a single part. Software development is analogous to traditional manufacturing, particularly since software systems are composed of smaller software objects. What is lacking in the software industry—that is ubiquitous in manufacturing—is the ability to swap components in and out of systems. If the ability to swap components was feasible, software component commerce would evolve, which would decrease the costs of designing and repairing systems. Unfortunately, there are several problems that have prevented our industry from swapping components. In this paper, we focus on a solution to one of these problems: using software components in parallel to increase system reliability.

Previously, we have published results on assessing how component failures affect other components when the components are placed in a series [14]. In this paper, however, we are focusing on ways for predicting how the flaws in one or more components that are placed in parallel can affect the resulting computation output. More specifically, this paper examines the frequency with which flaws in multiple versions of redundant, diverse software cause common-mode failures. Common-mode failures occur when multiple components fail such that their respective outputs are identically incorrect. Common-mode failures have the potential for easily defeating a voting

system for redundant components. This paper additionally presents a method for determining the tolerance of a voter to multiple-version failures. The goal of the latter analysis is to demonstrate the survivability of the voter to individual and multiple component failures.

A fault-injection-based method is used to simulate flaws in redundant diverse versions. The effect of the simultaneous faults is observed to determine if common-mode failures result. If the frequency of common-mode failures 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 failures. On the other hand, 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 components in parallel can increase the reliability of the system, even if we are unsure about or dissatisfied with the quality of the components.

2. Software Components

The combination of software components is analogous to the "series" and "parallel" combinations of mechanical systems. Series combinations are inherent when using components. In placing components in series, the quality of the whole is less than or equal to that of the worst component. Parallel construction must be the mechanism for buying back quality through redundancy. This paper presents techniques to reduce the uncertainty of the likelihood of common-mode failures existing between parallel software components.

Fault-tolerant computer systems that use redundant components 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, an error resulting from a flaw in software on one platform is certain to result in other redundant platforms as well, since the replicated software will have identical software flaws. In this case, the redundant hardware platforms can only protect the system from anomalous or transient errors resulting from the hardware. The reliability of the software system can be modeled as a series reliability block diagram, in this case, 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 components failing in an identical fashion, diverse multi-programming, also known as *N*-version programming, has been advocated [2][3]. 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 components. Diversity in program versions attempts to prevent redundant components from suffering from identical failure modes, or from failing simultaneously. To protect against common design errors, diversity in design is employed. Functional diversity involves specifying that different components have different functional requirements. For example, one program might do a linear search, while another, does 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-mode failures compromise the independence assumption between diverse versions [9][5]. A *common-mode failure* occurs when two or more software components fail in exactly the same way and at the same time. Common-mode failures are said to occur when there exists at least one input combination for which the outputs of the two versions are erroneous, and the outputs are identical for all possible input combinations [8]. Thus, if two versions have experienced faults that respond to all inputs in the same way, and there is at least one combination that causes them both to respond incorrectly, then a common-mode failure has occurred.

The result by 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 techniques presented later in this paper are indiscriminating for where faults are injected in the different versions. As a result, 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 completely different function in another

parallel component. The goal of this analysis is to provide both the frequency and locations of potential common-mode failures to developers of multi-version fault-tolerant systems. This information can be used to assess the likelihood of common-mode failures and to incorporate fault-tolerant solutions to potential problem areas in the code. Throughout this paper, we will assume *majority voting* is the scheme employed by the voter; our approach can easily be modified to accommodate other voting algorithms.

3. Diversity and Various Perspectives

Although software systems made of redundant software components 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 components 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 [13][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. These protection systems usually employ a voting scheme that uses a two-out-of-four logic, where if one channel fails, it is taken out of service and the system goes into a two-out-of-three logic and continues operation. This type of system is termed as being *single-failure proof*. 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” [11]. 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 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.

The U.S. Federal Aviation Administration (FAA) has 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.

4. Assessing the Likelihood of Common-mode Failures

Recognizing the importance of assessing the likelihood of common-mode failures in safety-critical applications, we have developed algorithms and a prototype software analysis tool to detect the potential for common-mode failures between multiple versions of software in a fault-tolerant system. Consider the simple *N*-version system depicted in Figure 1. This figure illustrates the traditional architecture of an *N*-version system that is composed of *N* independent components executing identical inputs sampled from the input space. A voter component (*V)* decides which output to release from the *N* components 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 multiple components, and (2) flaws in the voter that result in incorrect or unreliable outputs. The approach presented in this paper to both problems employs software fault-injection mechanisms at different places in the system of Figure 1.

