NASA/CR–2011-217150

Concept Development for Software Health Management

Jung Riecks, Walter Storm, and Mark Hollingsworth
Lockheed Martin Aeronautics Company, Fort Worth, Texas

May 2011

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include creating custom thesauri, building customized databases, and organizing and publishing research results.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at 443-757-5803

- Phone the NASA STI Help Desk at 443-757-5802

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CR–2011-217150

Concept Development for Software Health Management

Jung Riecks, Walter Storm, and Mark Hollingsworth
Lockheed Martin Aeronautics Company, Forth Worth, Texas

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL06AA08B

May 2011

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

CONTENTS

FOREWORD

Lockheed Martin Corporation, acting through its Lockheed Martin Aeronautics Company (LM Aero) operating unit, has prepared this document for the National Aeronautics and Space Administration's (NASA) Langley Research Center under contract NNL06AA08B, delivery order number: NNL07AB06T. The work documented herein was performed from October, 2008 through July, 2009.

Contributors included Jung Riecks, Walter Storm, and Mark Hollingsworth. Additional support was provided by: Claudia Marshall, Dan Harbour, Diane Nixon, and Tom Schech.

INTRODUCTION

This report documents the work performed by Lockheed Martin Aeronautics (LM Aero) under NASA contract NNL06AA08B, delivery order NNL07AB06T. The Concept Development for Software Health Management (CD-SHM) program was a NASA-funded effort sponsored by the Integrated Vehicle Health Management Project, one of the four pillars of the NASA Aviation Safety Program. The CD-SHM program focused on defining a structured approach to software health management (SHM) through the development of a comprehensive failure taxonomy that is used to characterize the fundamental failure modes of safety-critical software.

To enable the detection and mitigation of software errors through SHM, our approach is to treat software as another system device that exhibits failure modes according to a canonical failure reference of legacy and emerging safety-critical software. Many SHM concepts stem from failure modes and effects analysis (FMEA) of software in a manner similar to that used for hardware, however the failure modes for software are not well known, and the techniques for applying a software FMEA during system design are not widely published [1], [2]. Our goal was to address these shortcomings by quantifying the scope, magnitude and types of fundamental software errors that manifest themselves throughout the development of advanced flight-critical software. We developed our approach in two phases: 1) the creation of a taxonomy for fundamental software anomalies based on data from various advanced, flight-critical software development programs; and 2) the development of integrated risk models, mitigation schemes, design considerations and patterns based on fundamental failure data.

The following sections document the process and results of the study.

APPROACH

PREPARING THE DATA

The source of our study was the development of flight-critical software systems from a combination of several recent, advanced development and production programs. The background information required for the investigation and analysis was gathered from across various database systems and normalized to a common database. We used the resulting database as the source for our error classification and taxonomy development.

The analysis of the database was performed manually, as several subject matter experts read through and classified each anomaly report as a type of fundamental failure. The failure types were developed after several passes through the data, where the root causes were distilled to basic phrases or terms that adequately describe and classify their nature. Only those terms which adequately described at least 0.1% of all the cases studied were considered an eligible term for the fundamental failure type.

CLASSIFICATION DETAILS

As it turns out, all of the raw data sources for this analysis are (more or less) freeform text. From this, it was quickly evident that the only way to produce a comprehensive taxonomy was to read each account individually. We held many meetings with our program contacts to study the current anomaly report structures. In the current anomaly report structure, there is a multitude of information; however there is no easy way to outline the cause classification or root cause in detail. Nonetheless, we identified areas that still gave us some advantages. Using the current reporting system, we were able to identify the anomaly found, the phase in which it was introduced and its severity. This information is the foundation of our study and the basis for our recommendations.

CREATING THE BASELINE

The first step in creating the baseline data set involved eliminating all of the unnecessary information from the raw reports, and boiling them down to the fundamental symptoms, phases, severities, and root causes. The steps involved in the data elimination process were:

1. Delete all the blank sections
2. Delete unimportant sections for this project. (i.e. User ID, date,…etc)
3. Delete 'cancelled' or 'analysis' in status
4. Delete 'external', 'duplicate', 'not a problem', 'suspended' in final resolution
5. Delete 'No' in confirmed problem
6. Delete all the data which is not a software related problem in problem product

After this purging, the resultant database was the baseline for the project.

CREATING THE FAILURE TAXONOMY

There are four different sections from the anomaly reports that we receive from any given program. These sections are the: *Anomaly Behavior*; *Expected Behavior*; *Root Cause* and *Corrective Action Task*. All of these sections have a description field that is free format text which contains a limit of 2,000 characters. From the four sections above, we create sections that are named: *Anomaly*; *Cause Classification* and *Root Failure*.

1. The "Anomaly" contains a very short description of the problem behavior. The "anomaly" comes from the "Anomaly Behavior" and "Expected Behavior" sections from the original report.
2. The "Cause Classification" is the classification and abstraction of the failure. The "Cause Classification" information comes from the "root cause" and "corrective action task" section of the anomaly reports.
3. The "Root Failure" is the taxonomy of failures. The "Root Failure" information also comes from the "Root Cause" and "Corrective Action Task" section of the anomaly reports.

Since we do not have an outline of the Cause Classification and Root Failure, we first started with a sample group of anomaly reports to attempt to identify a pattern of Cause Classification and Root Failure. While we were working on this sample group, we realized that the anomaly reports are not a large enough sample group to discern a pattern of cause classification and root failure. We decided that we needed to review all of the anomaly reports to create the initial outline of Cause classification and Root Failure. The anomaly report data contains all the life cycle of the program. After examining several hundred anomaly reports, we started to see some patterns. The patterns enabled us to keep as much detail as possible with respect to the Cause Classification and Root Failure while still allowing enough entries to be statistically significant. This analysis was then refined into the final taxonomy described in the following section.

ANALYSIS RESULTS

Our taxonomy consists of 16 failure classes and 114 fundamental failure types. In order to define a specific failure type, the type must provide statistical significance for the term by adequately defining at least 0.1% of all anomaly reports studied. Each class and the fundamental types derived from them are described in the following sections.

FAILURE CLASSES

ALGORITHM

The *Algorithm* failure class defines a family of 31 software errors that represent, in general terms, fundamental errors in the software design. For example, errors such as invalid assumptions about the environment in which the system operates may be considered *Algorithm* errors.

Algorithm Failure Class

Failure Type	Definition
compound logic	incorrect compound logic (i.e. and, or, nand, nor…)
data transfer/message	incorrect algorithm of data transferring (refresh)
dead code	leftover code form past causes a problem
decision logic	incorrect decision logic (i.e. if-then-else, case statements, begin-end, mode transition, wrong execution sequence….)
design	logic of algorithm is incorrect
engineering unit	incorrect engineering unit is used in calculation
equation/calculation	incorrect equation or calculation
failure detection	incorrect failure detection algorithm
failure isolation	incorrect failure isolation algorithm
failure management	incorrect failure management logic (failure reporting)
failure reporting	incorrect failure reporting or trigger logic to generate failure report
incorrect signal	incorrect signal is used in calculation
initialization logic	incorrect initialization algorithm
initialization of values	incorrect initialization values
inverted logic	inverted true or false logic

Algorithm Failure Class (Cont'd)	
Failure Type	**Definition**
missing initialization	missing initialization function
missing limiter	missing limiter in the calculation
prototype	missing prototype
range	incorrect or unnecessary range in calculation or condition
relational operator	incorrect relational operator (i.e. >, <, >=, <= ...)
reset logic	incorrect reset algorithm
reset timing	incorrect reset timing
response to detected failure condition	incorrect repose to detected failure condition
sampling time	incorrect sampling time
setting value/variable	incorrect algorithm to setting values or variables
syntax	syntax error
test modeling	incorrect test modeling produce incorrect values for the test
threshold	incorrect threshold
timing	incorrect delay
typo	typo in algorithm causes disconnect between signals
validity check timing	missing or incorrect or inappropriate timing of validity check

BUS INTERFACE

The *Bus Interface* class defines a collection of error types that represent data source and bus translation errors. This is a relatively focused class with the following 4 error types.

Bus Interface Failure Class	
Failure Type	**Definition**
bit position	incorrect bit position
bus initialization failure	bus initialization failure
data source	incorrect data source is connected to bus interface
missing signal	missing a signal in bus interface

CONFIGURATION MANAGEMENT (CM)

Although often referred to in the context of process and tools, problems within CM manifest themselves as real problems in flight-critical software systems. Through this study, we identified the following 6 CM failure types.

Configuration Management Failure Class	
Failure Type	**Definition**
approval delay	correct version of SW was not approved.
implementation delay	
incorrect version of software	using incorrect version of SW
missing CR implementation	missing CR implementation
outdated requirement	did not update requirement to match a SW change
requirement incorporation delay	did not update SW to match a requirement change

COMPILER ERROR

The *Compiler Error* is a general class of error that is created by the tools in the software build chain. That is, an error in any specific tool used in the process of translating source code into executable code is considered a *Compiler Error*. In this study, the only type of compiler error identified was the generation of incorrect assembly code—most likely because the tools used to build the flight-critical systems in the study are mature and have been pre-qualified. In fact, when developing flight-critical systems using mature software development environments, compiler errors account for less than 0.5% of all software errors.

Compiler Error Failure Class

Failure Type	Definition
Incorrect Assembly Code	Incorrect Assembly Code

DATA DEFINITION

Incorrect representation of data structures in memory, data offsets and row ordering are all examples of *Data Definition* errors. During this study, we identified the following 6 distinct data definition error types:

Data Definition Failure Class

Failure Type	Definition
data structure	incorrect data structure
data type	incorrect definition of data type
enumeration	incorrect enumeration
lookup table data	incorrect lookup table data
offset	incorrect data offset for I/O or bus list or memory-mapped message
size	incorrect bit or byte size

DATA HANDLING

A *Data Handling* error is a class of software error that involves illegal, undefined or incorrect use of a data element or variable. *Data Handling* errors differ from *Data Definition* errors in that they do not manifest themselves at the module interface, and do not necessarily involve incorrect structure definitions. We have identified the following 14 types of Data Handling errors:

Data Handling Failure Class

Failure Type	Definition
bias	missing or incorrect bias
bit conversion	incorrect handling of 16bit and 32 bit conversions
breakpoint	incorrect breakpoint
byte/bit order	incorrect byte or bit order(i.e. endianness, byte swap, LSB and MSB reversed)
indexing	improper indexing into arrays or table

Data Handling Failure Class (Cont'd)	
Failure Type	**Definition**
input fault tolerance	incorrect tolerance to detect input fault
logic	incorrect data handling logic
masking data	masking data with incorrect values or not masking data which we are expecting to be masked
memory address	using incorrect memory address
mnemonics	incorrect mnemonics in hash table
scaling factor	using incorrect scaling factor
transition logic	incorrect transition logic
variable	incorrect variables or variable type to access data
variable scope	incorrect variable type (global, local)

DOCUMENTATION

The *Documentation Error* is a general class that defines errors in the documentation (requirements, design documents, flowcharts, state-charts, architecture diagrams, etc.) that lead to software anomalies downstream in the process. There were no emergent patterns from this study to define specific documentation error types with any statistically significant basis, even though 11% of all errors were of this type. Fortunately, *Documentation* errors—having a high phase-containment ratio—are often detected during the development phase in which they are created, or the very next phase in the process. We discuss the significance of this in more detail later[1].

