ISSN 2754-1118 (Online)
ISSN 2753-6599 (Print)

Reducing the Risk of a Software Common Mode Failure

Rob Ashmore, Mark Hadley and James Sharp

Dstl, Portsdown West, UK

Abstract

We describe the nature of a common mode failure, illustrating why software-based components can be particularly susceptible. A number of historical accidents and incidents are used to demonstrate this is not just a theoretical risk. Previous academic research, existing standards and historical civil air programmes are surveyed. This shows the importance of software common mode failure is recognised, but there is no preferred way of protecting against it. A set of criteria are proposed, which provide a means of assessing protections that have been implemented within a system design. These are used to highlight approaches to protection that are suggested for future air systems (and other safety-related applications).

1 Introduction

In an attempt to increase reliability, system architectures often include replicated items. The desired increase in reliability is based on an assumption, typically implicit, that each item will fail independently. A Common Mode Failure (CMF) is an event that undermines this desired independence.

If the replicated items are running software, then this has the potential to cause a CMF. This potential is heightened by the nature of software: using identical copies of the same software, and providing them with identical inputs, offers no protection against software CMF. This is very different to the situation for mechanical or discrete electronics components where identical items, which fail independently, can provide protection.

We are primarily concerned with protecting against software CMF in future air systems, but note that many of the same considerations apply to other safety-related applications.

Initially, in Section 2, a selection of historical examples illustrate the real-world importance of software CMF. An informal review of key academic literature is provided in Section 3. Treatment of software CMF in a selection of standards (and standards-like) documents is discussed in Section 4. Previous approaches adopted within the civil air domain are briefly described in Section 5. Criteria to assist in evaluating protections against software CMF are listed in Section 6, and are related to potential protection approaches that could be adopted in system designs in Section 7. Closing remarks are provided in Section 8.

2 Historical Examples

Software CMF is not just a theoretical concern. It has been the cause of a number of accidents and incidents, a selection of which are briefly summarised below:

Ariane 5 (Lions 1996): Two SRIs (Inertial Reference Systems[1]) were operating in parallel, with identical hardware and software. A numeric overflow was detected and, by design, the active SRI shut down. The backup SRI had shut down in the preceding time period for exactly the same reason. The lack of a functioning SRI led the loss of the launcher.

NATS System Failure (Walmsley et al. 2015): NATS run a primary System Flight Server (SFS), with a secondary SFS always ready in order to preserve availability. Prior to this incident, recent changes had exposed a latent fault in the SFS software. When an action caused this fault to become a failure, by design the primary SFS shut down. Since it ran identical software, the secondary SFS also shut down. Up to 1900 flights and 230,000 passengers were affected by the resulting outage.

Subsonic / Supersonic Boundary (JSSSC 2010): A front-line aircraft had been rigorously developed, thoroughly tested and dozens were in operational use. As part of an upgrade, a weapons test required extensive telemetry. Analysis showed that all aircraft computers failed and then restarted as the aircraft passed through Mach 1. The aircraft had sufficient momentum and mechanical control that it coasted through this anomaly without the pilot noticing.

Airbus A330-302 (TTSB 2021): On June 14, 2020, China Airlines flight CI202, an Airbus A330-302 aircraft, took off from Shanghai bound for Taipei. At touchdown, the aircraft experienced the quasi-simultaneous failure of the 3 Flight Control Primary Computers (FCPCs), which meant that ground spoilers, thrust reversers and autobrake became unavailable. The flight crew rapidly applied full manual brake to stop the aircraft safely about 30 feet before the end of the runway. The software at the core of this incident had supported more than 8.7 million flight cycles, with no other observations of this type of issue.

These examples show that software CMF can be a significant issue, even in systems that have been rigorously developed and that have a significant amount of service history.

3 Academic Research

3.1 Fault Avoidance

Avoiding the introduction of faults, as far as possible, is the first way of protecting against software CMF. For example, Powell et al. (2011) note that: "*The first defense against design faults is of course to attempt to avoid their occurrence in the first place by applying the techniques of fault prevention (i.e., 'careful design') and fault removal (i.e. 'verification and testing')*".

(Note that, generally speaking, the academic literature uses the term "design fault" to include faults from a number of software development phases, including, requirements, design, implementation, and maintenance.)

[1] SRI is Système de Référence Inertiel, i.e. Inertial Reference System

Fault avoidance typically involves the application of appropriately rigorous software development and assurance standards; for example, RTCA/DO-178C (RTCA 2011).

3.2 Fault Tolerance

Powell et al. (2011) also discuss protection against software CMF under the general heading of tolerance to design faults. The first-level response to tolerating design faults comes in two forms, data diversity and design diversity. Both of these forms aim to prevent identical software being presented with identical inputs, thus introducing some (typically, limited) independence.