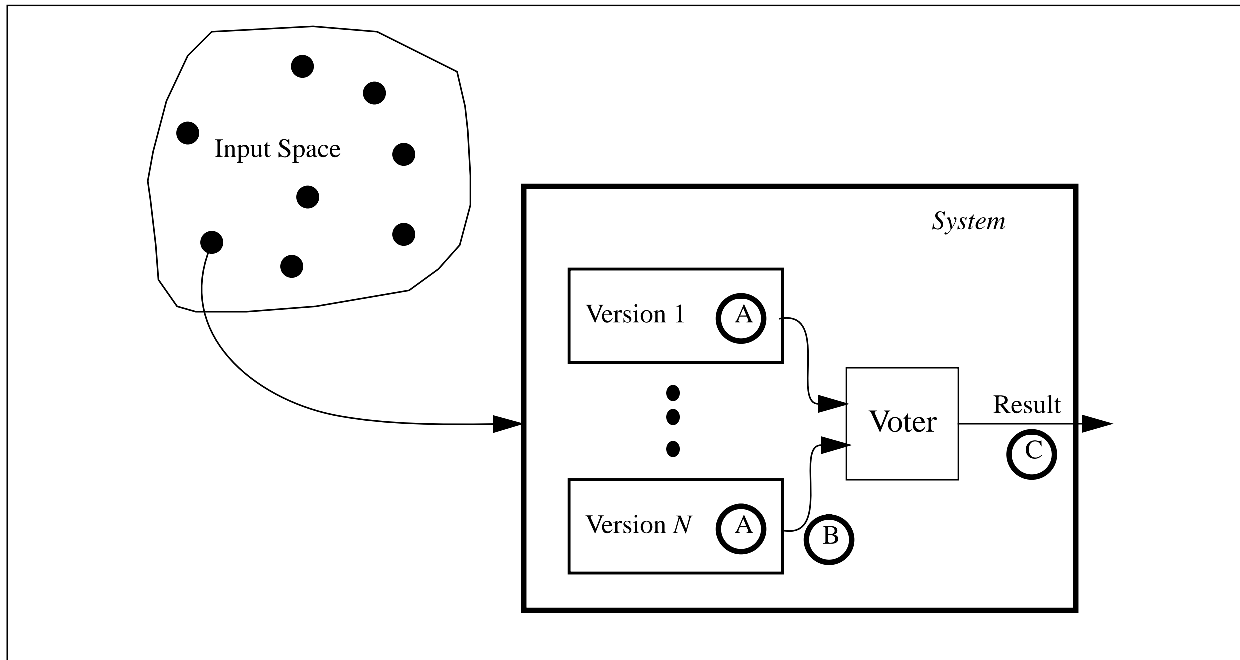

Figure 1: *N*-version system architecture

4.1 Anomalous events

Software that does not experience any problems in its 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. The number of different anomalies that software can 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 given input conditions.

For *N*-version fault-tolerant systems, the key anomaly class to be analyzed is the common-mode failure. The analysis techniques in this paper assess how likely a common-mode failure will occur during the execution of redundant components and where flaws in a program can result in common-mode failures. Other classes of anomalies that may compromise the dependability of component-based systems are also simulated through the analysis techniques presented later in this paper. Successfully fine-tuning fault injection for common-mode failure allows for *a priori*

predictions of whether the future behavior of an N-version subsystem will be acceptable.

In order to detect the potential for common-mode failures between multiple diverse versions, faults are injected within the executing program versions in locations corresponding to **A** in Figure 1. The effect of the injected fault 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 (2) or more versions produce an output which is both identical *and* different from the unperturbed execution, then a common-mode failure is said to have occurred due to the injected faults. The analysis does not reveal whether a common-mode failure has occurred in practice; rather the analysis only reveals the potential for a common-mode failure occurring in the future due to anomalous events in the concurrent execution of the software components.

The second problem of interest is knowing what the level of tolerance V has to other classes of failures that may or may not be common-mode. To assess the reliability of the voter, faults are injected in the outputs of a minority number of software components in locations **B** in Figure 1. The faults injected may be chosen to be common-mode or chosen to be distinct. If the voter produces an output that is either unreliable or unsafe due to a minority number of corruptions, then the voter may be deemed undependable for safety-critical applications. If the analysis shows that particular classes of common-mode failures can be tolerated by a voter, (*e.g.* version 1 and version 8 failing identically in a nine-version system), then the confidence placed on the redundant system under analysis is increased. Further, if the analysis shows that other classes of multiple version failures that do not satisfy the definition for common-mode failure can be tolerated, an even greater confidence in the fault-tolerant system is attained. It is important to realize, however, that the common-mode failure likelihood and the survivability of the voter assessed through analysis will vary from implementation to implementation as well as with the inputs used and the faults injected. Software fault-injection is simply a tool to play the “what-if” games necessary to assess tolerance to anomalous events that may occur during the execution of software. Algorithms for these analysis techniques are presented later in this paper.