HARDWARE

Hardware Errors are defined as a class of error that elucidate deficiencies or flaws in the physical systems upon which the software has direct or indirect influence. This study defines 1 type of hardware error:

Hardware Failure Class	
Failure Type	**Definition**
unexpected behavior	Hardware deficiency mitigated by Software

INPUT-OUTPUT (I/O) SYSTEM

I/O System Errors represent a class of errors that are resident in modules or subsystems which are responsible for providing data to (and getting data from) other modules or subsystems within the architecture. Although this class of error is not the most prevalent, I/O System errors have the highest average severity of all the error classes. Again, the significance of this will be discussed later in the report[2]. We recognize 4 distinct I/O System error types.

[1] See Error Analysis – Rankings by Occurrence.
[2] See Error Analysis – Rankings by Severity.

I/O System Failure Class	
Failure Type	**Definition**
data list	incorrect data list
I/O synchronization	Coordination of I/O timing, lists, etc.
order of data structure	incorrect order of data structure
signal assignment	missing or incorrect signal assignment

IMPLEMENTATION

An *Implementation Error* is defined as a general class of error through which a requirement or software change request was implemented incorrectly in the source code. This study did not reveal any significant or distinct implementation error types, and all implementation errors account for less than 1% of all anomaly reports studied.

INTER-PROCESS COMMUNICATION

We define, in general, *Inter-process Communication Errors* as incorrect hand-shaking between processes or parallel modules. This includes coordination of resources, failure management and overall timing issues. This study revealed 9 distinct inter-process communication error types.

Inter-process Communication Failure Class	
Failure Type	**Definition**
decision logic	incorrect decision logic (i.e. if-then-else, case statements, begin-end, mode transition, wrong execution sequence….)
engineering unit mismatch	engineering unit mismatch
failure management	incorrect failure management logic
I/O synchronization	I/O is not synchronized in inter-channel data box
initialization logic	incorrect initialization logic
logic	incorrect logic of inter-process communication
reset timing	incorrect reset timing
sampling time	incorrect sampling time
timing	incorrect delay

PERFORMANCE

The class of errors considered under the term *Performance* defines those errors which violate either real-time requirements or processor utilization thresholds. During our study, we were able to statistically substantiate the following performance error type:

Performance Failure Class	
Failure Type	**Definition**
Exceed Processor Utilization Target	Exceed Processor Utilization Target

SELF-TEST

As part of the development process for flight-critical systems, it is necessary to incorporate into the system a sufficient suite of pre-flight tests that verify the suitability of the system relative to the mission it is about to perform. This test sequence; often referred to as *Self Test* or built-in test, is designed to provide a *go/no-go* decision relative to predetermined fitness conditions. However, errors in the *Self Test* itself may yield erroneous results. Such is the class of error defined by this category, from which we identify the following 8 distinct types:

Self-Test Failure Class

Failure Type	Definition
improper test condition	running test with improper condition
design	incorrect test design
inadequate requirement	requirement is not specific enough to test
test timing	incorrect test timing
time management	inefficient use of time
value of location	location contains incorrect values in test pattern
values for test	incorrect values or reference for test
missing reset function	missing reset function in test procedure (for either necessary or work around)

SYSTEM INTEGRATION

System Integration defines a class of errors that arise when major system components come together or interact with moderate dependency. Such errors may be obvious right at system power-up, while others may not be identified until the system is subject to unique or unforeseen circumstances. Based on this study, *System Integration* errors have the most derived types of all the error classes. We identified 24 of them.

System Integration Failure Class

Failure Type	Definition
channel synchronization	channels are not synchronized
conflicting requirement	conflicting requirement
change request (CR)	incorrect CR was written, approved and incorporated.
data source	incorrect data source is connected to bus interface
engineering unit mismatch	signals from two different systems did not agree on units (i.e. radian, degree)
ICD and SW mismatch	ICD and SW are not matching
inconsistent interface order	inconsistent index(order) of I/O between systems
incorrect requirement	incorrect requirement
interface	incorrect interface
manual	incorrect manual (flight manual)

System Integration Failure Class (Cont'd)	
Failure Type	**Definition**
memory use	using incorrect kind of memory (i.e. use CPU check RAM instead of internal RAM)
missing data	missing data in a table of design document
missing datapump	missing data in data pump list
missing header file	missed include header file in the main code
missing signals in ICD	missing signals in ICD
missing SW update	hardware changed but SW did not change
missing testpoint	symbol is missing for test symbol table
no requirement	there is no requirement for an issues so it needed to be created
parameter	incorrect parameter
parameter order	parameter order
rate synchronization	rate synchronization
requirement not clear	not enough guide lines to understand requirement
testpoint name	symbol name of signal and signal in code are not the same
unnecessary requirement	unnecessary requirement needed to be deleted

TOOLS

Unfortunately, tools also introduce errors into software systems. Through our study, we identified the following 2 *Tool Error* types:

Tool Failure Class	
Failure Type	**Definition**
Algorithm	tools generates incorrect signal or values
input data	missing or incorrect input data so tool generate junk code

USER/PILOT

Any errors associated with the operation of the system purely from the perspective of the user or pilot, under normal operating conditions, fall under the *User/Pilot* class. That is, errors identified through specific flight tests or failure conditions—perhaps employing a pilot or user—are not considered *User/Pilot* errors. Through this study, there were no instances where any action on behalf of the user or pilot caused a software failure that was not properly matched to another error class. All qualifications considered; we identified the following type of User/Pilot error type:

User/Pilot Failure Class	
Failure Type	**Definition**
preference	results that are not necessarily incorrect or unsafe but pilots want to change so they feel more comfortable or low Cooper-Harper ratings

ERROR ANALYSIS

Once we identified the proper taxonomy, we were able to perform some useful analysis on the resultant data. This section describes our analysis and the corresponding results.

BACKGROUND

Similar to many risk management approaches[3], our approach considers the primary drivers of **probability** and **severity**. We also add a third dimension—the **likelihood of detection**. Although similar in name to what one may encounter in a failure mode and effects analysis worksheet[4], this parameter measures how long a given type of software error is likely to remain present in the system before it is found. That is, it is a measure of the delta between the phase in which an error is detected and the phase in which the root cause analysis determined it was likely injected.

The primary difference between our analysis and other risk assessments is that our results are based on data and events that already exist and have transpired rather than estimating a probability of occurrence and a severity. We then use the entire collection of data to make predictive inferences and suggestions for solutions that can mitigate high-risk areas through software health management.

THE RISK PRIORITY NUMBER

The Risk Priority Number (RPN) is a fundamental measure of risk associated with each failure type. It is a parameter, normalized to a value between 0 and 1000, which clearly indicates the relative risk priority of elements within the taxonomy. It is calculated as:

$$RPN = O \times S \times D$$

Where:

$$\begin{aligned}
&O := Relative\ Frequency\ of\ Occurance \\
&S := Severity\ of\ Error \\
&D := Phase_{Detected} - Phase_{Injected}
\end{aligned}$$

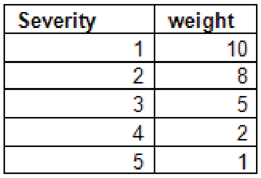
CALCULATING RELATIVE FREQUENCY

The relative frequency of a class is calculated by the sum of all anomalies under that class divided by the number of anomaly reports in the most frequent class. It is represented as a normalized number between 0-10.

[3] i.e. quantitative or probabilistic risk assessment
[4] See http://en.wikipedia.org/wiki/Failure_mode_and_effects_analysis for an example.

CALCULATING RELATIVE SEVERITY

The severity term is calculated by normalizing the anomaly severity codes against a weighted scale. Each anomaly report we analyzed had an associated severity code ranging from 1-5, where severities 1&2 directly affect safety of flight. To accurately represent this separation, we normalized the severity code as a number between 1 and 10 according to the following table:

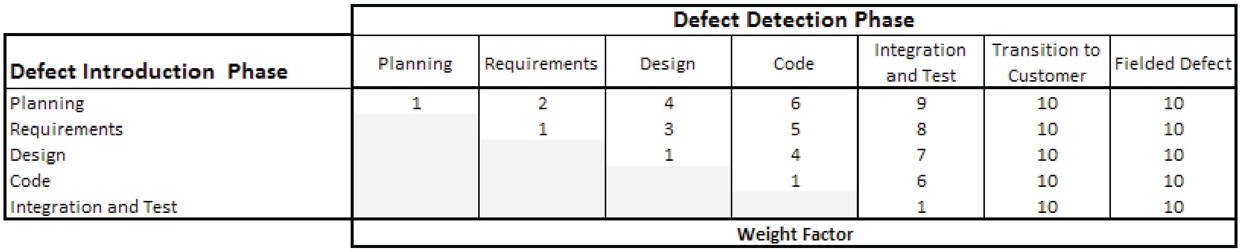


Severity	weight
1	10
2	8
3	5
4	2
5	1

CALCULATING THE DETECTION PARAMETER

The final parameter of the RPN represents how long a software error remained within the system since the error was first introduced. That is, it is an indicator of how likely a certain class of error will go undetected by the established verification and validation (V&V) process.

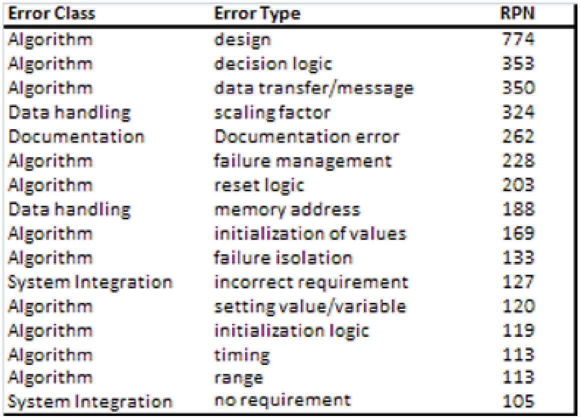
To create the parameter, we analyzed each anomaly report and calculated the weighted delta-phase factor directly from the table below. For example, if an anomaly was detected during Integration and Test, and the root cause of the error was found to be an error in the Requirements of that module, then the delta-phase value is 8.



	Defect Detection Phase						
Defect Introduction Phase	Planning	Requirements	Design	Code	Integration and Test	Transition to Customer	Fielded Defect
Planning	1	2	4	6	9	10	10
Requirements		1	3	5	8	10	10
Design			1	4	7	10	10
Code				1	6	10	10
Integration and Test					1	10	10
	Weight Factor						

PRESCRIPTIONS OF THE RPN MODEL

In general, any element with an RPN greater than 100 can be considered *high-risk*. Although this cutoff is open to conjecture, the upper end of the RPN spectrum surely deserves attention. For instance, the top-most element—algorithm design—can emerge as an entire field of study in its own right. The table to the right shows elements from the entire taxonomy whose RPN is greater than 100.



Error Class	Error Type	RPN
Algorithm	design	774
Algorithm	decision logic	353
Algorithm	data transfer/message	350
Data handling	scaling factor	324
Documentation	Documentation error	262
Algorithm	failure management	228
Algorithm	reset logic	203
Data handling	memory address	188
Algorithm	initialization of values	169
Algorithm	failure isolation	133
System Integration	incorrect requirement	127
Algorithm	setting value/variable	120
Algorithm	initialization logic	119
Algorithm	timing	113
Algorithm	range	113
System Integration	no requirement	105

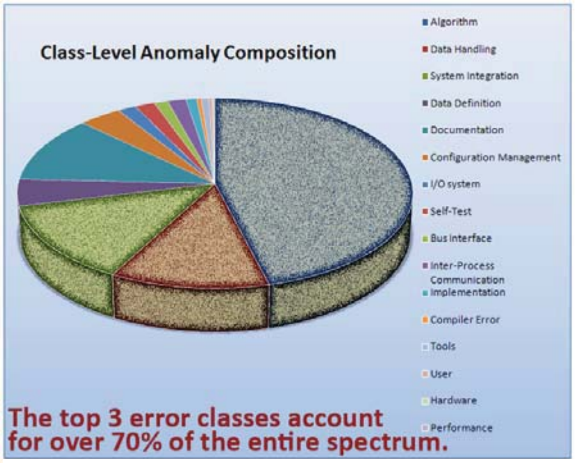
DETAILED CLASS ANALYSIS

The following sections present a detailed analysis of each error class. The analysis shows the RPN for each specific error type of the taxonomy as well as the type's relative distribution profile within the class. The following table is a summary of those error classes which have a limited number of types.

