Simulating Specification Errors and Ambiguities in Systems Employing Design Diversity

Jeffrey Voas
Reliable Software Technologies
Suite 250
Research Division
21515 Ridgetop Circle
Sterling, VA 20166
jmvoas@RSTcorp.com

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

Abstract

This paper looks at methods for predicting how likely it is that an *n*-version software system will suffer from common-mode failures. Common-mode failures are frequently caused by specification errors, specification ambiguities, and programmer faults. Since common-mode failures are detrimental to *n*-version systems, we have developed a method and a tool that observes the impact of simulated specification errors and specification ambiguities. These observations are made possible by a new family of fault injection algorithms designed to simulate specification anomalies. As a side-benefit, this analysis also provides clues concerning which portions of the specification, if even slightly wrong or misinterpreted, will lead to identical failures by two or more versions. This suggests which specification directives have the most impact on the system's functionality.

Keywords

n-version programming, fault injection, common-mode failure, specification

Biographies

Jeffrey Voas is a Co-founder and Chief Scientist of Reliable Software Technologies. Voas has coauthored a text Software Assessment: Reliability, Safety, Testability (John Wiley & Sons, 1995). Voas is currently co-authoring a second text "Software fault-injection: inoculating programs against errors", due to be published by Wiley in 1997.
Lora Kassab graduated from the College of William and Mary in May 1997 with an MS in computer science. She is a computer scientist in the Information Technology Division at the Naval Research Laboratory in Washington D.C. Her interests are computer security, testing, and distributed systems.

*This work was performed when Kassab was a graduate student at the College of William & Mary.

1 Introduction

Software systems suffer from a variety of problems: incorrect requirements and specifications, programmer faults, and faulty input data. These problems can cause software to exhibit undesirable behavior, including crashing, hanging, or simply just producing wrong output.

At best, *software testing* reduces programming faults, but software testing can do little for specification errors or corrupt input data anomalies. *Formal methods* are geared toward thwarting large classes of inconsistencies and ambiguities in specifications and can even detect programmer faults. But even formal methods can be misapplied or fail to detect specification errors.

Here, we will provide a set of algorithms that are similar to traditional software testing approaches, but instead of testing the software, they test the resilience of the software to specification errors. New software fault injection algorithms will be introduced here. These algorithms provide insight into how a system will behave if the specification is erroneous.

Previously, we have published results from using fault injection in applications that did not employ design diversity [9]. In this paper, we are focusing on ways for predicting whether the identical failure by two or more parallel versions is possible if the versions' faults can be traced back to a common specification error or specification ambiguity. More specifically, we are interested in how identical failures by *diverse* versions affects an *n*-version system. (We have also done similar work that deals exclusively with using fault injection to simulate random anomalies in diverse versions [10], but that is not our focus here.)

In our approach, each version is forced to experience an incorrect internal computation that can be mapped back to a common specification error. By showing that common-mode failures are infrequent after the specification is forcefully mutated in a manner that simulates specification errors and ambiguities, we can plausibly argue for placing diverse software versions in parallel (with the use of a voter).

In *n-version programming*, different software versions, written to the same specification but developed independently, execute in parallel (See Figure 1). It is imperative that there is *no* communication between the teams responsible for developing the different versions. Quarantining the different teams is essential such that misunderstandings from one team do not affect the understanding of other teams. But quarantining teams is not always enough–uncorrelated faults in distinct versions can lead to identical failures.

Although design diversity thwarts certain types of faults, it does not thwart all faults [6, 2]. Knight and Leveson [6] demonstrated that different programmers can make the same logical error. An additional result from Knight and Leveson demonstrated cases where different logical errors yielded common-mode failures in completely distinct algorithms and in different parts of similar algorithms.