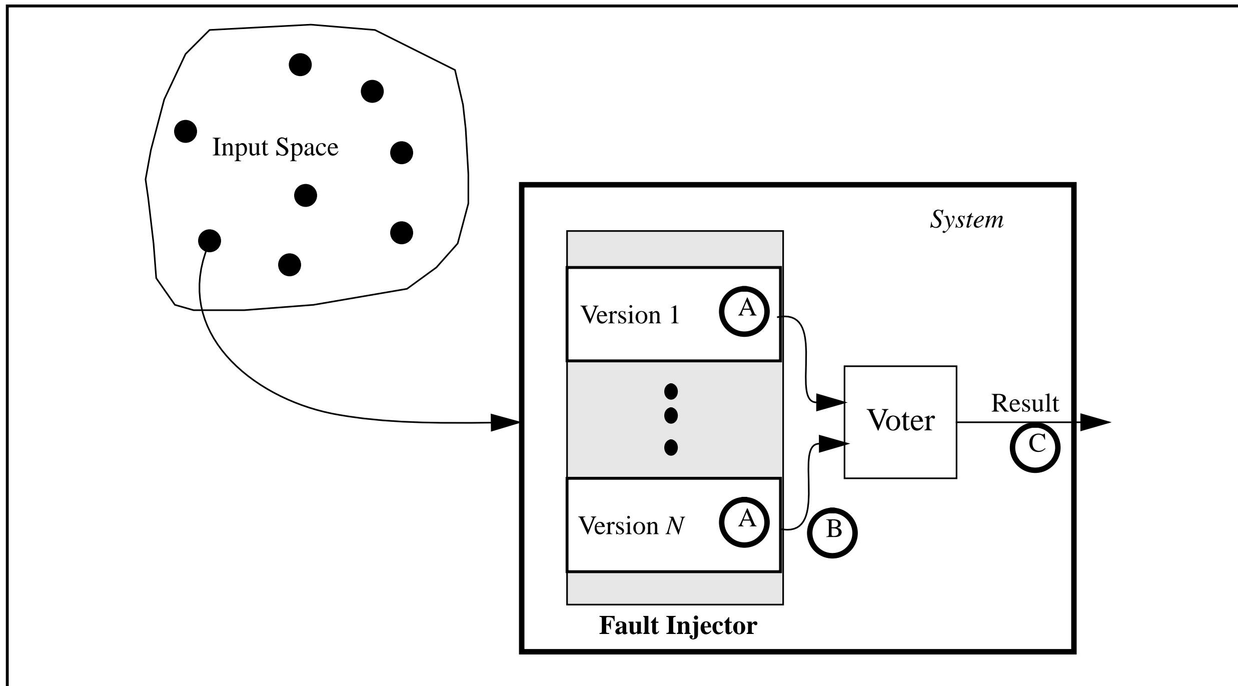

Figure 2: *N*-version system with simulated faulty versions.

4.2 Implementing fault injection

The process of fault-injection involves injecting anomalies into a piece of software using *instrumentation*. Instrumentation is the process of inserting code into the code that is being analyzed, and then compiling and executing the modified (or *instrumented*) software.[1] Code instrumentation can be accomplished in one of two manners: *intrusively* or *non-intrusively.* During intrusive instrumentation, code is added to the software being analyzed and then compiled in. Non-intrusive instrumentation involves executing extra code outside of the program being analyzed, with communication hooks into the program providing access to the state of the program so that its behavior can be observed. Because of the communication that must occur, the term "non-intrusive" is only partially accurate.

In this paper, the anomalies that will be simulated will include:

1. There are methods that employ interpreters to avoid the compilation step we cite, but they are too primitive to be of any real interest.

1. those that can arise from code defects, which are caused by programming errors or design defects. These anomalies occur at locations **A** in Figure 1; and
2. failures exiting a software version, as shown at locations **B** in Figure 1.

The anomalies simulated through fault injection in point 1 above are used to observe how flaws in multiple versions can result in common-mode failures. Anomalies simulated in point 2 above will be used to predict how component failures can affect the output of the voter at location **C**. The likelihood that a system will experience a common-mode failure is different than the question of whether the voter is tolerant to certain classes of common-mode failures. Section 4.3 looks at the first of these two issues; Section 4.4 addresses the latter.