Error Class	Error Type	RPN
Documentation	Documentation error	262
Implementation	requirement implementation error	46
Tools	Algorithm	30
Compiler Error	Incorrect Assembly Code	29
Pilot	Preference	12
Hardware	unexpected behavior	8
Performance	Exceed Processor Utilization Target	7
Tools	input data	1

A roll-up the individual error types reveals some notable observations about the individual error classes themselves. Perhaps the most notable of which is that the top three error classes—*Algorithm*, *Data Handling* and *System Integration*—account for over 70% of all software errors, as illustrated in the graph shown in Figure 10, at right.

Not only are the top three classes the most frequent; with RPN values between 100 and 1000, they are also in the high-risk category, as seen in Figure 11 below.



The top 3 error classes account for over 70% of the entire spectrum.

Figure 1 - Class-Level Analysis

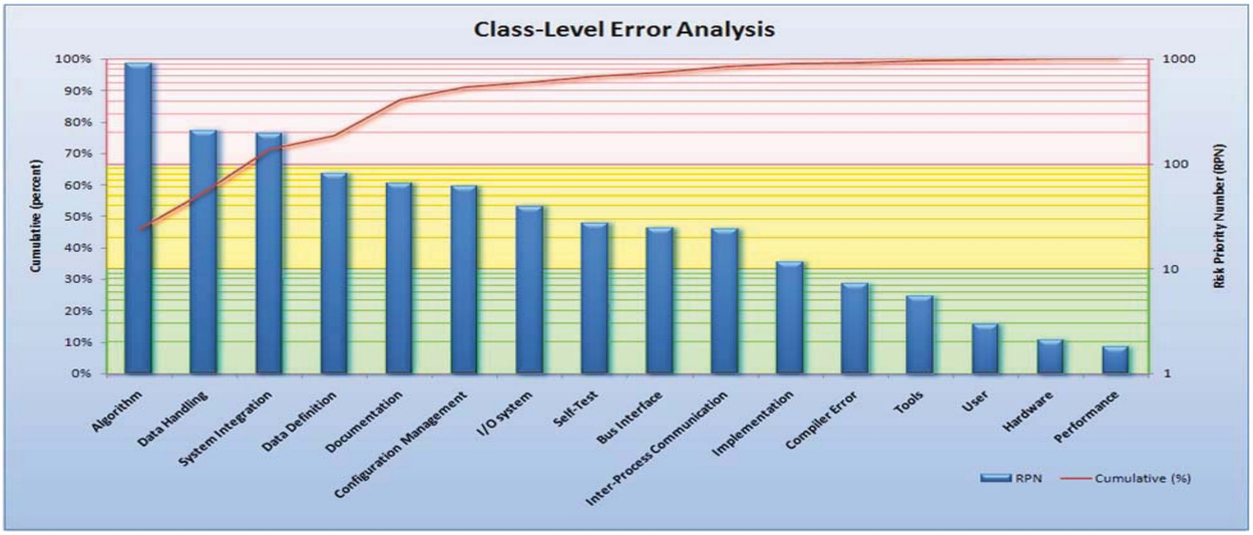


Figure 2 – Class-Level Error Profile

RPN COMPONENT ANALYSIS

At this point, we discuss the individual parameters of RPN for the failure class analysis. The most dominant discriminator for RPN analysis is the occurrence parameter. There is some distinct differentiation between severity and detection as well, but not nearly as drastic as occurrence. The following sections present the results of each RPN parameter individually.

OCCURRENCE PARAMETER

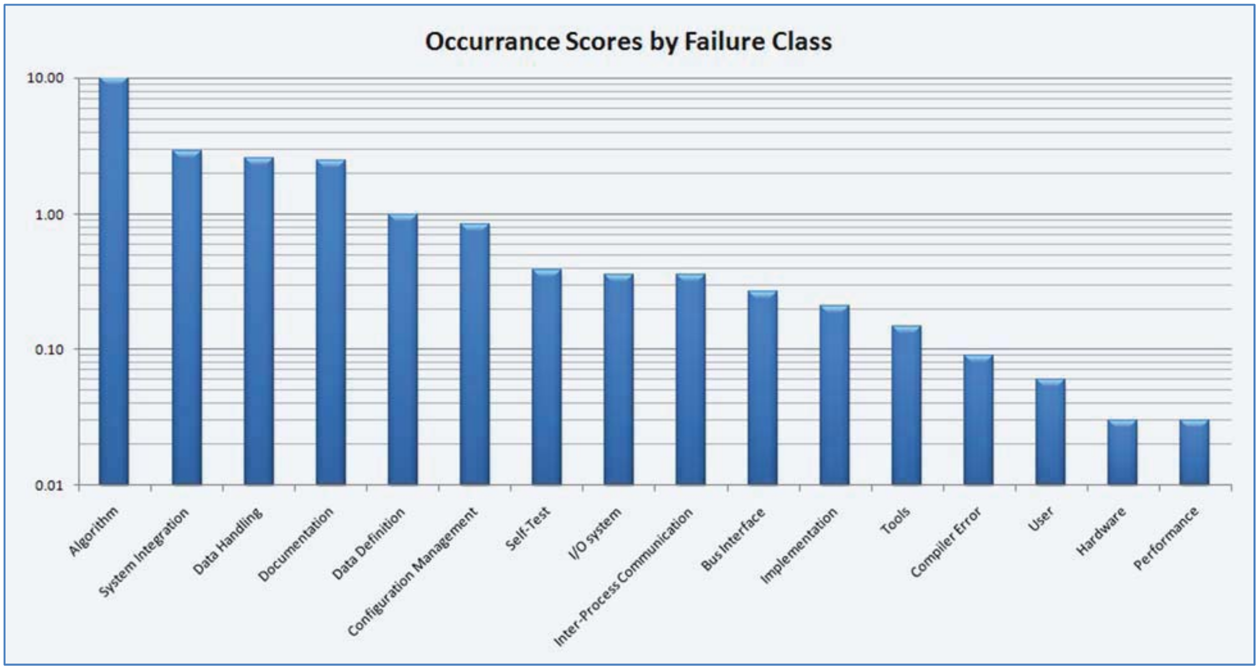


Figure 3 – Occurrence Dimension

The occurrence parameter is the most discriminating factor of all the failure classes. Figure 3, above, shows the breakdown by failure class. Note that there are several displacements from the raw RPN breakdown. This is because, although some errors are more frequent than others, they may not be as severe or as hard to detect—which justifies the failure analysis across the three fundamental dimensions of occurrence, severity, and likelihood of detection.

SEVERITY PARAMETER

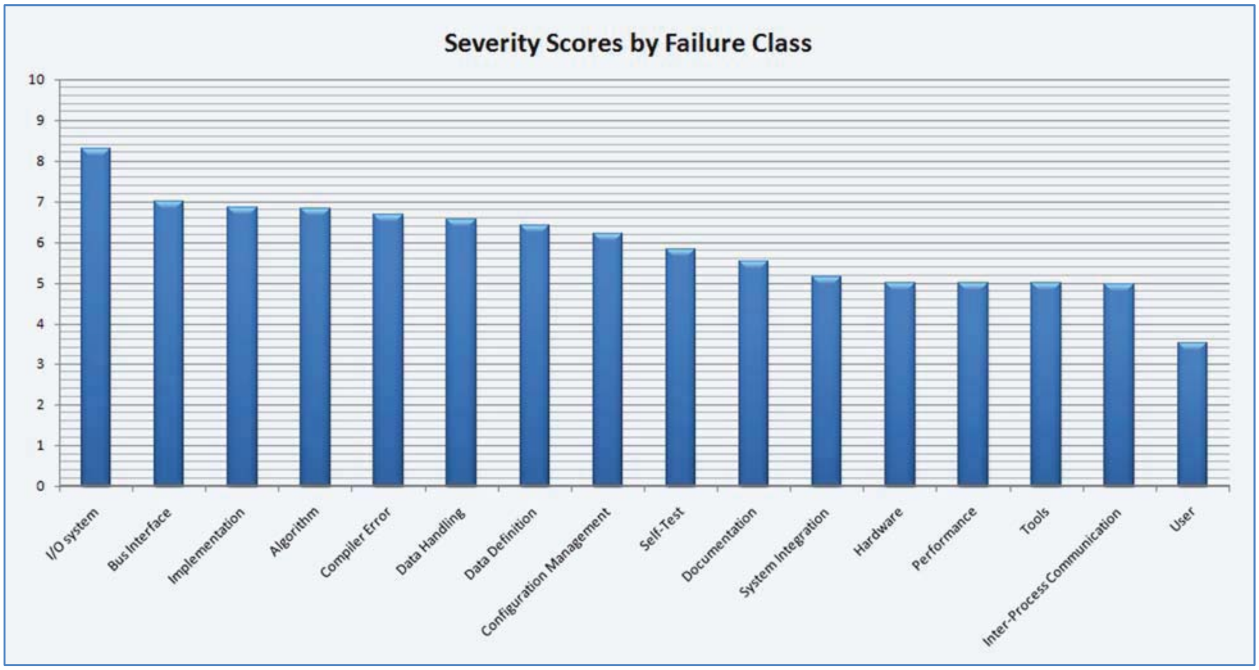


Figure 4 – Severity Dimension

The severity dimension, illustrated in Figure 4 above, shows that the dominant failure class is I/O system. That is, most errors in this class are likely to affect safety of flight—resulting in grounded aircraft or specific operating limits.

DETECTION PARAMETER

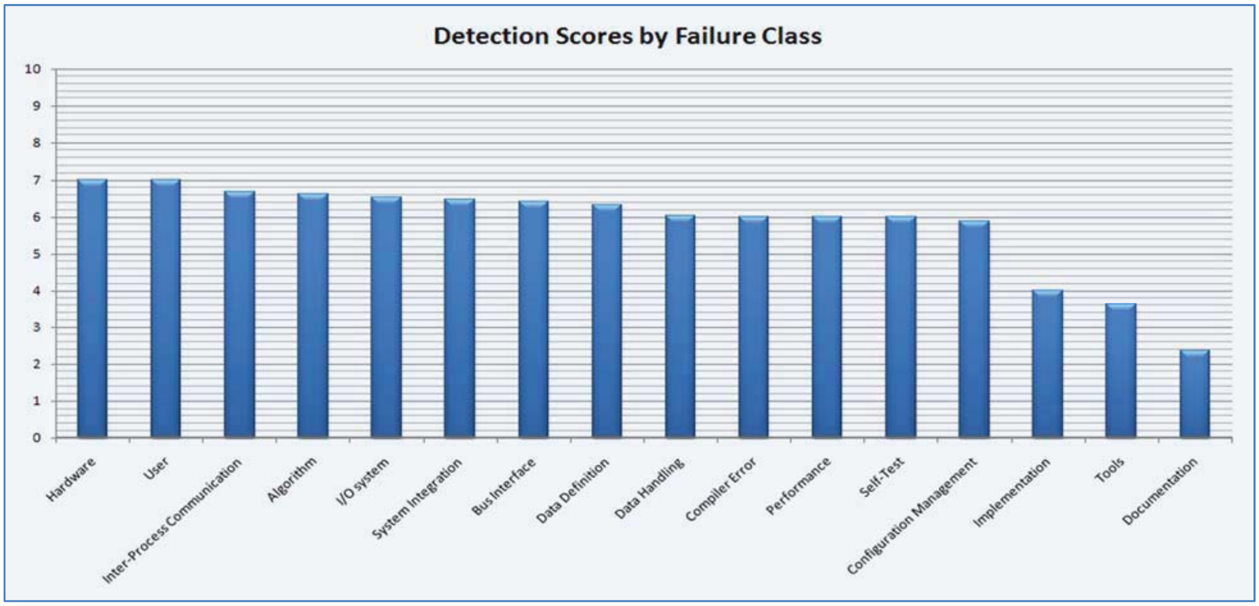


Figure 5 – Detection Dimension

The detection parameter also offers some useful insight into the nature of the errors. Figure 5 shows that hardware and user errors exist longest in the development cycle, while implementation, tools, and documentation error types are detected rather quickly.