Data diversity is focused on inputs to the software, whereas design diversity is concerned with the software itself. Broadly speaking there are three basic approaches (see, for example, Powell et al. (2011), Siewiorek et al. (2004) and Laprie et al. (1990)) to introducing design diversity, these being recovery blocks, n-version programming, and n-self checking.

The following subsections provide further information on approaches to fault tolerance.

3.3 Data Diversity

This type of diversity can be achieved by providing identical software with different inputs, for example from sensors located at different points on the airframe. The key premise behind data diversity is that software problems arising during operational use are subtle and hence only triggered for small parts of the input space (see, for example, Ammann and Knight (1988)). Hecht (1993) collected evidence from a number of sources, including spacecraft avionics and telephone switches, which also supports this view.

However, only a limited amount of diversity can be introduced by providing identical software with different input data. In particular, this data typically comes from different sensors that are sampling the same (or at least a very similar) real-world situation. It is unlikely, for example, that two properly-functioning air data sensors will provide radically different data.

One situation in which wildly different data could be provided is following a sensor failure. The software would be expected to handle this failure, but data diversity, in the form of different sensors providing inputs to different copies of the software, would provide an additional layer of protection. An alternative way of providing protection against this specific type of problem would be checking inputs before they are used and, if necessary, replacing out-of-range values with a known-safe alternative.

In theory, it is possible to create artificial data diversity, for example, by adding noise to input signals. In practice it can be difficult to determine an appropriate way to alter inputs, so that their essential meaning is preserved but they are sufficiently different to provide some protection against software CMF.

3.4 Design Diversity — General

The intent of design diversity is to create versions of software that fail in non-identical ways. This typically involves changes to earlier parts of the software development lifecycle (e.g. design and implementation).

Design diversity typically provides greater protection than data diversity as, for example, it is unlikely that two separate programming teams will have made identical errors. That said, as discussed below, common mode failures can still occur even with non-identical

software. Furthermore, design diversity incurs greater costs, as additional software development is required.

3.5 Design Diversity — Recovery Blocks

Randell introduced the concept of recovery blocks in 1975 (Randell 1975). These were motivated by the observation that it is standard practice to structure a program of any significant complexity into a series of blocks (e.g. modules, procedures, subroutines, methods).

Before a "significant" block is executed, a recovery point is stored. The first alternate block is run and its output is subjected to an acceptance test. If the output is not acceptable then the program is rewound to the recovery point, the second alternate block is executed and its output is tested for acceptance. The process is repeated until either an acceptable output is found or all alternates have been exercised.

Theoretical issues with the recovery block approach include inaccuracies in the acceptance test, and the possibility of interacting tasks causing a domino effect of cascading recoveries. Tomek et al. (1993) also note that it can be difficult to achieve diversity in the alternate implementations of the recovery blocks; this is partly due to their relatively small size, and partly due to the fact that they are forced to operate on identical inputs.

An experimental evaluation of the recovery block approach was conducted by Anderson et al. (1985). This was based on what was then considered to be a medium-scale development (approximately 8000 lines of CORAL[2] 66), specifically, a naval command and control system. The software was developed by professional programmers to "normal commercial" standards. Of 222 potential failures identified during the evaluation, 165 (approx. 74%) were adequately addressed using the recovery block mechanism. The authors believe that the prototype recovery algorithms (i.e. those for storing and rewinding to the recovery point) were a limiting factor in their results. The use of recovery blocks did, however, lead to larger code size, extra memory demands and longer run time.

3.6 Design Diversity — N-Version Programming

In this approach, several alternate versions are run in parallel and their output is passed to an adjudicator. The adjudicator provides a consolidated output for use by the rest of the system. For example, this consolidation could check that outputs are within pre-defined bounds and then return an average; another alternative would be the use of a simple majority voter.

One of the earliest evaluations of n-version programming was performed in 1996 by Knight and Leveson. In this study (Knight and Leveson 1996), 27 different students each wrote a program to satisfy the same set of requirements. Each program had to pass a series of acceptance tests. Once each program was "acceptable", the collection of programs was subjected to a large barrage of tests. Several common faults were detected between the programs and, more significantly, the number of common faults was higher than would be predicted if the programs exhibited independent faults.

A more recent study (from 2006) was conducted by van der Meulen and Revilla. This considered over 36,000 programs that were submitted as potential solutions to problems posed on a programming website. They concluded (van der Meulen and Revilla 2006) that

[2] CORAL is the Computer On-line Real-time Applications Language

using different languages led to more diverse programs (e.g. authors were more likely to code a for loop incorrectly when using C than when using Pascal) and, furthermore, that the benefit of multi-version programming reduces as the individual program versions themselves get more reliable.

One problem that n-version programming cannot solve is that of an incorrect requirement. There are also other aspects of programs where apparent diversity could be undermined. For example, Voas et al. (1997) cite a case where common mode failures were injected in program initialisation, despite other aspects of the programs being quite diverse.