4.3 Simulating faulty components

In Figure 2, the highlighted portion of the system is the actual code, which is where instrumentation will be used 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 locations **B**, the interface between the components and the voter. Since we cannot know if there are defects in the versions, we will simulate defects via fault injection. To simulate faulty components, instrumentation is employed that injects anomalies using potentially two methods: *mutation* and *state perturbation*. Mutation changes the syntax of existing code statements. State perturbation usually adds code to modify the program states created by the original code. Explanations for how to implement these two approaches can be found in [15] and [12]. The work described in this paper uses state perturbation, though the algorithms do not preclude mutation.

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: $\frac{n!}{(n-r)! \cdot r!}$. For an N-version system, there are $\sum_{r=1}^{N} \frac{N!}{(N-r)! \cdot r!}$ 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-point common-mode failures is presented in Figure 3.

The algorithm is shown for three (3) versions for clarity, without loss of generality in *N* versions. This algorithm provides warnings each time the output satisfies the definition for a “single-point” common-mode failure. A “single-point” 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

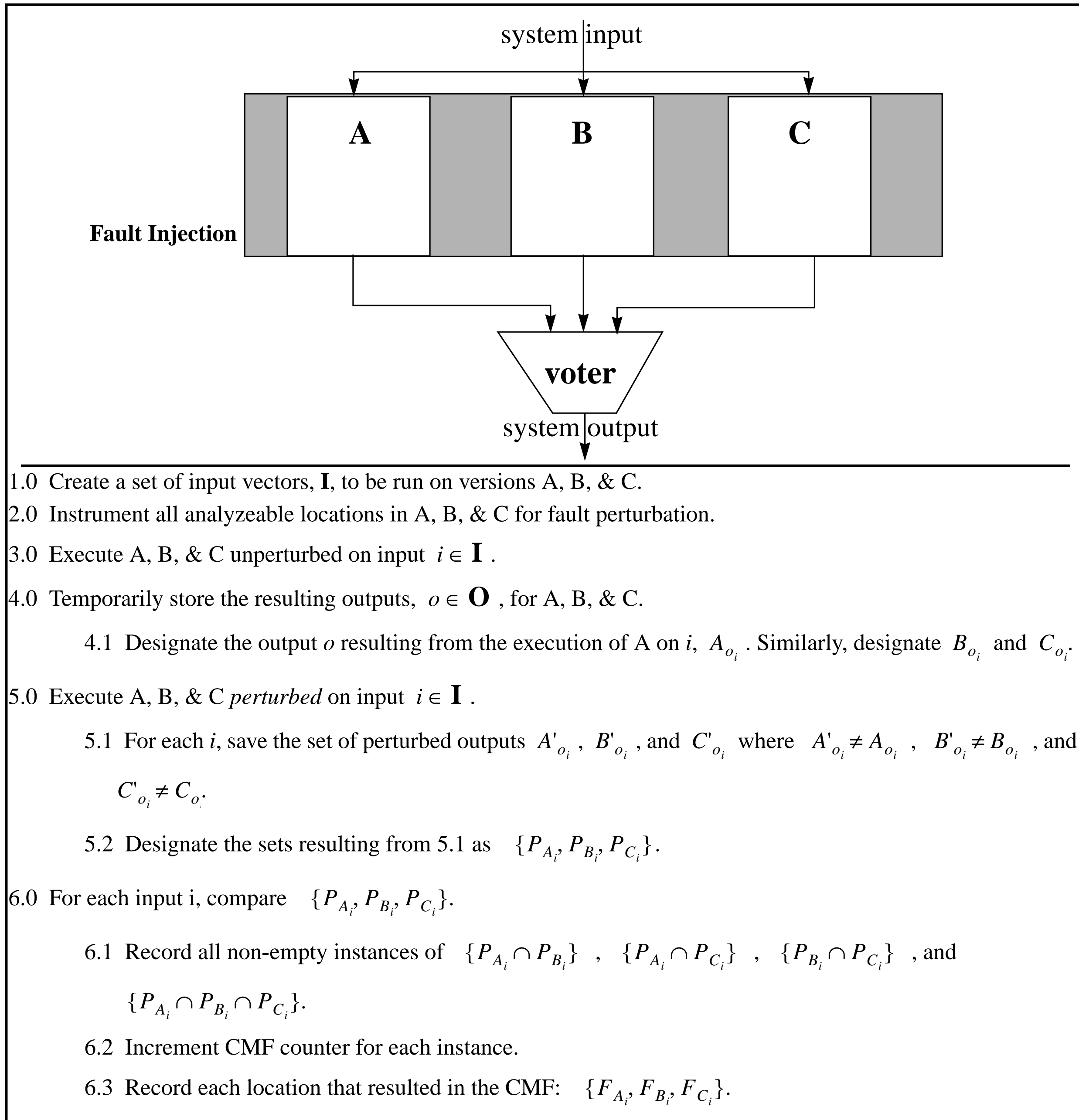

Figure 3: Algorithm for detecting the presence of common-mode failures in multiple version programming.