ALGORITHM ERROR CLASS PROFILE

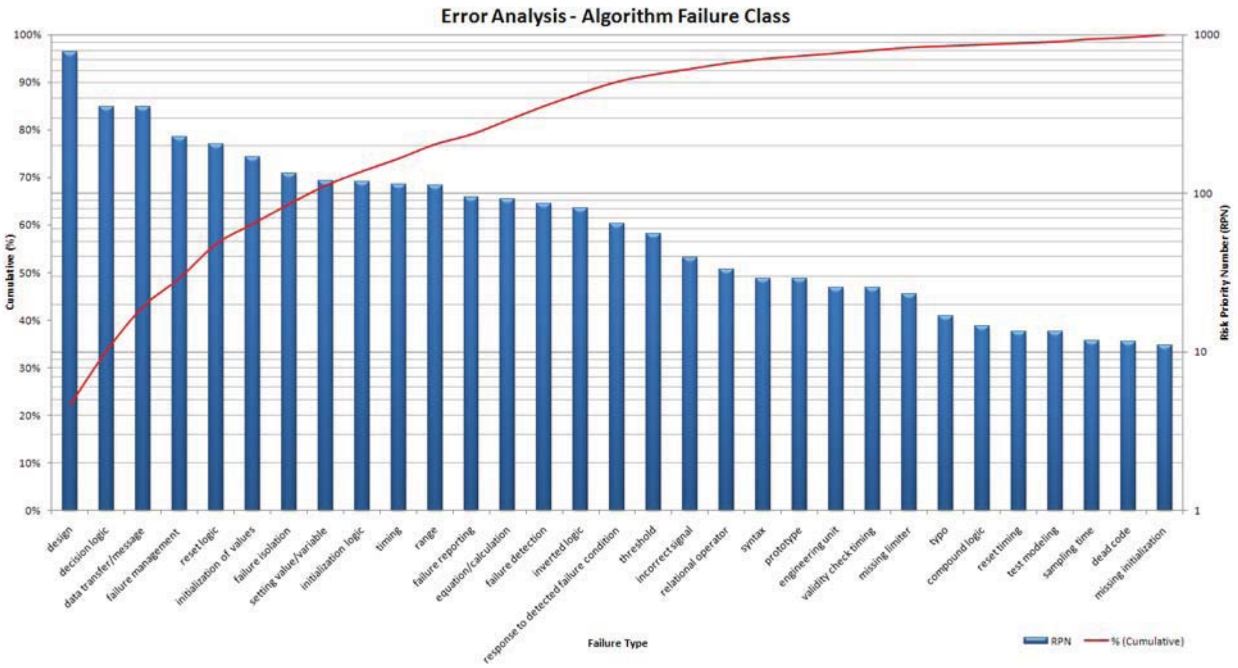


Figure 6 - Algorithm Error Profile

Considering the *Algorithm* failure class, overall algorithm design has the highest RPN and also accounts for 22% of all algorithm errors. Decision logic and data transfer/messaging components come in next; where the top three combined account for nearly half of all the algorithm errors.

Some examples of an *Algorithm* error may be: incorrect power-up or initialization routines after a reset that cause failure monitors to trip in another module; good-channel average selection algorithms that inadvertently include the bad signal in the calculation; or perhaps a set of limit values that are not used when different loading or air vehicle configurations are selected from another subsystem. In hindsight, these types of errors may seem obvious and may lead one to believe more unit-testing is required. The reality is, however, that these types of errors may be so embedded in the algorithm that unit tests would not exercise the unforeseen states properly. Consider the case of the limiter value switching algorithm. A unit test may verify that the set of limits is properly switched under all conditions through which a request may be made. But if the logic in the algorithm is designed to never make the proper request, the limit set is never switched.

This report is not intended to provide philosophical or anecdotal justification of the data presented; however this particular case is considered at length in [3]. Essentially, proper algorithm design requires intimate knowledge of the environment in which the software is to operate as well as sufficient domain knowledge to consider purposeful or inadvertent changes to that environment. This study reveals the gravity of this error class and recommends that technologies be developed to address it.

BUS INTERFACE ERROR CLASS PROFILE

The bus interface errors we studied all have an RPN lower than 100, but greater than 10. Based on the entire set of data represented in this study, RPN values between 10 and 100 could be considered *medium-risk*, where RPN values lower than 10 represent *low-risk* items. The distribution of error reports classified as interface error types are fairly evenly distributed across the specific types within the class, as identified by the cumulative percentage line in red.

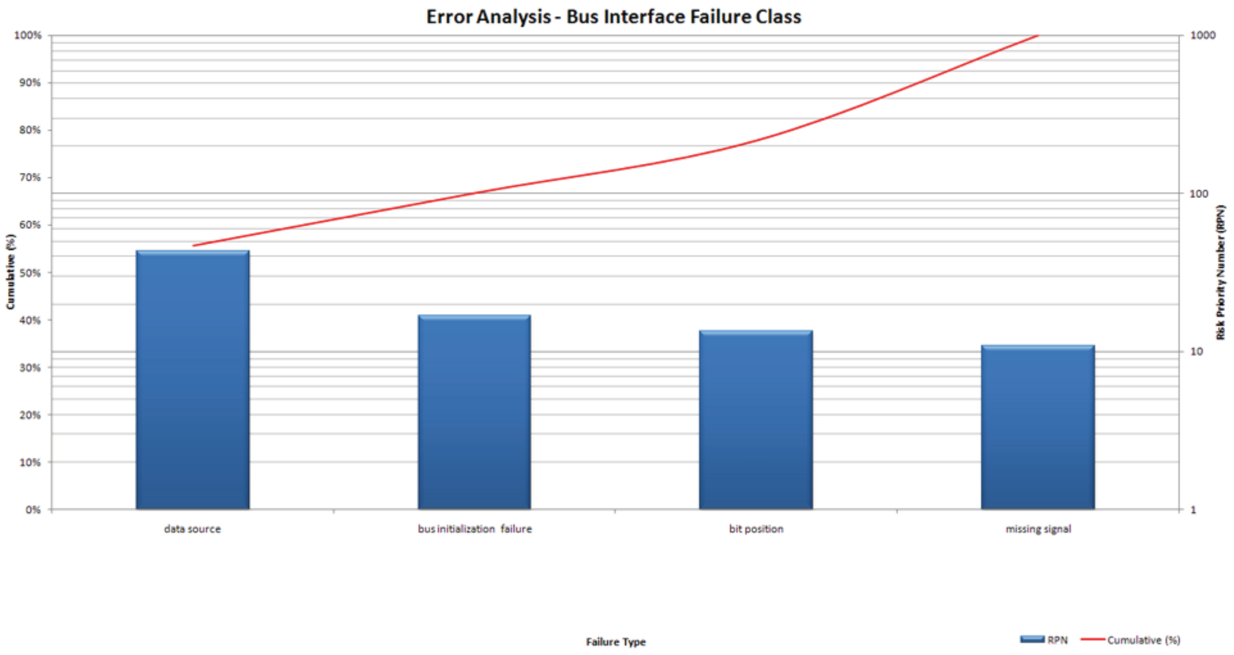


Figure 7 – Bus Interface Error Profile

CONFIGURATION MANAGEMENT ERROR CLASS PROFILE

All CM errors are in the medium-risk RPN range. Many of these errors can be addressed by existing processes.

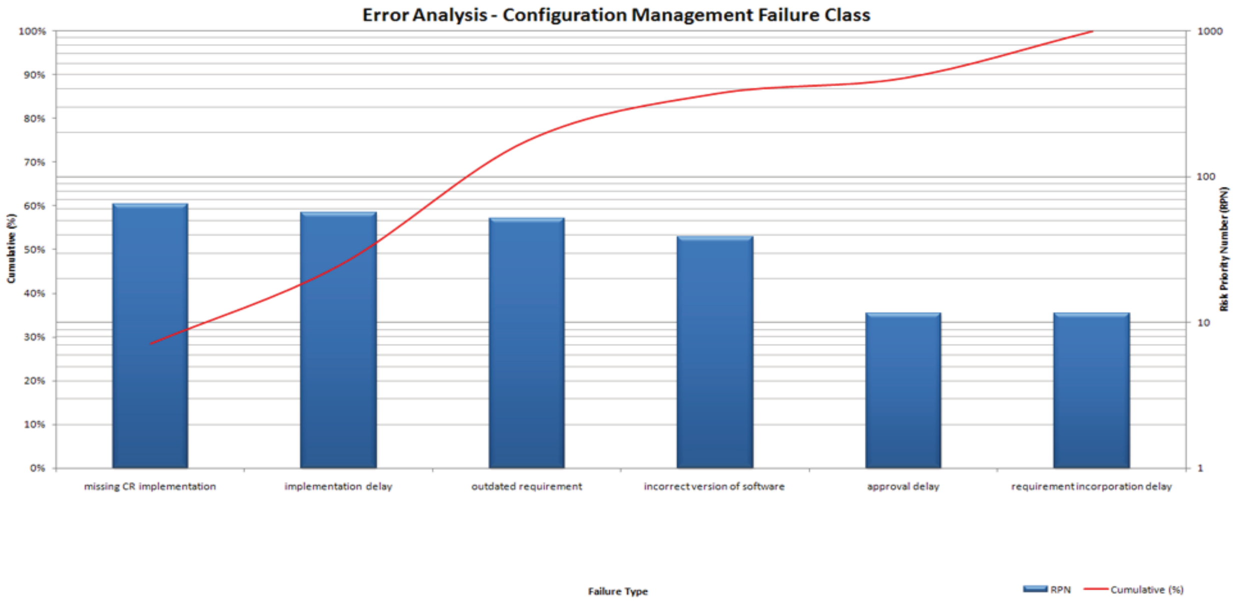


Figure 8 – Configuration Management Error Profile

DATA DEFINITION ERROR CLASS PROFILE

Data definition errors are also medium-risk errors and can be addressed earlier by more detailed data and interface models.