For this paper, we need to differentiate three different types of failures: (1) failures of versions that satisfy the definition for common-mode failure, (2) non-common–mode version failures, and (3) voter failures. *Common-mode failure* occurs when two or more identical software versions are affected by faults in exactly the same way [4]. More specifically, common-mode 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 all outputs are identical for all possible inputs for this combination of versions. Thus, if two or more versions respond to all inputs in the same way, and there is at least one input that causes this set of versions to fail, then common-mode failure has occurred. *Non-common–mode failures* are simply version failures that cannot be correlated with other versions

and thus do not satisfy the definition of common-mode failure. *Voter failures* occur when the voter makes a wrong decision because of the inputs it received.[1]

It is reasonable to further classify the severity of common-mode failures, since certain classes of common-mode failures are more likely to trip up the voter than others. Here, *severity* is equal to the number of versions that are in agreement on the "wrong output." For example, a common-mode failure between two versions is less likely to result in voter failure for a 9-version system than for a 3-version system. This explains why specification ambiguities that affect only a handful of the versions can be benign if the system is sufficiently large (in terms of the number of versions).

Many people have written off *n*-version programming as a dead approach to attaining high integrity software because of the *n*-version problem. But to our amazement, *n*-version programming is alive and well in several different safety-critical domains, and it is particularly popular outside of the United States. For example, Airbus uses diverse software versions for the A320/A330/A340 electrical flight controls systems [7, 1]. (But *n*-version programming cannot fix everything—the *n*-version Airbus flight control system still has a fatal flaw that has not been fixed or thwarted by any of the *n*-version capabilities [5].)

In the United States, the FAA's position, based on industry's feedback, 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 prescribed levels of assurance but will not substitute for other requirements [3]:

> 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.

2 Software Fault-injection

A significant amount of research has focused on methods to detect and eliminate errors earlier in the software life-cycle *e.g.*, prior to implementation. Even so, errors related to misunderstandings, ambiguities, or faulty assumptions will find their way into specifications. This is inevitable.

Many people have spent careers trying to develop techniques that eliminate all program errors. (Although laudable, the fact remains that not all errors need elimination: only those errors that have nasty consequences.) Because certain errors can be tolerated, we wish to isolate those classes of specification errors that if implemented in an *n*-version system, will cause the voter to make a bad choice. For now, a "bad" choice will be the same as a "different" choice (i.e., the specification error had not been programmed).[2] By demonstrating that particular classes of specification errors and ambiguities are unlikely to impact the voter's decision, increased confidence is warranted for employing redundant, diverse versions in parallel.

[1] We will ignore the possibility of faulty voters.

[2] Because we have built other utilities to detect different types of internal states and output events, "bad" could also be defined as other failure classes (such as "unsafe").

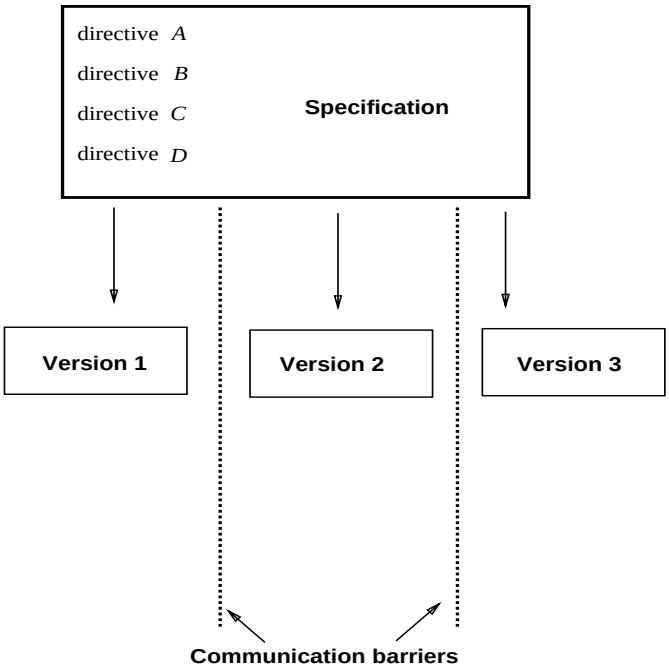

Figure 1: One Specification and Three Independent Versions.