set of inputs on all versions in two different phases: the unperturbed run and the perturbed run. In the unperturbed 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 [15]. If the output from the perturbed run is different from the unperturbed 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 locations in all versions for each input used. A comparison of all perturbed outputs for a given input is performed to determine if a common-mode failure resulted. A common-mode failure is counted when the perturbed outputs are identical and different from the unperturbed outputs for a given input.

4.4 Simulating Component Failures

The algorithm just presented provides a method for predicting how likely it is that code defects can cause "single-point" common-mode failures. Now, a technique for predicting how likely component failures are to affect the results of the voter is presented. There are two classes of multiple-version failures that are of interest to fault-tolerant systems:

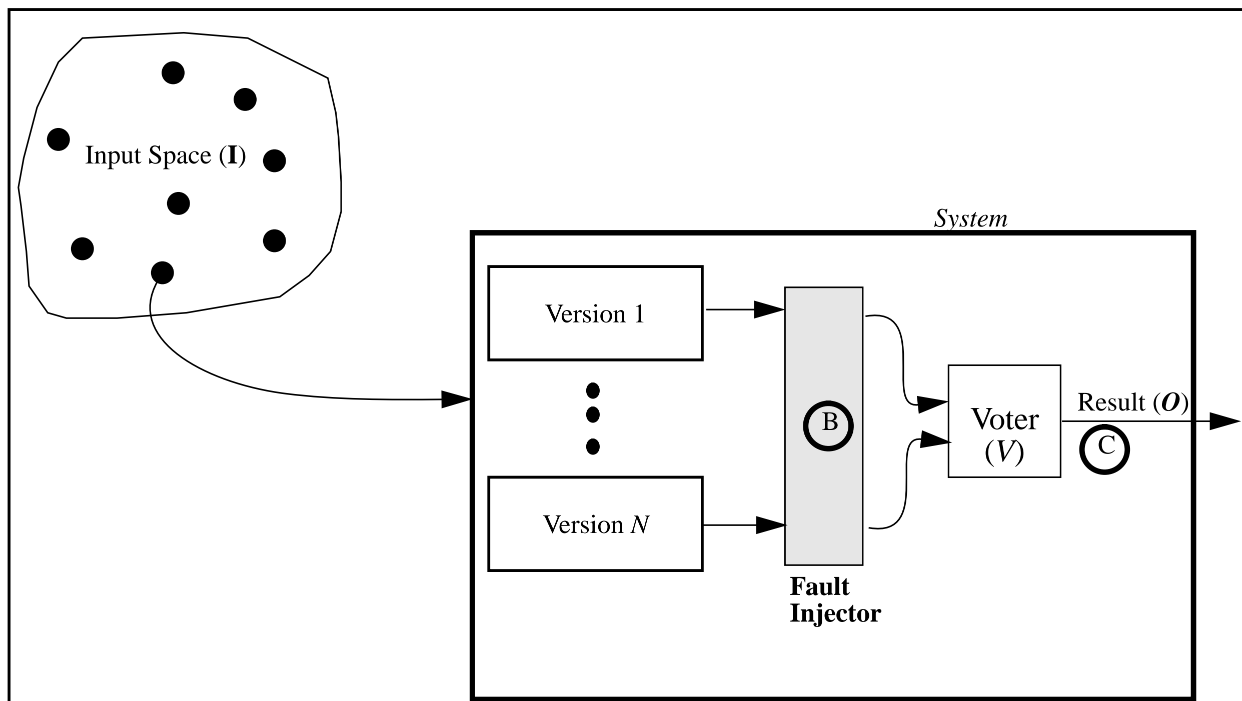

Figure 4: *N*-version system with simulated component failures

1. common-mode failures, and
2. non-common-mode failures.

Figure 4 shows the point at which fault injection instrumentation can be used to simulate component failures, which are simply failures that have exited the versions. The goal of this analysis is to determine if problems at locations **B** can propagate to location **C** in Figure 4. Our approach applies perturbation functions to the outgoing data from fixed combinations of versions.