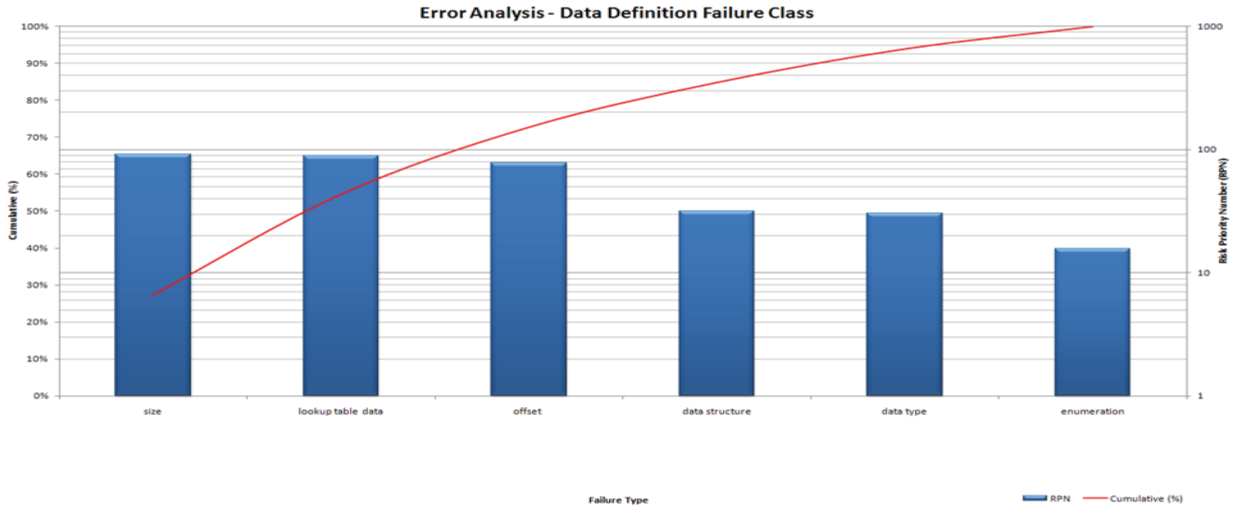


Figure 9– Data Definition Error Profile

DATA HANDLING ERROR CLASS PROFILE

The two high-risk error types for the data handling error class are: scaling factor and memory address. This is essentially the interface between subsystems and can be addressed with more detailed interface modeling and design verification techniques.

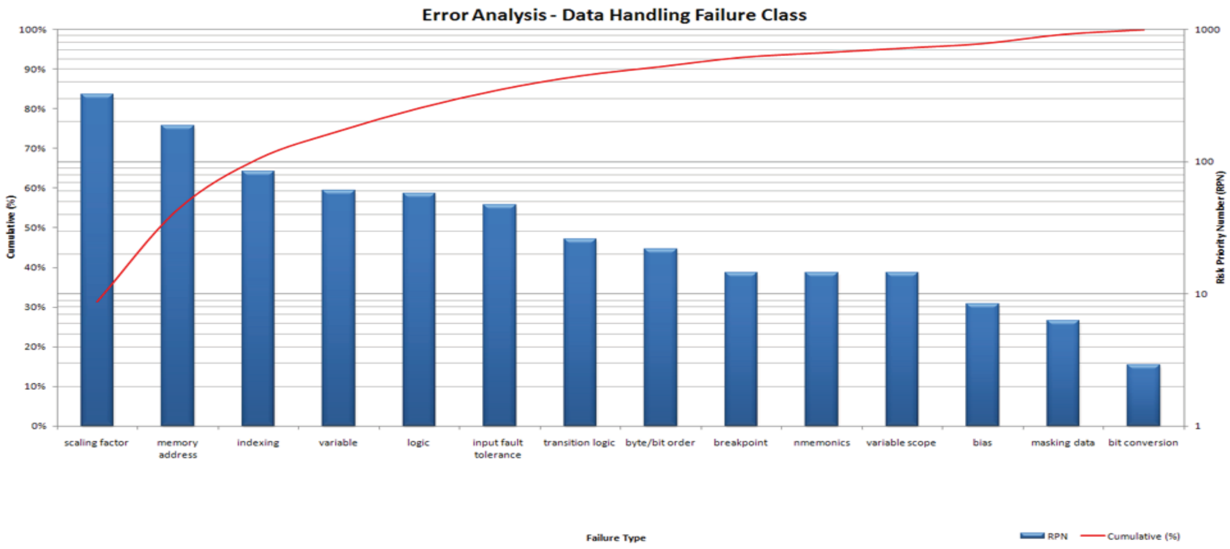


Figure 10 – Data Handling Error Profile

INTER-PROCESS COMMUNICATION ERROR CLASS PROFILE

IPC errors are generally low-risk. Timing and synchronization errors can practically be caught only in a lab environment, although formal analysis and design verification can address several of the others.

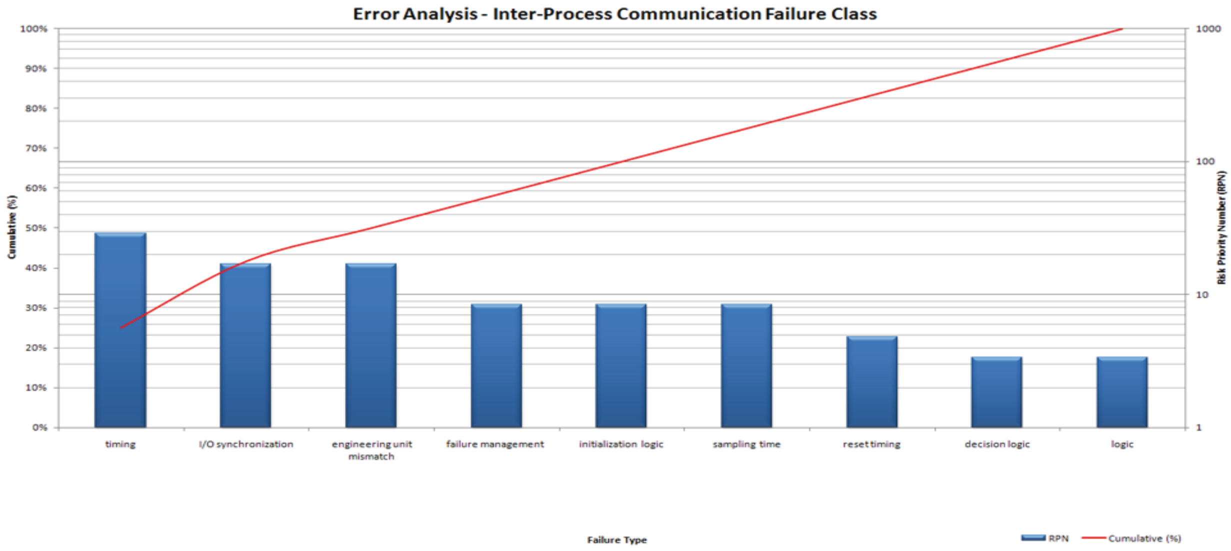


Figure 11 – IPC Error Profile

INPUT/OUTPUT SYSTEM ERROR CLASS PROFILE

I/O errors are generally difficult to find during development and exist for a significant time in the product lifecycle. More detailed and realistic modeling could address these issues, but would require a detailed cost-benefit analysis to determine break-even points for mitigating the risk.

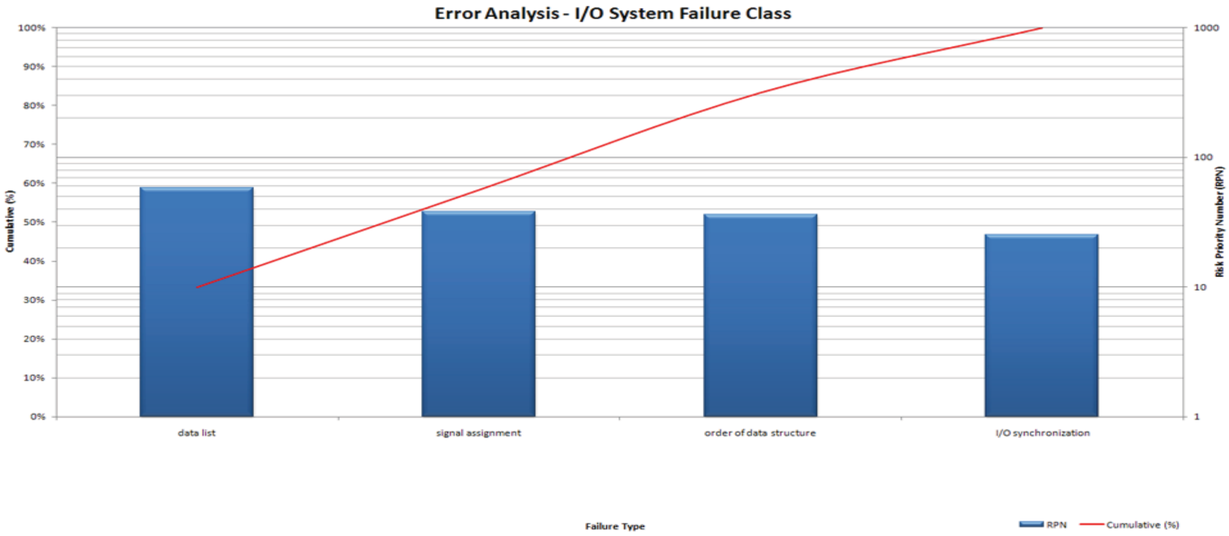


Figure 12 – I/O System Error Profile

SELF-TEST ERROR PROFILE

Self-test errors are of marginal concern and could be addressed through process and technique.

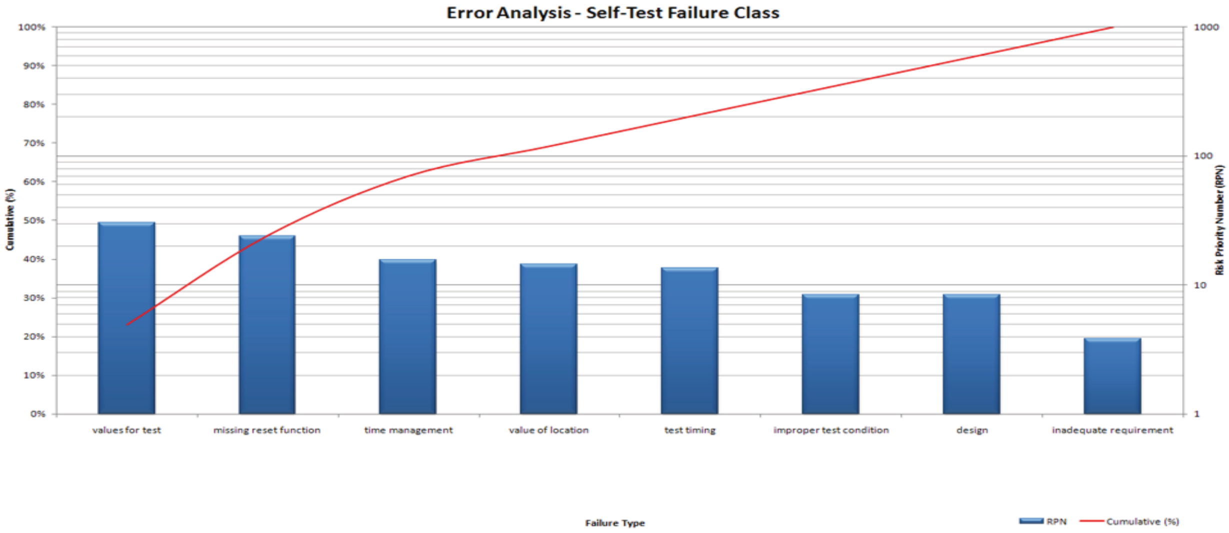


Figure 13 – Self-Test Error Profile

SYSTEM INTEGRATION ERROR CLASS PROFILE

The system integration class contains many specific failure types. This observation in itself shows that a significant amount of errors, in general, are of this class. Although software may work well in individual modules or unit-test levels, it is when the modules are integrated with a larger system that all of the environmental assumptions and erroneous invariants begin to surface. This error class requires an entire dedicated study, as the root of the errors lie in the original requirements and specifications that needed interpretation.