Let's begin by considering the simple *n*-version system in Figure 2. This figure illustrates the traditional architecture of an *n*-version system that is composed of *n* independent versions and a software voter *V*. *i* is the input value fed to each version in parallel. *V* determines which output to release from the *n* versions. We employ software fault injection to determine whether identical programmer faults in two or more versions will cause identical version failures. If so, then we know that the voter will also succumb to the identical programmer faults. And if the faults have a common root cause such as a faulty specification, then we know which classes of specification errors must not occur.

The process of performing software fault injection involves adding code to the code under analysis; the added code is called *instrumentation*. The modified program is then compiled and executed. The instrumentation is involved in either injecting anomalies or observing the impact of the anomalies.

There are many different types of specification-based anomalies that could be simulated using fault injection. The key classes of specification-based anomalies that should be simulated via fault injection methods are:

1. Those anomalies that can arise from actual *specification errors*, where if each programmer implements the specification correctly, then each version will perform some internal computation differently. (Whether this forces the versions to produce an incorrect output is another question.)

2. Those anomalies that can arise from *specification ambiguities*, where at least one programmer implements some directive $\mathcal{C}$ in manner $\mathcal{C}_a$, and the remaining development teams implement it in a semantically different manner, $\mathcal{C}_b$. (Again, whether this forces the versions to produce an incorrect output is another question.)

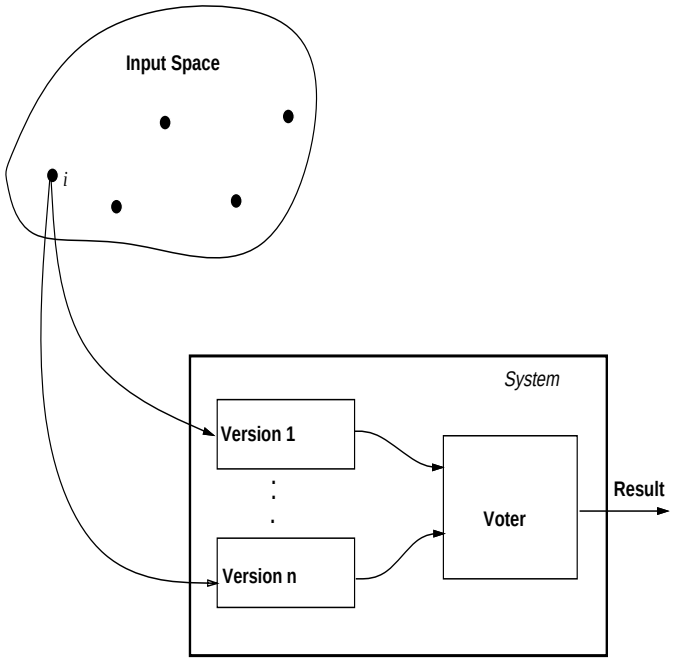

Figure 2: *n*-version System with Voter and Inputs.

2.1 Analysis Assumptions and Requirements

This approach is a *behavioral analysis* technique. The appropriate time to apply this technique is after the *n*-version system is built and ready for deployment. This approach assumes that the following are available:

1. Version outputs can be captured before they enter into the voter.
2. Source code to the versions is available.
3. A specification S from which we can isolate specific directives, $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, ...

In addition, the approach does not require the following be true, but it is prudent that these assumptions are also true.

1. The versions are reliable and have been well tested. This assumption is only included to justify our simulating classes of anomalies that are not representative of *programmer faults*. Also, if the versions are still suffering from reliability problems, then the versions are not yet ready for this analysis.
2. The voter is reliable. (We do not care which voting approach is implemented, as long as it is implemented correctly.)

2.2 Simulating a Faulty Specification for *n*-version Systems

Intuition suggests that wrong *n*-version specifications will always cause a *voter failure*. But when the voter does not change its vote after common specification errors are injected into the versions,

we must wonder why. Once we find those directives that if modified cause the voter to switch its vote, the question we want answered is "are we sure those directives are correct'?"