In a majority voting scheme, if $\left\lceil \frac{N}{2} \right\rceil$ versions produce the same output, that output will be the chosen by the voter. So to address the survivability of the voter, we simulate failures coming out of $\left\lfloor \frac{N}{2} \right\rfloor$ or less versions—which constitutes the minority. If the analysis shows that these simulated failures change the result of the voter, then a warning of voter unreliability is produced.

An algorithm for analyzing the reliability of the voter for a multi-version system is shown in Figure 5. Refer to Figure 4 for the notation used in the algorithm of Figure 5. For the case of three (3) or less versions, the analysis can still be performed, however the fault perturbation is performed on a single version. A key parameter in this algorithm involves deciding what "identical" value will be used as the replacement value for the outputs of the versions in the combination. For example, should a random output be chosen and assigned to the combination outputs? Or should the mean of the outputs of the versions be computed and assigned to the

1.0 Create a set of input vectors, **I**, to be run on all *N* versions.
2.0 For a sampled set of $i \in \mathbf{I}$, do:
- 2.1 Execute all *N* components on inputs $i \in \mathbf{I}$, producing results $O_i \in \mathbf{O}$ from *V*.
- 2.2 For each combination of **X** components ranging from 2 to $\lfloor N/2 \rfloor$, do:
  - 2.2.1 Execute all *N* components on input $i \in \mathbf{I}$, but perturb the outputs from **X** in an identical manner to produce O'_i.
  - 2.2.2 If $O_i \neq O'_i$, then increment *counter*.

3.0 Provide 'WARNING' if $counter > 0$

Figure 5: Algorithm for analyzing the reliability of the voter for an *N*-version system

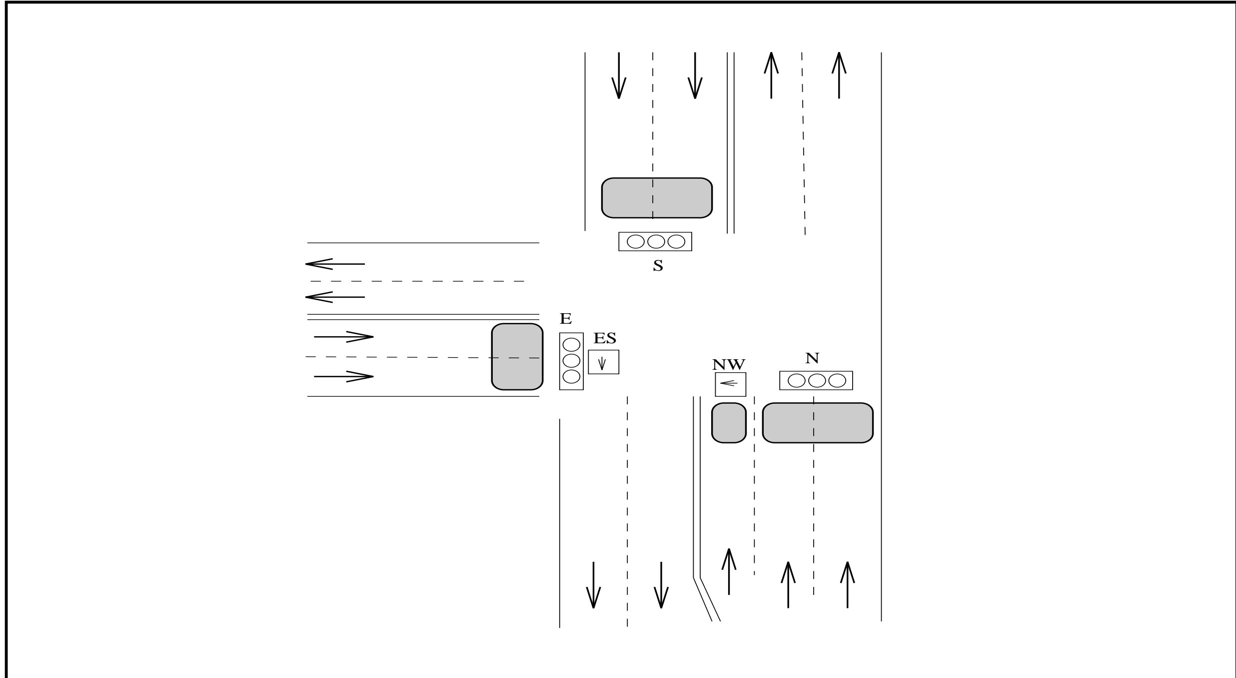

Figure 6: Traffic controller application

versions. One method would be to randomly select a version's output, perturb that value, and then substitute this corrupted value in as the result for each version in the minority combination. This method will assess the voter's survivability to simulated common-mode failures in a minority number of components.