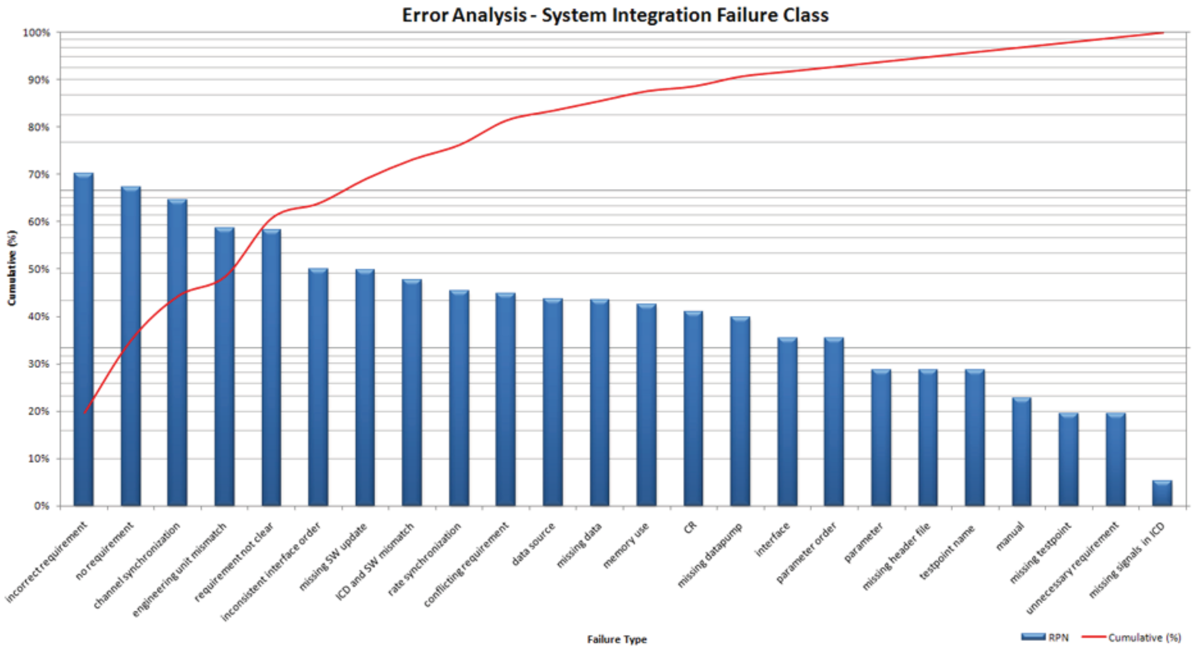


Figure 14 – System Integration Error Profile

ROOT FAILURE CAUSE AND EFFECT RELATIONSHIP ANALYSIS

Having calculated the RPN for the Fundamental failure types, we moved our focus from individual risk assessment to examining the relationships between the fundamental failure types. We made charts to show the relationships. This section describes the root failure cause and effect relationship charts and our analysis on it.

BACKGROUND

When we were working on the failure type taxonomy, we realized that some of the failure types have cause and effect relationships. For instance, the failure types of “algorithm: initialization of values”, “algorithm: timing”, and “algorithm: initialization logic” would all be related in the failures of initializing correctly to start a new mode during a mode transition. This has shown up in concrete examples where a process switched into a new mode before another process generating inputs had switched to the new mode. In this case, the analysis engineers would record the defect in one of the three failure types but it is a mistake to consider that failure type in isolation from the other two. We constructed diagrams indicating the failure types that we should consider together. We connected related failure types by arrows. The direction of the arrows is from the broader scoped failure type to the more specific failure type. Then we pulled together the connected parts into logical groupings centered on the largest of the 17 failure classes. Several of the 17 failure classes ended up split between logical groupings.

GROUND RULES

1. The relationships were not necessarily direct cause-effect relationships, but were rather a logical correlation between the two.
2. An error or confusion in one area might tend to imply an error or confusion in the related area.
3. Each failure type appears only once in the diagrams. We split the diagrams so that no relationships were lost. Only the requirements class appears in multiple diagrams to indicate where the requirements come into those diagrams.
4. We color coded the 114 failure types to indicate their RPN percentile among the failure types by:

   - Red = 5% Highest RPN failure types

   - Orange = Next 10% RPN failure types

   - Yellow = Next 15% RPN failure types

   - Blue = Next 20% RPN failure types

   - Green = Remaining Lowest 50% RPN failure types

   In this report we call these the "RPN percentile groups". The red and orange blocks are the "high-RPN" failure types. The yellow and blue blocks are the "medium-RPN" failure types.

OVERVIEW OF ROOT FAILURE CAUSE AND EFFECT RELATIONSHIP CHART

We organized the 114 failure types into related items and formed seven logical groups. The seven logical groups are Requirement, Configuration Management (CM), External Problems, Documentation, Algorithm, System Integration/Communication, and Self-Test.

Figure 15 shows the top-level organization of these seven groups. The “Requirements” category is at the center because it affects virtually all of the other categories. “External Problems” category does not consist exclusively of software problems but they are problems that require software modification to overcome them. The “Algorithm” category is the largest and contains a concentration of high-RPN failure types. “System Integration/Communication” is also a large category with some high-RPN failure types. The “Self-Test” category has no high-RPN failure types. “Documentation” was a large category only because we did not sub-divide it. We left the “Configuration Management” category as a stand-alone item because it involves every step in the software development process. We can look at the “Configuration Management” category as a *process* problem that runs parallel with other categories of problems. For its small size, it has a large number of medium-RPN failure types.

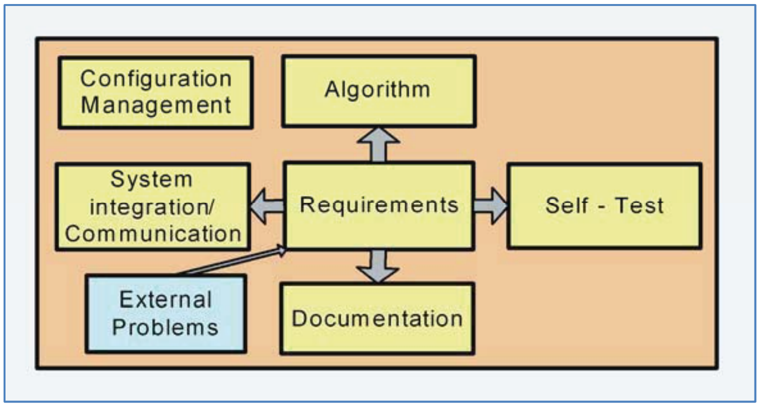


Figure 15 – Related Root Failure Categories

Here is the number of different RPN percentile groups in each category:

Requirements: 2 orange, 1 yellow, 1 green
CM: 2 yellow, 4 blue, 2 green
External problems: 1 blue, 3 green
Documentation: 1 red
Algorithm: 3 red, 9 orange, 6 yellow, 8 blue, 20 green
System Integration/Communication: 1 red, 1 orange, 7 yellow, 8 blue, 17 green
Self-Test: 1 yellow, 2 blue, 13 green

DOCUMENTATION AND EXTERNAL PROBLEMS CATEGORY

Figure 16 shows the Documentation category. Documentation errors are in the top 5% RPN due to the rate of occurrence. These failures accounted for over 11% of the total failures. The severity score was average and the detection score was low (meaning they were easy to detect and were removed quickly). We did not analyze or sub-divide this failure type category. We did not try to analyze the relationships between these failures and others. We did not try to determine if other failures influenced the documentation errors or vice-versa. There might be some connection between them.

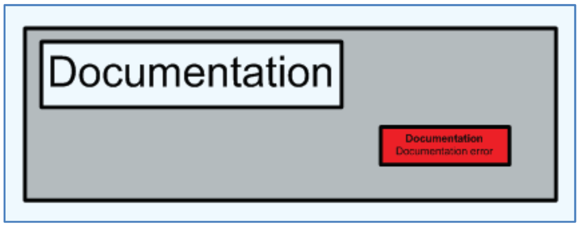


Figure 16 – Documentation Category

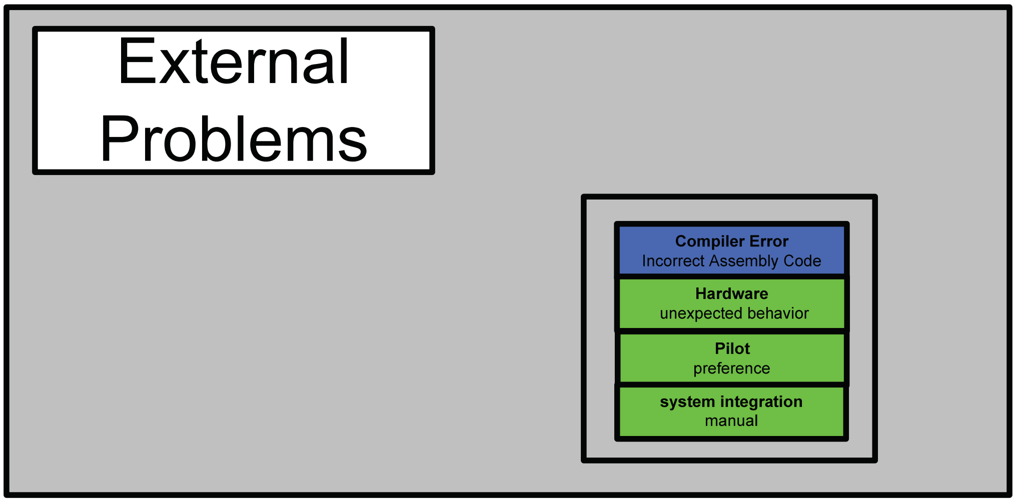


Figure 17 – External Problems Category

Figure 17 shows the External Problems category. It is a "Catch-All" category for a small number of problems. The root causes of these failures are all external to the core software development process of the application code. They are primarily are due to requirements for the application software to mitigate unexpected failures in other areas. Except for "Compiler Error: Incorrect Assembly Code", all these failure types are in the low-RPN range (green). The "Compiler Error: Incorrect Assembly Code" has unremarkable severity and detection scores. The "Pilot: preference" failure type is due to test pilots not agreeing or changing their preference. It has a low severity score but a relatively high detection score. None of these failure types has a high occurrence rate, but their detection scores are high. The "system integration: manual" refers to errors in the flight manual. This failure type has an especially high detection score although its severity score is low.

REQUIREMENTS CATEGORY

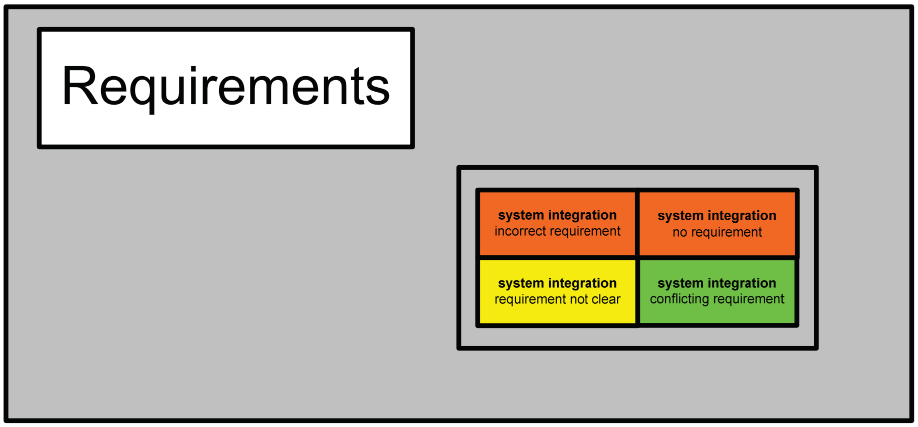


Figure 18 – Requirements Category

Figure 18 shows the Requirements category. These are all system integration problems. Requirements rarely conflict and are usually clear enough. They are more likely to be missing or incorrect. There are two high-RPN failure types. The RPN differences of the Requirements category are mostly due to the rate of occurrence. There are no clear relationships between these failure types or with any other failure types.