Specification errors are simply faulty specification directives that define internal computations. Simulating specification errors provides a prediction of how tolerant the voter will be to real specification errors concerning the *n*-versions.[3]

As an example, suppose that the specification has a directive to the programmers that says that the ALTITUDE variable is a function of f_1 + f_2. Suppose this function is wrong, and that it should be f_1 - f_2. Unless this if found before the code is programmed, this specification error will likely find its way into the versions. After all, the specification is usually the final authority on correctness.

Here, fault injection will be used to force a corrupt ALTITUDE value in each version on each test case. This is done in a manner reflective of a *common specification error*. This involves finding the appropriate source-code statements in the versions where the ALTITUDE computation directive is implemented, and then injecting the common anomaly into each implementation.

In fault injection, *perturbation functions* are the source code instrumentation utilities that inject *data state mutants* [8]. Data state mutants are corruptions to the values that particular variables have internally as the software executes. Developing new perturbation functions that simulate common specification errors in multiple versions was a key research task of this project.

For this example, a perturbation function will force the value of ALTITUDE to be reduced or increased by an equal amount, $\pm\Delta$, in all versions. If Δ were -30, then 30 would be subtracted from the value that each version computed for ALTITUDE = f_1 + f_2. It may well be that different versions have different values after executing the statements for this directive, and if so, we wish to retain that natural diversity. So for $\Delta = -30$, if one version had ALTITUDE equal to 40, then after fault injection, the value will be 10. If another version had ALTITUDE equal to 500, then it will get a value of 470.

After much thought, we decided that it is important to retain existing diversity in versions. We could have taken a different approach and selected a unique value *Q* (that no version had) and then given each version an ALTITUDE value of *Q*, But instead, we would do the following: ALTITUDE = f_1 + f_2 $\pm$ Δ.

The algorithm that we will employ for a single specification error to a numerical data type follows:

<u>Algorithm for Simulating a Specification Error:</u>

1. For some test case *i*, run the *n*-version system and store the output from each of the *n* versions in an array *A* of size *n*. Let *O* denote *V*'s decision based on the *n* version outputs in *A*.

2. Select a computation directive $\mathcal{C}$ from the specification *S* that is expected to be implemented in each version. (We will assume that $\mathcal{C}$ is implemented in each version, as we expect that each version is already highly reliable in isolation.)

3. For *i*, apply the standard perturbation function defined in [8] ($\pm$ 50% of current value as range for selecting new value)) to the result computed from the implementation of $\mathcal{C}$ in *one*

[3] One class that we cannot simulate easily here is incomplete specification errors, and thus in this paper, we will not address this class.

randomly selected version *e* from the *n*-version set.

4. In *e*, calculate the offset ($\pm\Delta$) between what the original result from $\mathcal{C}$ was and what the new value from $\mathcal{C}$ is after the internal value is perturbed. Note that Δ is a function of *i*, *e*, and the perturbation function.

5. For the other $n-1$ versions, the offset of $\pm\Delta$ is forced into the internal result computed by their implementation of $\mathcal{C}$.

6. Execute all *n* versions on *i* using the $\pm\Delta$-based perturbation function, and collect the output from the voter, *O'*, for this *i*.

7. If $O \neq O'$, then the voter was not tolerant to the specification error affecting $\mathcal{C}$. Also, if there exists *x* versions ($x \geq 2$) whose outputs equal *K* after fault injection but whose outputs in *A* did not equal *K*, then a single-input common-mode failure of severity *x* has occurred.

8. Perform the previous steps for a set of test cases and for each $\mathcal{C}$ keeping a count of the number of failures.

The possibility exists that some implementation of $\mathcal{C}$ is not executed when *i* is selected. If this occurs for all versions, then ignore *i* and select another input. Otherwise, perform the algorithm as explained, and perform the offset injection (in Step 5 of the algorithm) in those versions where $\mathcal{C}$ is exercised. As the number of versions increases that do exercise $\mathcal{C}$ for *i*, the likelihood that $O \neq O'$ decreases for this *i*. And if some versions are executing different calculations than their counterparts, then it is certainly possible that the voter will not produce the desired results for reasons other than identical programmer faults. (An example here would be specification ambiguities, which we will discuss in the next section.)