A weakness of the three studies referenced above is that they are based on programs written either by students or by "hobby programmers". None of them is based on the type of software that would emerge from a rigorous development process like that associated with aircraft systems. It is not clear, for example, how limits on allowable cyclomatic complexity may constrain implementation options. Likewise, the extent to which software education implicitly reduces diversity is not clear, nor is it clear whether international development programmes offer an opportunity to increase this type of design diversity, for example, due to different educational approaches and cultural norms (Olson and Olson 2003).

The main conclusion to draw from the above examples is that whilst n-version programming can offer some benefits it does not protect against all software common mode faults. Furthermore, the level of benefit that it provides is less than would follow from a naive assumption of complete independence between the versions.

3.7 Design Diversity — N-Self Checking

The n-self checking approach can be considered as a hybrid of the other two approaches. It involves the use of at least two self-checking software components. A component could be made by combining a version of the software and an acceptance test. Alternatively, it could be made by combining several versions of the software with an adjudicator.

The approach of n-self checking software has been studied less extensively in academia than either recovery blocks or n-version programming. Despite that, as indicated later (in Section 5), it forms the basis of at least one system in the civil sector.

3.8 Design Diversity — Different Functionality

In some applications it may be possible to achieve design diversity by implementing software that achieves the same system-level outcome but using different functionality. The use of a different control law in an aircraft flight control system is a potential example. Likewise, implementing software deliberately intended to manage degraded modes of operation, is another.

In some cases, an easy way of achieving some design diversity may be to use an older version of software alongside the most recent one. This might be appropriate if the change between the two software versions was solely related to performance issues. It is not appropriate, however, if the new version fixed a safety-related issue.

3.9 Watchdog Timers

The previous discussions have focussed on fault tolerant approaches that are embodied in software. Another approach involves the use of a watchdog timer: see, for example, Namjoo and McCluskey (1995).

In general, the system is designed so that it signals the watchdog once during each pre-scheduled interval. If the timer is not reset (and, in some cases, if it is reset more than once) during an interval then the watchdog raises an error. This error can be trapped and used, for example, to restart a processor.

Hence, watchdog timers, which are often implemented in hardware, can be used to provide protection against particular types of software error, including those that cause applications to crash and those that result in infinite loops.

3.10 Summary

The academic literature includes a variety of mechanisms for protecting against software design faults that would otherwise cause CMF. Avoiding, as far as possible, the introduction of faults by the application of suitably rigorous software development processes, is important but it does not fully resolve the problem. Additional mechanisms for protecting against software CMF have different strengths and weaknesses. In particular, there is no reliable way of entirely removing the possibility of software CMF: when multiple copies of software are used, it will always represent a risk. Likewise, whilst the value of diversity is noted, there is no preferred way of mitigating this risk. Furthermore, there does not appear to be a standard way of assessing the level of protection (or risk mitigation) provided by a particular approach, or combination of approaches; as indeed is highlighted in the introduction to Annex C, Part 3 of IEC 61508:2010:

> *"Given the large number of factors that affect software systematic capability it is not possible to give an algorithm for combining the techniques and measures that will be correct for any given application."*

4 Standards

The following paragraphs summarise how software CMF is covered in a number of standards, including ones specific to the air domain as well as ones applicable to other domains that use safety-related software.

DEF STAN 00-970: This UK Defence Standard, "*Certification Specifications for Airworthiness*" (UKMOD 2020), does not directly highlight CMF as an issue, whether specific to software or in any other context.

However, software CMF is indirectly included via references that describe an Acceptable Means of Compliance (AMC). For example, Requirement UK25.1309e is concerned with safety-related Programmable Elements (PE). The associated AMC references ARP 4761 (SAE 1996) and RTCA/DO-178C (RTCA 2011). As discussed below, both of these include specific comments on software CMF.

MIL-HDBK-516C: This U.S. Department of Defense Military Handbook, "*Airworthiness Certification Criteria*" (USDoD 2014), includes several mentions of common mode failure.

For example, within Section 6, "Flight Technology", to satisfy one of the criteria requires "*the probability of a common mode failure is extremely remote ($1x10^{-9}$ or as specified by the procuring activity) and is verified by fault tree and hazard analysis*". Likewise, in Section 7, "Propulsion and Propulsion Installations", satisfying one of the criteria requires

"*controls and subsystems are systemically and operationally isolated to avoid possible cascading failures due to any single or common cause*".

These extracts emerge from subsystem considerations, specifically vehicle control and propulsion. Within MIL-HDBK-516C, software-specific considerations are in Section 15, "Computer Systems and Safety". That section notes typical certification source data include "*various technical analyses and studies (e.g. common mode analysis, ...)*".

These extracts show that CMF is highlighted in MIL-HDBK-516C, both in general and specifically in relation to software. However, there is no detailed information on how best to protect against this possibility.