The algorithm for non-common-mode failure is identical to the algorithm in Figure 5, except that the outputs from **X** in Step 2.2.1 are perturbed in a random manner rather than identically. This algorithm will assess the survivability of the voter to component failures that do not necessarily satisfy the definition for common-mode, yet still pose reliability problems for *N*-version systems. Clearly, a voter uses all results when deciding, regardless of whether the input to the voter is correct or not. What we want to be able to know is how often incorrect results in a minority combination of components cause the voter to make the wrong decision.

5. Experiment

In order to validate the usefulness of the algorithms, a prototype *N*-version analysis tool has

been developed and applied to an *N*-version fault-tolerant system. 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. 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 6 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 6 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

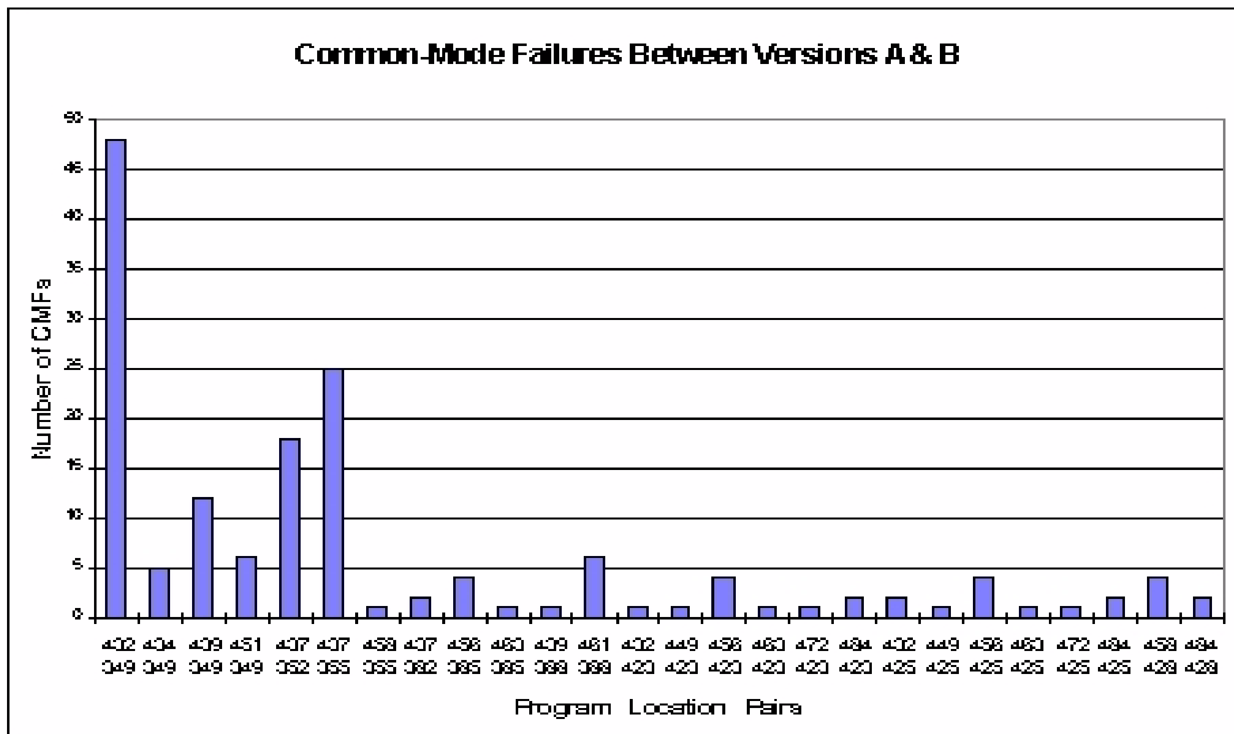

Figure 7: Observed common-mode failures between versions A and B

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.

In order to provide software developers a means for improving the fault tolerance of the system under analysis, the program locations where common-mode failures occurred (Step 6.3 in Figure 3) are presented together with their distribution in Figure 7 and Figure 8. The analysis reveals the potential vulnerable locations in the source code for the three versions where common-mode failures might originate. The program locations where common-mode failures were detected along with the frequency of detection between versions *A* and *B* are shown in Figure 7. By examining this histogram, the problem areas in the code can be quickly assessed. The most frequent common-mode failures in location pairs (*B*,*A*) were (432,349), (437,355), and (437,352). Figure 8 shows a similar diagram for common-mode failures occurring between versions *A*, *B*, and *C*. The critical locations in versions (*C*,*B*,*A*) for common-mode failures are (932,437,352), (932,437,352), and (964,432,349). 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