This algorithm simulates the situation of a precise "off-by-something" error ($\pm\Delta$) in directive $\mathcal{C}$ affecting the data states in each version at the appropriate place. By using an offset, and not just forcing a constant value into each version, we do not disturb other "natural" diversity that may already exist in the different versions. For a fixed $\mathcal{C}$, this algorithm will be applied for many different *i*'s and different Δ's. This suggests how sensitive voter *V* is to $\mathcal{C}$ in *S*.

Note that "off-by-something" errors are not the only specification errors that can be simulated. For example, we could have an "off-by-some-percentage" that simulates a multiplier effect. So instead of ($\pm\Delta$), we might want $\times K$, where *K* is a constant in $(0, \infty]$.

2.3 Simulating an Ambiguous Specification for *n*-version Systems

In order to simulate an *ambiguous specification* directive, a directive that can be interpreted in several ways, we only need to make a small change to the previous algorithm.

Algorithm for Simulating a Specification Ambiguity:

1. For some test case *i*, run the *n*-version system and store the output from each of the *n* versions in an array *A* of size *n*. Let *O* denote *V*'s decision based on the *n* version outputs in *A*.

2. Select a directive $\mathcal{C}$ from the specification *S* that is expected to be implemented in each version.

3. For *i*, apply the default perturbation function defined in [8] to the result computed from the implementation of $\mathcal{C}$ in *one* randomly selected version *e* from the *n*-version set.

4. In *e*, calculate the offset ($\pm\Delta$) between what the original result from $\mathcal{C}$ was and what the new value from $\mathcal{C}$ is after the internal value is perturbed. Note that Δ is a function of *i*, *e*, and the perturbation function.

5. For some random number *r* (where *r* is in $[0..n-2]$), randomly select *r* versions, none of which can be *e*.

6. Take this set of *r* versions, find in each version where $\mathcal{C}$ is implemented, and force an offset equivalent to $\pm\Delta$ into the internal result from those implementations of $\mathcal{C}$.

7. Execute all *n* versions of the system (whether they had $\pm\Delta$ applied to their program states or not), and collect the output from the voter, *O'*, for this *i*.

8. If $O \neq O'$, then the voter was not tolerant to the specification ambiguity affecting $\mathcal{C}$ in the $r+1$ versions. Also, if there exists *x* versions ($x \geq 2$) whose outputs equal *K* after fault injection but whose outputs in *A* did not equal *K*, then a single-input common-mode failure of severity *x* has occurred.

9. Perform the previous steps for a set of test cases and for each $\mathcal{C}$ keeping a count of the number of failures.

This algorithm assumes that a computational directive $\mathcal{C}$ could be interpreted in one of *two* ways. This algorithm could be extended for 3 or more different interpretations of $\mathcal{C}$. This algorithm randomly selects which versions will be assigned the first interpretation and leaves the other versions alone.

3 Experiment

Our experimentation involved using five different versions of a controller for managing the traffic lights and turn arrows at a particular intersection. Admittedly, this result is for a toy example; we have tried to attain a real safety-critical *n*-version system that is written in C or C++ but had no success.

The original specification for this system was written by Adam Porter at the University of Maryland, and was modified and given to students at the College of William and Mary. The reason that modifications were made was so that manual correlation between specification directives and source code computations could be easily made. (Note that these versions were not intended to be part of a true *n*-version system, however team independence was mandated.)

In this specification, traffic can move going northbound (N), southbound (S), east bound (east to north (E), east to south (ES)), and north to west (NW). There is a traffic light controlling all northbound, 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. Figure 3 depicts the intersection.