ARP 4761: This Aerospace Recommended Practice, "*Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*" (SAE 1996), records the potential for CMF, both in general and from the specific perspective of software. It provides a range of techniques for identifying elements that are potentially susceptible to CMF. However, it does not provide a description of ways of protecting against software CMF.

RTCA/DO-178C: This *de facto* assurance standard, "*Software Considerations in Airborne Systems and Equipment Certification*" (RTCA 2011), discusses architectural considerations, with further subsections considering partitioning, multiple-version dissimilar software and safety monitoring.

It notes that partitioning may be achieved by using different hardware for each software component. It also notes that, alternatively, partitioning provisions may be made to allow multiple software components to run on the same hardware platform. Regardless of the adopted approach, partitioned software components should not affect each other's code, inputs, outputs, or data storage.

With respect to multiple-version dissimilar software, RTCA/DO-178C notes that processes completed before dissimilarity is introduced remain a potential source of common errors (e.g. n-version software being developed from the same set of requirements). It also notes that the degree of dissimilarity, and hence the degree of protection, is not usually measurable. As such, dissimilarity is usually used to offer additional protection, beyond that achieved by other DO-178C activities.

IEC 61508: Typically, aviation-specific standards would be preferred to IEC 61508, which adopts a more generic outlook. However, IEC 61508 can still provide useful guidance. For example, in relation to software requirements, Part 3 thereof (IEC 61508-3:2010) notes that:

> *"In order to address independence, a suitable common cause failure analysis shall be carried out. Where credible failure mechanisms are identified, effective defensive measures shall be taken."*

Thus, the importance of protecting against common mode (or, using the terminology of IEC 61058, common cause) failure is also highlighted in relation to both software architecture and software design.

In addition, Annex C thereto includes guidance on a number of techniques and measures that may offer some defence against common mode failure. Examples include fault detection, diverse monitor techniques, re-try fault recovery mechanisms, and graceful degradation.

Nuclear: Whilst it would not be appropriate to apply nuclear standards to aviation directly, they can nevertheless provide informative guidance. One example of this is the US

Nuclear Regulatory Commission (NRC) Regulatory Guide (RG) 1.152, "*Criteria for Use of Computers in Safety Systems of Nuclear Power Plants*" (USNRC 2011), which states:

> *"With the introduction of digital systems into plant safety system designs, concerns have emerged about the possibility that a design error in the software in redundant safety system channels could lead to a common-cause failure or common-mode failure of the safety system function. Conditions may exist under which some form of diversity may be necessary to provide additional assurance beyond that provided by the design and quality assurance programs that incorporate software quality assurance and verification and validation."*
>
> *"With respect to software diversity, experience indicates that the independence of failure modes may not be achieved in cases in which multiple versions of software are developed from the same software requirements."*

Again, this guidance highlights the possibility of software CMF, whilst also noting that some factors (e.g. an incorrect requirement) can defeat assumed levels of software independence.

More-detailed, software-specific guidance for the nuclear industry is provided by a "common position of international nuclear regulators and authorised technical support organisations" on "licensing of safety critical software for nuclear reactors" (TF SCS 2022). Released in 2021 and revised the following year, this recent document notes, for example:

> *"The potential for common cause software failures across all defence barriers utilising computer based systems should be analysed."*
>
> *"Options must be taken as to what are the most effective methods and techniques to ensure diversity of failure behaviour across diverse software programs. ... Current practice is founded on engineering judgment, which identifies what are considered to be the best means to force diversity with some limited support from research. The possibility of providing further scientific support to these judgments is an open research issue."*

These extracts (and other content) highlight the importance of software CMF, whilst recognising there is no preferred way of introducing diversity to protect against it.

Summary: Software CMF is highlighted in a number of standards and guidance documents. In some cases, possible ways of protecting against this are mentioned (including partitioning and n-version programming). There is, however, no consensus on a preferred approach to protecting against software CMF.

5 Civil Air Sector

5.1 Airbus A320

The following description is based on that provided by Brière and Traverse (1993).

The Airbus A320 was the first civil aircraft equipped with a digital electrical flight control system. This system is designed to tolerate both hardware and software design faults. The aircraft was certified and entered service in the first quarter of 1988.

Two dissimilar types of computer are used in the flight control system:

- Elevator and Aileron Computers (ELACs). These are manufactured by Thomson CSF and are based around the 68010 microprocessor.
- Spoiler and Elevator Computers (SECs). These are based on the 80186 microprocessor and are manufactured in conjunction with SFENA/Aerospatiale.

Each computer features a control channel and a monitoring channel. That is, each computer is self-checking (in the sense of being part of an n-self checking implementation). In total, four different pieces of software exist, covering all combinations of type of computer (i.e. ELAC or SEC) and type of channel (i.e. control or monitor).