CONFIGURATION MANAGEMENT CATEGORY

Figure 19 shows the Configuration Management category. Most of these failures are related to Change Request (CR) process delays and their impact on system integration. This category has two yellow failure blocks and several blue blocks. It is a significant failure category. The RPN differences of the Configuration Management category are mostly due to the rate of occurrence. This is the first category with relationships between failure types. Several "system integration" failure types appear in this diagram because of their relationships with the "configuration management" failure types. The two yellow blocks, "CM: implementation delay" and "CM: missing CR implementation" are grouped together with the green "CM: requirement incorporation delay" to collect the problems with delays in already approved changes. This collection relates to several "system integration" failure types, all having to do with incompatible software or interfaces. The "system integration: missing SW update" failure type can be caused by the "CM: implementation delay", or "CM: missing CR implementation" failure types. The same relationship is true for the "system integration: inconsistent interface order" and "system integration: ICD and SW mismatch" failure types. The green "CM: approval delay" is green because it does not occur often, but its severity score is high. It can contribute to the "CM: incorrect version of software" failure type, which is blue.

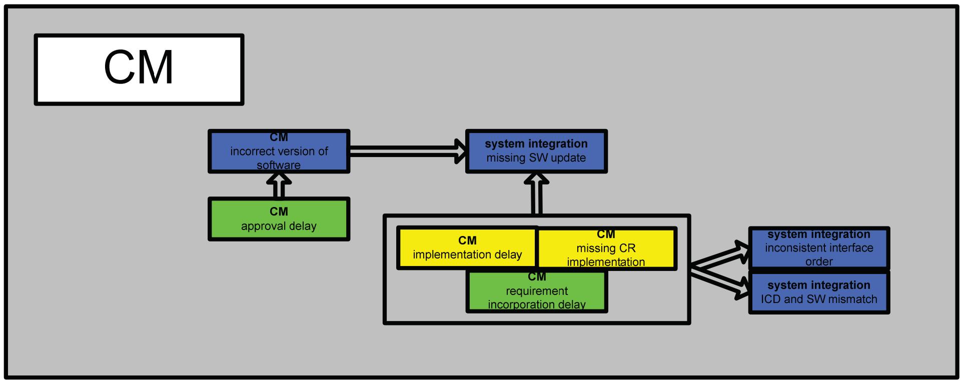


Figure 19 – Configuration Management Category

ALGORITHM CATEGORY

Figure 20 illustrates the Algorithm category. This is a significant and interrelated category of failure types. It shows the relationship between algorithm design, inter-process communication, and requirements category. It is the most significant collection of related failure types. It includes the top two RPN-ranked failure types, "algorithm: design" and "algorithm: decision logic". The "algorithm: design" failure type alone accounts for over 10% of all the root failures in the study. The next highest is "algorithm: decision logic", which accounts for over 5% of all the root failures in the study. The final red root failure type in the diagram is "algorithm: failure management". This type involves the logic of signal redundancy, selection, and verification. It accounts for about 3% all the root failures. The designs in that system should not require a great deal of modification in the normal design loop. Another noticeable part of the Algorithm diagram is the three related orange failures of "algorithm: initialization logic", "algorithm: timing", and "algorithm: initialization of values". Together these are over 4% of all the root failures. This failure type includes problems in timing of initializations when modes change and the inputs are not correct for the new mode. In addition, state variables may not have been reset correctly when new mode started running. Several of the failure types group together. In the upper left of the diagram is a set of three signal definition problems, “data definition: lookup table data”, “algorithm: incorrect unit”, and “algorithm: incorrect signal”. These are problems which are interior to the algorithm but they can be influenced by the “system integration” fault types of “system integration: missing data” or “system integration: engineering unit mismatch”. This set of failure types can cause “algorithm: equation/calculation” failure types. Another significant collection of failure types deals with the range processing of signals. It consists of the “algorithm: range”, “algorithm: threshold”, and “algorithm: missing limits” failure types. This set also can influence the “algorithm: equation/calculation” failure type. One set of failures which is unrelated to other failures is the set of random “mutation” type failures, “algorithm: syntax”, and “algorithm: typo”. Usually the compiler detects these types of errors immediately but the ones that slip through can be very difficult to detect. It is difficult for the compiler to detect a variable name typo that ends up matching the wrong, but otherwise valid, variable. It is also difficult for compilers to spot the “if(A = B)” vs. “if(A == B)” problem unless the first one is specifically disallowed. These failures can go undetected for a long time. We have also included “algorithm: dead code” in this set although it may have relationships to CM failure types which we have not established yet. The “algorithm: reset timing” failure type is green. It has a low occurrence rate but a high severity score. It is influenced by the “algorithm: reset logic” failure type, which is orange due to a high occurrence rate. The “algorithm: reset timing” failure type is secondary to the “algorithm: reset logic” failure type. There is a significant set of discrete logic problems consisting of (listed in order of decreasing RPN) “algorithm: decision logic”, “algorithm: inverted logic”, “algorithm: relational operator”, and algorithm: compound logic”. The “algorithm: decision logic” failure type is red due to its high rate of occurrence. It may include some failures that belong in the other more specific logic categories if we examined them further. These failures are largely self-initiated due to the complexity of the logic and do not have relationships to other failure types. They are structural / discrete logic defects that may be detected if formal methods can be applied. Toward the right of the diagram are several failure management / failure reconfiguration blocks. Many of these are have significant RPN values. The entire collection is “algorithm: failure detection”, algorithm: failure reporting”, “algorithm: failure management”, “algorithm: failure isolation”, “algorithm: response to detected failure condition”, “interprocess communication: failure management”, “data handling: input fault tolerance”, and “bus interface: bus initialization failure”. At the lower left of the diagram is a large collection of low-RPN green/blue blocks dealing primarily with interprocess communication timing problems. The red “algorithm: design” block has already been discussed.

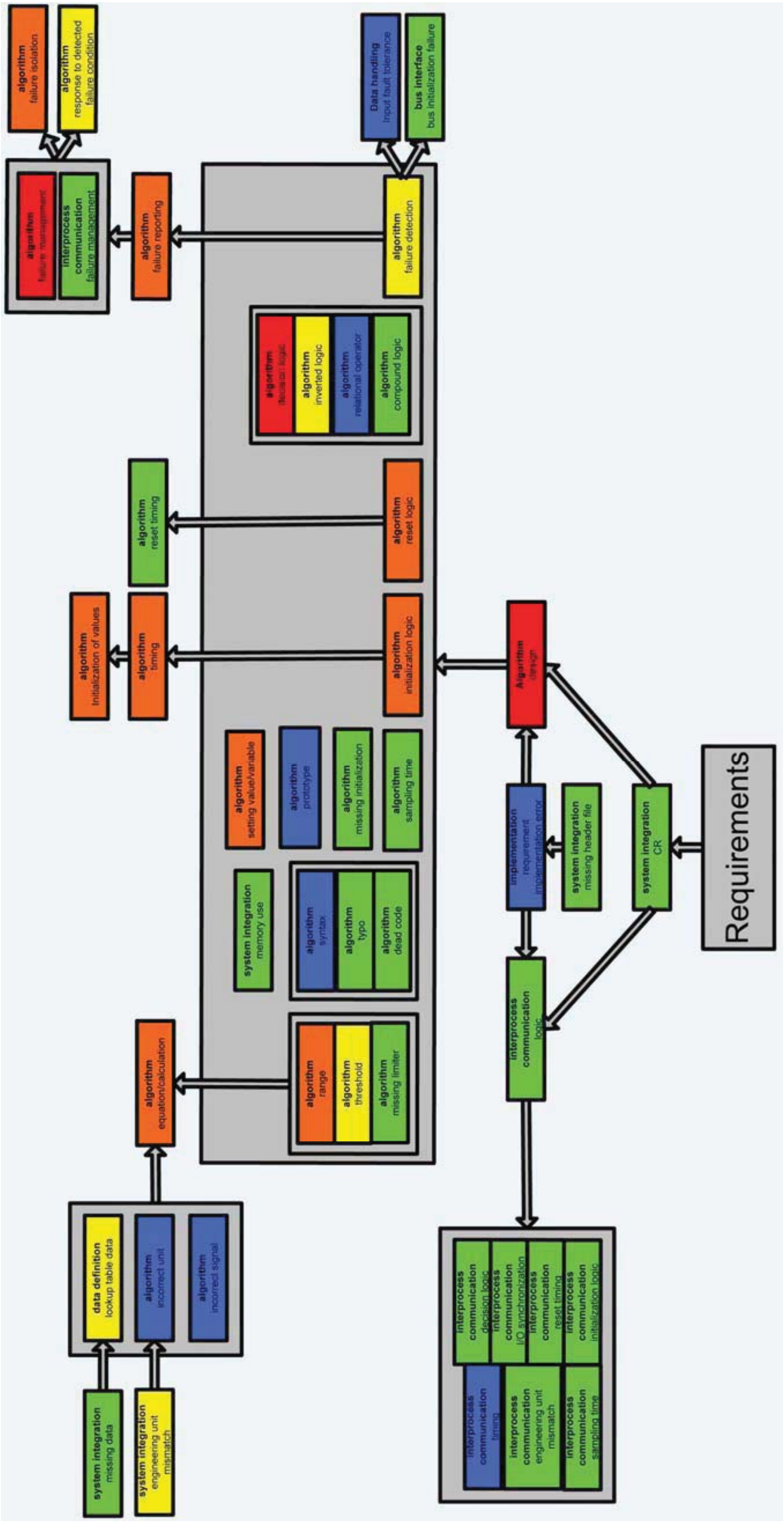
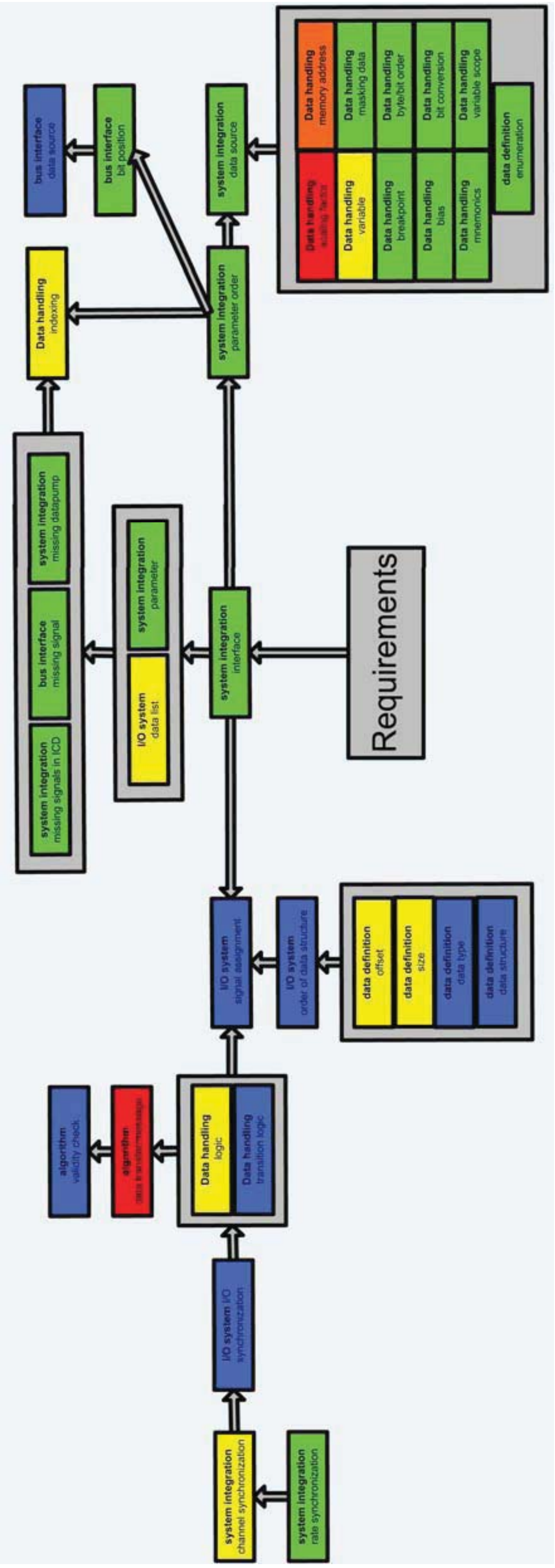

algorithm failure isolation
algorithm response to detected failure condition
algorithm failure management
interprocess communication failure management
algorithm failure reporting
algorithm failure detection
Data handling Input fault tolerance
bus interface bus initialization failure
algorithm decision logic
algorithm inverted logic
algorithm relational operator
algorithm compound logic
algorithm reset timing
algorithm reset logic
algorithm Initialization of values
algorithm timing
algorithm initialization logic
Algorithm design
algorithm setting value/variable
algorithm prototype
algorithm missing initialization
algorithm sampling time
implementation requirement implementation error
system integration missing header file
system integration CR
Requirements
system integration memory use
algorithm syntax
algorithm typo
algorithm dead code
interprocess communication logic
algorithm equation/calculation
algorithm range
algorithm threshold
algorithm missing limiter
system integration missing data
system integration engineering unit mismatch
data definition lookup table data
algorithm incorrect unit
algorithm incorrect signal
interprocess communication decision logic
interprocess communication I/O synchronization
interprocess communication reset timing
interprocess communication initialization logic
interprocess communication timing
interprocess communication engineering unit mismatch
interprocess communication sampling time