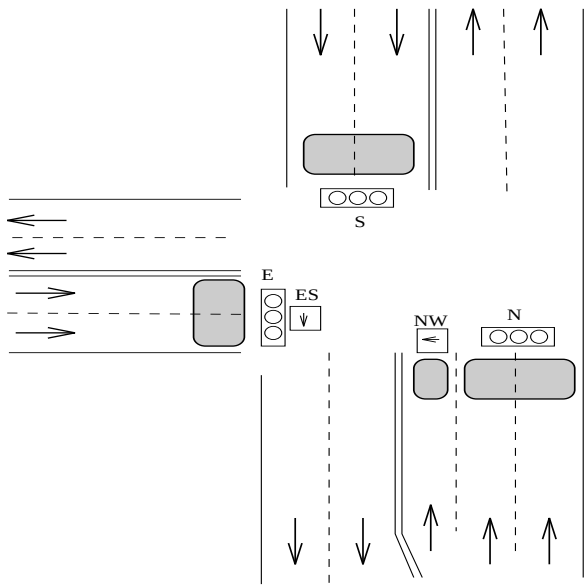

Figure 3: Intersection.

There are 4 sensors under the roadway: one for all eastbound lanes (E), one for all southbound (S) lanes, one for the northbound (N) lanes, and one for the north to west (NW) turning 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. Each software version receives the sensor signals as input and then generates the appropriate traffic light controls (outputs) for all lanes at the intersection. The outputs indicate the color (GREEN, YELLOW, or RED) of every traffic light to reflect the current traffic flow.

This experiment employed the algorithm for simulating a specification error. In the specification, there were 11 key calculations that the programmers were instructed to handle inside the traffic light software:

1. **North Green** Give traffic headed North the green light.

2. **North Yellow** Give traffic headed North the yellow light.

3. **North Red** Give traffic headed North the red light.

4. **East Green** Give traffic headed East the green light.

5. **East Yellow** Give traffic headed East the yellow light.

6. **East Red** Give traffic headed East the red light.

7. **South Green** Give traffic headed South the green light.

8. **South Yellow** Give traffic headed South the yellow light.

9. **South Red** Give traffic headed South the red light.

10. **Northwest Arrow** Give traffic heading North the turn arrow to go West.

11. Southeast Arrow Give traffic heading East the turn arrow to go South.

In our experiment, we executed twenty test suites on the five version system. Each test suite is a chain of events at the intersection, where an event is either a single sensor emitting or multiple sensors emitting simultaneously. Since there were 11 $\mathcal{C}$s for which specification errors could be simulated and a total of 510 inputs from all of the test suites, the *n*-version system was executed for a total of 5610 test cases. For each of the 5610 test cases, we collected the results generated from the five versions without any perturbations to verify that there was a majority ruling on the output of the system.

If there was not a majority when the voter compared the outputs from each of the versions, then the test case was never executed with perturbed specification directives. The test case was omitted simply because the versions were not able to agree on the output under "normal" circumstances. In most existing *n*-version systems, the individual versions are thoroughly tested before integration into the *n*-version system, and thus the likelihood of undetected programmer faults is slim. The versions used in this experiment were not tested as thoroughly as in real safety-critical *n*-version systems, however, there was a majority agreement from the five versions for 69% of the test cases.

If there was a majority output from the versions, then we executed the five version system for every specification directive that was selected to be perturbed. Of these test cases, 23% resulted in the voter making a different decision. The results from the analysis of applying the algorithm for simulating a specification error are summarized in Figure 4: the indices along the x-axis are the eleven specification directives and the y-axis corresponds to the number of single-input common-mode failures that occurred. Note that the specification directive that caused the voter the most problems was the directive handling the red light for Eastbound traffic; it caused the voter to fail 164 times.

The reason for this appears to be complexity in the specification. There were test scenarios where events at traffic sensors occurred simultaneously. For instance, two cars might request a Northwest turn and a Southeast turn from the controller at the same instant. These more complicated input events confused several of the programming teams, and it appears to have made the East Red directive extremely sensitive to errors. Also, the test cases that were employed exercised the East Red directive more frequently. Hence if you have access to an operational profile, that information should be used for even more accurate predictions concerning directive sensitivities.

Had this been a real system and had simultaneous events like this occurred frequently during daily operations, then this analysis would have isolated the specification directives that were more capable of making the traffic intersection unsafe. This suggests which specification directives need greater attention during validation.