The software was developed to RTCA/DO-178A Level 1, which was at that time the most stringent civil aviation software standard. The basic rule adopted by Airbus was that the software is built in the best possible way. Dissimilarity between the four pieces of software is an additional precaution; it is not used to reduce the required software quality effort. (This matches the approach espoused in the more recent RTCA/DO-178C. (RTCA 2011))

For any given function, one computer is designated as being active, with the others being available as "hot spares". Failure detection is achieved by comparing the control output with the monitor. For a failure to be declared this difference needs to be greater than a threshold, and to persist for a given period. If a failure occurs, the computer is isolated from external systems.

Even though the architecture includes four control and monitoring computers, only one of these is required to control the aircraft. Three such computers would be sufficient to meet the safety requirements (derived from system safety assessments); the fourth is included for operational reasons, as it allows a take-off to tolerate a single failed computer.

In addition to the digital electrical flight control system, there is also a mechanical back up. This is connected to the trimmable horizontal stabilizer (giving control of pitch) and the rudder (giving control of yaw). Note, however, that the safety objectives of the fly-by-wire system are defined without taking credit for the mechanical backup.

As of December 1992, the A320 had accrued 1.5 million flight hours. These included one case where, due to the combination of an air conditioning failure and a batch of ELACs being fitted with a component that did not fully meet the required temperature operating range, both ELACs were lost. Failure detection and reconfiguration was successful, with control being handled by the SECs.

5.2 Boeing 777

The following description is based on Yeh (2001).

The Primary Flight Computers (PFCs) in the Boeing 777 form a triple-triple redundant system. Safety requirements apply to PFC failures that could preclude continued safe flight and landing. The associated numerical probabilities are 10^{-10} per flight hour for both functional reliability and functional availability.

The 777 has three channels, each of which has an identical PFC. Within each channel there are three dissimilar computing lanes. This dissimilarity is based on the use of different hardware, specifically: an Intel 80486; a Motorola 68040; and an AMD 29050. This dissimilar hardware leads to a requirement for different Ada compilers.

Although the software in each computing lane was developed by different teams, all of them used the same language, namely Ada. In addition, as noted in a separate article by Yeh, Boeing's experience during the 7J7 development (which led to the 777 flight control system) was that the separate development teams had to ask so many questions to clarify the requirements that their independence was irreparably compromised (Yeh 1988).

A median value select algorithm is used on the output from the PFCs. This is supplemented by cross-lane monitoring within each PFC. There is also cross-channel consolidation between the PFCs (to maintain convergence) and cross-channel equalisation (to keep PFC channel tracking within statistically analysable bounds).

The 777 implementation is best viewed as being "replicated n-version programming". The "replicated" reflects the use of three identical PFCs. The "n-version programming" reflects that three computing lanes are used in each PFC; note, however, that the majority of dissimilarity between these lanes comes from the hardware (and associated compiler) rather than the software's design and implementation.

5.3 Summary

These examples provide valuable experience from real-world attempts to implement software design diversity. Neither of these involved any specific claim, quantified or otherwise, or the efficacy of this approach. In one case, it was simply an additional precaution; in the other only limited design diversity was actually achieved. Overall, this experience suggests design diversity cannot be relied upon as a protection measure against software CMF.

6 Evaluation Criteria

6.1 Structure

The preceding subsections have shown that, whilst it is an important issue, there is no preferred way of protecting against software CMF. Likewise, there is no commonly-agreed way of establishing the level of protection offered by a particular combination of approaches.

To overcome this issue, a set of criteria have been developed that, in combination, allow a qualitative assessment to be conducted. The selected criteria can be organised into three categories: the first addresses fault *prevention*; the second covers fault tolerance approaches against different types of potential failure-causing *inputs*; the third considers fault tolerance approaches based on the nature of the software's *outputs*.

6.2 Prevention

A single criterion relates to fault *prevention*. It considers the *software development processes* that have been used. Ideally, these would follow a widely-recognised industry standard (e.g. RTCA/DO-178C (RTCA 2011)).

6.3 Inputs

The following paragraphs summarise *input*-related criteria.

A failure related to an *unexpected external influence* from another piece of software. Such a situation may arise when one piece of (co-hosted) software runs for longer than expected, thus reducing the amount of processing time available for other software functions. Another way this situation could arise would be one application unintentionally corrupting the memory of another application. The standard way of protecting against such failures is an appropriately robust partitioning scheme.

A failure related to *a specific input bit pattern (or patterns)*. Relevant bit patterns include those corresponding to Not a Number (NaN) or Infinity (INF) (e.g. following division by zero), as well as (for example) patterns that evaluate to very small floating-point numbers. Data diversity offers a degree of protection against this potential failure mechanism. Checking inputs before they are used (i.e. defensive programming) offers a lesser, but still useful, degree of protection.