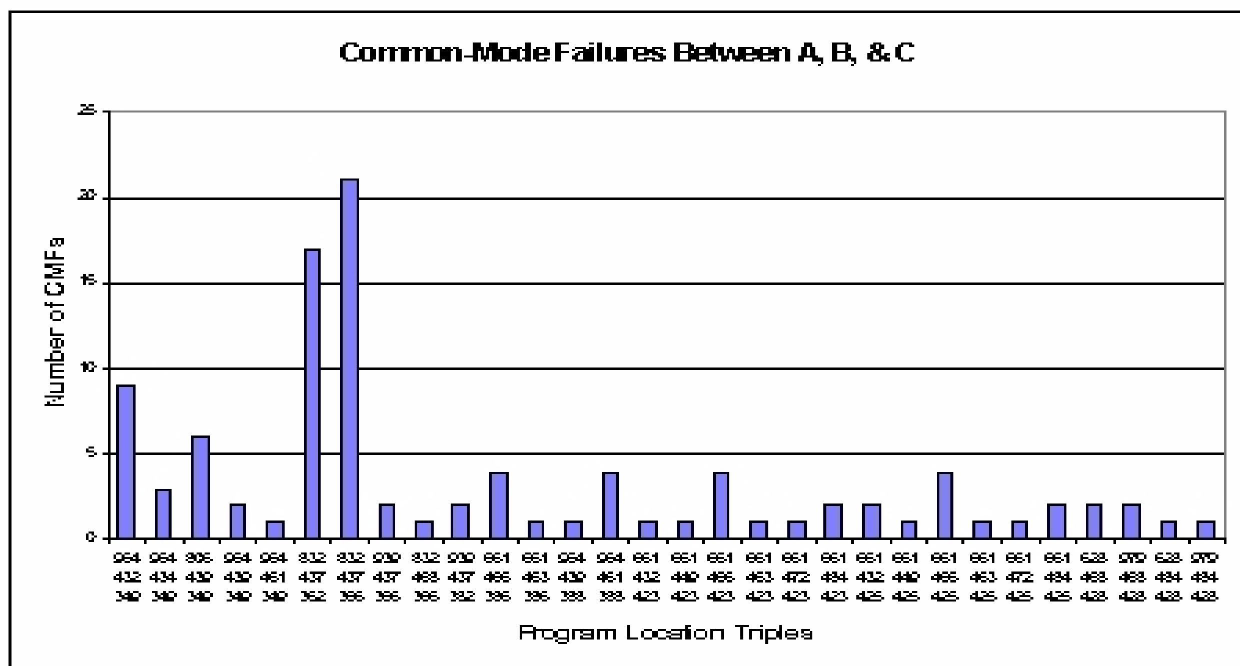

Figure 8: Observed common-mode failures in versions A, B, and C

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.

The analysis results presented in this paper are an initial application of the common-mode failure algorithm presented in Figure 3. Further research will assess the survivability of the voter to both common-mode and non-common-mode failures of components. The potential for decreasing the common-mode failure likelihood by swapping out one version for another will also be explored.

6. Conclusions

We described an approach for deciding how likely redundant components are to result in common-mode failure. If a common specification is used for developing redundant components, then common-mode failure cannot be automatically dismissed, and may actually be engendered [9]. It is intuitive that supposedly equivalent components that were built at different times by different organizations and not from the same specification (although the original specifications might be functionally equivalent) would possess a reduced likelihood of common-mode failure. But even if untrue, the approach described here can aid in deciding how worrisome common-mode failure is for a particular *N*-version system.

The recent loss of the $8 billion Ariane 5 rocket can in part be attributed to redundancy [10]. Two redundant inertial reference systems (SRI) in the Ariane rocket (that have identical computers and identical hardware) both suffered from the same operand error within 72 milliseconds of each other. After the failure of the second system occurred, incorrect (corrupted) data was sent to the main On-Board Computer (OBC), which then calculated an angle of attack that was so severe that the boosters became dislocated from the main stage, which then caused the self-destruct mechanism of the launcher to execute. Because both SRIs were executing the same code, there was no diversity between versions, and hence this was the most trivial form of guaranteed common-

mode failure.

When a developer cannot decide between competing components in the component-based marketplace that we someday envision, traditional parallel construction with multiple components is an alternative *if* the developer can be given confidence that the likelihood of problems resulting from common-mode failure is low. Although this is more expensive than purchasing a single component, purchasing several components and placing them in parallel may still be more cost effective than building a single component from scratch.

Acknowledgments

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

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

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

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

[14] 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.

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