4 Summary

This paper has looked at a method for predicting how likely it is that an *n*-version software system will trip up the voter when the specification is defective. By knowing the sensitivity of voters to different specification anomalies, we can isolate those parts of the specification must be "solidly" correct. If those portions of the specification that need to be correct are not, our approach predicts that common-mode failures are more likely to occur.

We have focused on systems that employ design diversity. There is no reason that fault injection cannot be used to simulate specification anomalies for single-version systems.

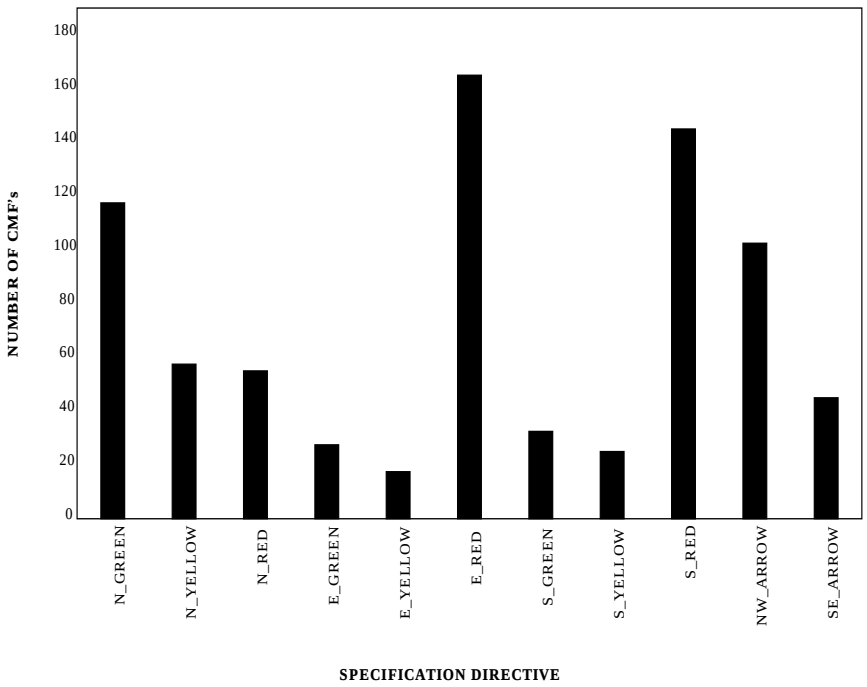

Figure 4: Specification directives and single input common-mode failures

It is known that earlier elimination of errors in the life-cycle is cost-prudent. Little research has ever been done to our knowledge that targeted eliminating only those specification problems that will have a detrimental impact on the system. The approach we have outlined here is a first step in that direction.

We admit that it would be preferable to have the results of this analysis much earlier in the life-cycle. But at this point, we do not know how to achieve that.

Acknowledgements

This work has been partially supported by DARPA Contract F30602-95-C-0282 and NIST Advanced Technology Program Cooperative Agreement No. 70NANB5H1160. The authors thank Prof. Adam Porter (University of Maryland) for supplying us with the original specification used in the experimentation. Prof. Porter also provided Figure 3 to us. The authors thank Frank Charron for his help in building the tool that was needed for the experimentation.

References

[1] D. Briere and P. Traverse. AIRBUS A320/A330/A340 electrical flight controls - a family of fault-tolerant systems. In *FTCS 23*, pages 616–623, Toulouse, June 1993.

[2] 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.

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

[4] B. W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems*. Addison-Wesley, 1989.

[5] Peter Ladkin. A320 Flight-Control Computer Anomalies *Software Engineering Notes Risk Forum* 18(78).

[6] 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.

[7] P. Traverse. Dependability of digital computers on board airplanes. In *DCCA 1*, Santa Barbara, CA, August 1989.

[8] J. Voas. PIE: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Eng.*, 18(8):717–727, August 1992.

[9] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman. Predicting How Badly 'Good' Software can Behave. *IEEE Software*, July 1997.

[10] J. Voas, A. Ghosh, F. Charron, and L. Kassab. Reducing Uncertainty about Common-mode failures. To appear in *Proceedings of the International Symposium on Software Reliability Engineering*, Albuquerque, NM, November, 1997.