A failure related to an unlikely, *transitory environment or aircraft attitude*. Examples of such environments include crossing the Equator, flying over the North Pole, crossing the International Date Line and breaking Mach 1.0. Since the output from different (non-failing) sensors would be expected to be correlated, data diversity would not be expected to offer protection against this mechanism. Design diversity may offer some protection and, furthermore, the transitory nature of the inputs means recovery via restart (e.g. enabled via a watchdog) may also provide some protection.

A failure related to an *unlikely, but sustained environment or aircraft attitude*. Potential examples could include the aircraft having a negative forward velocity or the aircraft being inverted. As with the previous criterion, data diversity would not be expected to offer protection. In addition, the sustained nature of this criterion means that recovery via restarts would also offer little protection. Conversely, data diversity and design diversity may offer some protection. A testing regime that investigated a wide range of such situations would also help.

A failure related to *a particular sequence of events* (i.e. a failure that is caused by a specific internal state rather than a particular input). A potential example here could be a sequence of inputs that led to a navigation filter becoming unstable. The importance of the sequence means that both data diversity and design diversity would offer some protection. It also means that such issues are harder to discover via testing.

6.4 Outputs

The following paragraphs summarise *output*-related criteria.

A case where *no output* is provided. This could occur, for example, if a processor gets stuck in an infinite loop, or if the Operating System (OS) crashes. Hardware watch dogs are a standard protection approach for this criterion, because they readily facilitate failure detection and failure recovery.

A case where *a known incorrect output is provided from one computational branch (of several)*. This situation could arise, for example, due to a failing sensor giving an invalid input to a single computational branch, with the invalid nature of the input being recognised and the branch setting an output error flag. Note that the main reason why different computational branches provide different types of output is because of the presence of some diversity (either in data or design): this diversity provides some protection against this criterion. An adjudicator, or voter, that discounts the incorrect output is also required.

A case where *a known incorrect output is provided from all computational branches*. This situation could arise in a situation where a "bad" input is provided and there is no diversity

in the implementation. Being able to provide pre-defined "safe" outputs (as part of input checking) offers some protection against this situation. Restarting all the computational branches is another viable protection method (provided the aircraft can sustain controlled flight whilst the restart occurs).

A case where *an unknown incorrect output is provided from one computational branch (of several)*. Note that, as indicated above, this situation can only occur if there is some diversity in the implementation. An undetected failed sensor, which only feeds a single computational branch, could cause this situation. An adjudicator, or voter, that compares the outputs from the different branches offers some protection in this case.

A case where *an unknown incorrect output is provided from all computational branches*. This situation could arise if the platform (and hence, by extension, the embedded software) is used outside its design envelope. Diversity, and particularly design diversity, may offer some limited protection. A suitably wide, and imaginative, testing regime could also help: this is likely to benefit from a combination of software engineers and domain experts.

7 Suggested Protection Approaches

The preceding subsections have established that software CMF is an important issue, having been the cause of previous incidents and accidents, the subject of academic research, a topic in relevant standards documents and a consideration in civil aviation platforms. Nevertheless, there is no preferred way of protecting against software CMF.

It is suggested that subsystems where software CMF may pose a hazard adopt several protective approaches. These are likely to include multiple examples drawn from the following list; where appropriate additional approaches (not listed here) may also be used.

Fault Prevention, which aims to minimise the number of faults present in the software. This is generally achieved through the use of processes that apply an appropriate level of control to software development activities. RTCA/DO-178C (RTCA 2011) is an example of such processes.

Partitioning, which isolates failures in the region where they occur, rather than allowing them to cascade across the entire system. This is related to, but also supplemental to, partitioning that protects one application's memory area from the actions of another application.

Data Diversity, which involves providing different copies of software (which may or may not be identical) with different inputs. This is easiest to adopt when data diversity is naturally generated, e.g. from different sensors. Artificial data diversity can be introduced, but it can be difficult to determine the appropriate level of diversity to be used.

Input Checking, which (not surprisingly) involves checking inputs before they are used. This is a natural part of defensive programming and supports partitioning, e.g. by preventing bad data polluting subsequent computations.

Design Diversity, which typically involves using software developed by separate programming teams to perform the same function.

Failure Detection, which is concerned with identifying software failures so that they can be adequately managed and, ultimately, recovered from.

Failure Recovery, which typically involves returning the software to a "known safe" state and resuming processing.

Extensive Testing, which might involve imaginative test construction on the part of test engineers and domain experts, combined with automated test approaches, e.g. fuzzing.

For ease of reference, Table 1 indicates the relationship between evaluation criteria and the approaches suggested above. Within this table, a "++" symbol indicates a case where the approach may be expected to provide significant benefit; a "+" symbol indicates a case where moderate benefit may be expected.

Table 1 ~ Indicative Relationship between Evaluation Criteria and Suggested Approaches for Providing Protection Against Software CMF

	Fault Prevention	Partitioning	Data Diversity	Input Checking	Design Diversity	Failure Detection	Failure Recovery	Extensive Testing
Prevention: Software development processes	++							
Inputs: Unexpected external influences		++						
Inputs: Specific input bit pattern (or patterns)			++	+				+
Inputs: Unlikely, transitory environment or aircraft attitude					+	++	++	
Inputs: Unlikely, but sustained environment or aircraft attitude			+		+			++
Inputs: Particular sequence of events			++		++			
Outputs: No output						++	++	
Outputs: Known incorrect output from one computational branch			++		++	+	+	
Outputs: Known incorrect output from all computational branches				+		++	++	
Outputs: Unknown incorrect output from one computational branch			++		++	+	+	+
Outputs: Unknown incorrect output from all computational branches			+		+			++

It should be noted that Table 1 is intended to illustrate one way to substantiate an argument that protection against software CMF had been appropriately considered. In particular, the table's contents are subjective in nature and not intended to provide strict direction, nor form the basis of any algorithm to support the combination of identified techniques.

Consequently, it is not appropriate to dissect Table 1 in forensic detail. Nevertheless, there is merit in highlighting some key points.

When considering the criteria (i.e. looking across each of the rows), it is apparent that each criterion has at least one "++", with most criteria having at least one additional "+". There are two exceptions, which are criteria that only have a single "++", specifically, "Prevention: Software development processes" and "Inputs: Unexpected external influences". This is a consequence of the focus adopted for this paper. It would be possible to add significantly more detail on both of these aspects (e.g. use of formal static analysis tools, analysis of potential interference paths in multi-core processors) with a corresponding increase in symbols in the table. In addition, it is expected that these activities would be undertaken as part of any safety-related software development. Consequently, the two single "++" criteria are not judged to represent significant limitations in the approaches that are available to protect against software CMF.

When considering the approaches (i.e. looking down the columns), it is apparent that most approaches have multiple entries, both "++" and "+". There are three exceptions to this. Two of these exceptions are "Fault Prevention" and "Partitioning", which directly relate to the two criteria discussed above. The other exception is "Input Checking". As with the previous items, the apparently limited value of input checking is a direct consequence of this paper's intentionally narrow focus on software CMF. In particular, input checking is of significant wider value in safety-related software implementations.

Overall, constructions like Table 1 should help systems engineers understand, in a qualitative sense, levels of protection against software CMF. It also highlights that protection will likely have to be built from a number of approaches, and that common safety-related software approaches, e.g. fault prevention, partitioning, and input checking, are by themselves unlikely to be sufficient.

8 Closing Remarks

Software CMF is a significant issue, having been the cause of a number of accidents and incidents. It (and protection against it) has been the subject of significant academic research. Its importance is highlighted in standards and it has influenced development approaches on key civil air programmes. However, there is currently no preferred way of protecting against software CMF, nor is there an established mechanism for assessing any protection that has been implemented. To address this limitation, a set of evaluation criteria have been developed (and used on a previous platform-specific piece of work). These allow suggested protection approaches to be advanced for future air domain programmes (and other safety-related systems).

Disclaimers

This article is an overview of UK MOD sponsored research and is released for informational purposes only. The contents of this document should not be interpreted as representing the views of the UK MOD, nor should it be assumed that they reflect any current or future UK MOD policy. The information contained in this document cannot supersede any statutory or contractual requirements or liabilities and is offered without prejudice or commitment.

Acknowledgments

The authors gratefully acknowledge the comments made by the anonymous reviewers. These significantly improved the article.

References

Ammann P. E. and Knight J. C. (1988). *Data Diversity: An Approach to Software Fault Tolerance*. IEEE Trans. Computers, 37 4 (1988) 418-425.

Anderson T., Barrett P. A., Halliwell D. N. and Moulding M. R. (1985). *Software Fault Tolerance: An Evaluation*. IEEE Trans. Soft. Eng., SE-11 12 (1985) 1502-1510.

Brière D. and Traverse P. (1993). *AIRBUS A320/A330/A340 Electrical Flight Controls - A Family of Fault-Tolerant Systems*. In 23rd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-23), pages 616–623, Toulouse, France, 1993.

Hecht H. (1993). *Rare Conditions — An Important Cause of Failures*. In: Practical Paths to Assurance, Proceedings of the Eighth Annual Conference on Computer Assurance. (COMPASS '93) pp.81-85, 1993. IEEE.

IEC 61508-3:2010. *Functional Safety of Electrical/electronic/programmable electronic Safety-related Systems — Part 3: Software requirements*. IEC 61508-3, Edition 2. International Electrotechnical Commission. Geneva. 2010.

JSSSC. (2010). *Software System Safety Handbook: A Technical and Managerial Team Approach, Sub-section F.4, "Flight Controls Fail at Supersonic Transition"*. Joint Software System Safety Committee, US Department of Defense. Available from: https://ac.cto.mil/wp-content/uploads/2019/06/Joint-SW-Systems-Safety-Engineering-Handbook.pdf. Accessed 19th July 2023.

Knight J. C. and Leveson N. G. (1996). *An Experimental Evaluation of the Assumption of Independence in Multi-version Programming*. IEEE Trans. Soft. Eng., SE-12, 1 (1996) 96-109.

Laprie J-C., Arlat J., Béounes C. and Kanoun K. (1990). *Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures*. Computer, 23 7 (1990) 39-51. IEEE.

Lions J-L. (1996). *ARIANE 5: Flight 501 Failure*. Report by the Inquiry Board chaired by Prof. J-L. Lions. Available from: http://www.di.unito.it/~damiani/ariane5rep.html. Accessed 19th July 2023.

Namjoo M. and McCluskey E. J. (1995). *Watchdog Processors and Capability Checking*. Twenty-Fifth Int. Symp. on Fault-Tolerant Computing — Highlights from Twenty-Five Years, June 1995, 94-97.

Olson J. S. and Olson G. M. (2003). *Culture Surprises in Remote Software Development Teams: When in Rome doesn't help when your team crosses time zones, and your deadline doesn't*. Queue, 1(9), pp.52-59.

Powell D., Arlat J., Deswarte Y. and Kanoun K. (2011). *Tolerance of Design Faults*. In: Jones C. B. and Lloyd J. L. (editors). *Dependable and Historic Computing*. Lecture Notes in Computer Science, Volume 6875. Springer Nature, London.

Randell B. (1975). *System Structure for Software Fault Tolerance*. IEEE Trans. Soft. Eng., SE-1, 2 (1975) 220-232.

RTCA. (2011). *Software Considerations in Airborne Systems and Equipment Certification*. RTCA/DO-178C. RTCA Inc. Also available as EUROCAE Document ED-12C.

SAE. (1996). *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*. ARP4761, December 1996. SAE International.

Siewiorek D. P., Chillarege R. and Kalbarczyk Z. T. (2004). *Reflections on Industry Trends and Experimental Research in Dependability*. IEEE Trans. Secure and Dependable Computing, 1 2 (2004) 109-127.

TF SCS. (2022). *Licensing of Safety Critical Software for Nuclear Reactors: Common position of International Nuclear Regulators and Authorised Technical Support Organisations*. Regulator Task Force on Safety Critical Software (TF SCS). Available from: https://www.onr.org.uk/software.pdf. Accessed 20th July 2023.

Tomek L. A., Muppala J. K. and Trivedi K. S. (1993). *Modeling Correlation in Software Recovery Blocks*. IEEE Trans. Soft. Eng, 19 11 (1993) 1071-1086.

TTSB. (2021). *CI202 Occurrence Investigation*. Executive Summary of the Final Report translated into English, September 2021. Taiwan Transport Safety Board. Available from: https://www.ttsb.gov.tw/media/4913/ci202_executive-summary_release.pdf. Accessed 19th July 2023.

UKMOD. (2020). *Certification Specifications for Airworthiness: Part 1 — Fixed Wing Combat Air Systems*. DEF.STAN.00-970, Issue 17, August 2020.

USDoD. (2014). *Airworthiness Certification Criteria.* MIL-HDBK-516C, December 2014.

USNRC. (2011). *Criteria for Use of Computers in Safety Systems of Nuclear Power Plants*. Regulatory Guide 1.152, Revision 3, July 2011. U.S. Nuclear Regulatory Commission.

van der Meulen M. J. P. and Revilla M. A. (2006). *The Effectiveness of Software Diversity in a Large Population of Programs*. IEEE Trans. Soft. Eng., 34, 6 (2006) 753-764.

Voas J., Ghosh A., Charron F. and Kassab L. (1997). *Reducing Uncertainty About Common-Mode Failures*. In: Proceedings of the Eighth International Symposium on Software Reliability Engineering, 1997, pp. 308–319.

Walmsley R., Anderson T., Brendish C., McDermid J. A., Rolfe M., Sultana J., Swan M. and Toms M. (2015). *NATS System Failure 12 December 2014 — Final Report*. Independent Enquiry Panel chaired by Sir Robert Walmsley KCB. Available from: https://www.nats.aero/wp-content/uploads/2015/05/Independent-Enquiry-Final-Report-2.0.pdf. Accessed 19th July 2023.

Yeh Y. C. (1988). *Design Considerations in Boeing 777 Fly-By-Wire Computers*. In 3rd IEEE Int. Symp. on High-Assurance Systems Engineering, pages 64-72, 1988.

Yeh Y.C. (2001). *Safety Critical Avionics for the 777 Primary Flight Controls System*. Digital Avionics Systems Conference, 2001 (20th DASC).