Figure 20 – Algorithm Category

SYSTEM INTEGRATION / COMMUNICATION CATEGORY

Figure 21 shows the System Integration / Communication Category. It includes a significant number of high/medium RPN failure types and includes many relationships.



bus interface data source
bus interface bit position
system integration data source
Data handling memory address
Data handling masking data
Data handling byte/bit order
Data handling bit conversion
Data handling variable scope
Data handling scaling factor
Data handling variable
Data handling breakpoint
Data handling bias
Data handling mnemonics
data definition enumeration
Data handling indexing
system integration parameter order
system integration missing datapump
bus interface missing signal
system integration missing signals in ICD
system integration parameter
I/O system data list
system integration interface
Requirements
I/O system signal assignment
I/O system order of data structure
data definition offset
data definition size
data definition data type
data definition data structure
algorithm validity check
algorithm
Data handling logic
Data handling transition logic
I/O system I/O synchronization
system integration channel synchronization
system integration rate synchronization

Figure 21 – System Integration / Communication Category

The high-RPN root failures here are “algorithm: data transfer/message”, “data handling: scaling factor”, and “data handling: memory address”, which account for about 4%, 4%, and 3% of the all the root failures, respectively. These data dictionary interface problems can be dealt with using system engineering tools such as SysML or AADL. The tools should be system-wide. Part-task interface controls do not have the same benefits unless they are coordinated. The “data handling: scale factor” failure type points to the difficulty of tracking fixed-point scaling correctly through all the engineering units, hardware interfaces, etc. The engineering disciplines use different units when they address fixed point scaling and bias. Electrical diagrams will have Volts, current, and other engineering units. Software engineers want least significant bit (LSB) values, full range max/min, etc. And all are further complicated by biases, both physical and computational, along the way. Possibly engineers need a tool to help with fixed-point range, bias, scale, engineering units/LSB, etc. Several system integration / communication blocks have already appeared in other diagrams where they had significant relationships with the blocks there. We divided the diagrams so that no relationships were broken. All the blocks here connect to the main diagram. The red “algorithm: data transfer/message” failures can be caused by the set of “data handling: logic” and “data handling: transition logic”. They can, in turn, cause “algorithm: validity check” failures. In the upper, center of the diagram is a collection of missing interface items, “system integration: missing signals in ICD”, “bus interface: missing signal”, and “system integration: missing datapump”. These are all green blocks and are not very significant. They can be caused by the “I/O system: data list” failure type which is yellow due to a high severity score. In their turn, they can contribute to the “data handling: indexing” failure type, which is yellow due to a high occurrence rate. This reflects problems caused by shifting data when a signal is missing. In the bottom left of the diagram is a collection of medium-RPN data definition failure types. They are “data definition” offset, size, data type, and data structure. The final large collection of failure types is the data handling collection to the bottom right of the diagram. These are data dictionary issues. The “data handling: scaling factor” and “data handling: memory address” failure types are the most significant by far. They have been discussed above.

SELF-TEST CATEGORY

Figure 22 shows the Self-Test Category. There are no high-RPN root failures here and only three medium-RPN failure types. The most serious root failure is the yellow “outdated requirement” root failure which accounts for slightly over 1% of all the root failures. There are two blue failure types, “self-test: values for test” and “tools: algorithm”. These reflect the problem of generating “truth data” from the tools for use in the self-test. All the rest of the blocks are green. At the top, center of the diagram are a collection of top-level design problems. They are “self-test procedure: missing reset function”, “self-test: test timing”, “self-test: time management”, and “performance: exceed processor utilization target”. At the center, right are two green blocks that reflect the need to include testpoints in the code for monitoring or test value insertion. They are the “system integration: missing testpoint”, and the “system integration: testpoint name” failure types. At the bottom, left of the diagram are two requirements issues: outdated and unnecessary. At the bottom right of the diagram are several issues with modeling and generating valid truth data.

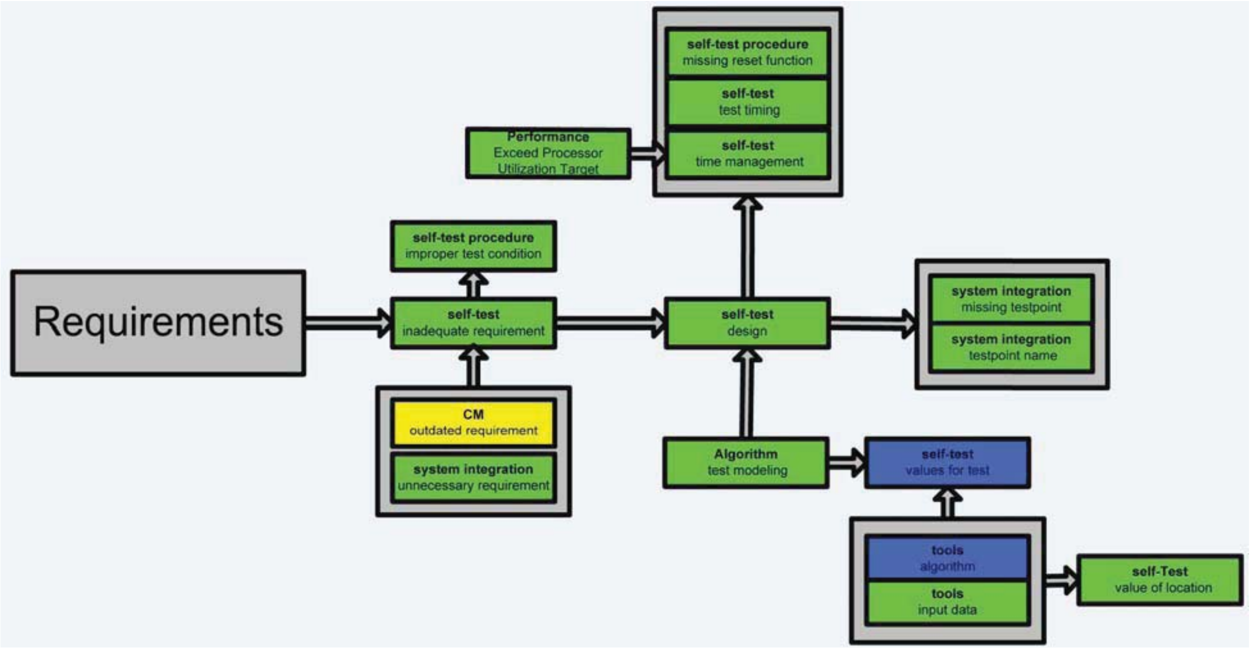
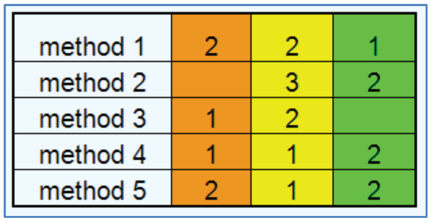


Figure 22 – Self-Test Category

APPLICATION OF DATA ANALYSIS RESULTS TO EVALUATING FUTURE TECHNOLOGIES

The data analysis results can be used to analyze the impact of the technologies, for example, possibly applying formal methods to the algorithms. Looking at figure 20, the algorithm-related defects are a mixture of discrete logic errors like “algorithm: decision logic” and floating-point calculation errors like “algorithm: design”. An application of formal methods could be used to identify and remove discrete logic defects in the early development stages. In figure 20, formal methods would reduce the number of errors in “algorithm: decision logic", "algorithm: failure management", "algorithm: initialization logic". An adjustment could be made in the Occurrence or Detection numbers for those entries in the RPN calculations. Under the System Integration / Communication section, the collection of data handling failures points to the possible benefit of an automated data-dictionary driving the interface generation tools. Additionally, evidence points to the benefits of having model based design tools that encompass the entire system. In particular, requirements failure types may be reduced by using system level design tools like SysML or AADL. Conflicting or imprecise requirements would be spotted by Formal Methods where it could be applied. In general figure 20, shows that the data dictionary information is a problem (size, location, address, bit order, etc). However, it is very hard to find a single technology that covers the entire problem space.



method 1	🟧 2	🟨 2	🟩 1
method 2	🟧	🟨 3	🟩 2
method 3	🟧 1	🟨 2	🟩
method 4	🟧 1	🟨 1	🟩 2
method 5	🟧 2	🟨 1	🟩 2

Figure 23 – Related Root Failure Categories

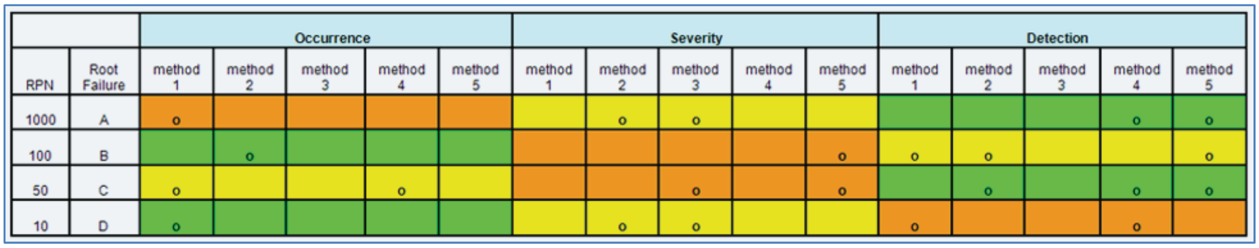
However, it is believed with high confidence that a significant number of software problems can be reduced before entering the next phase of the program by identifying the correct combination of technology to cover the problem space.

Here is one example of how the data analysis results can be used to identify possible combinations of technologies for software health management:

1. Create Matrix of evaluation of technologies with each root failure.

   A. Select technologies/ methods that you want to examine.
   B. Prepare a table that contains information of the RPN and which factor is the most and the least dominating factor of the RPN. (Color Code in example. Orange = the most dominant factor, Yellow = 2nd dominant factor, and Green = the least dominant factor)
   C. Evaluate all the Technologies/Methods chosen with respect to the occurrence, severity, detection of each root failure. (Figure 23 illustrates this process)

2. Evaluate each Technology/Methods by affectability with respect to the most and least dominant factor of the RPN. (Figure 24 is the example of this process)



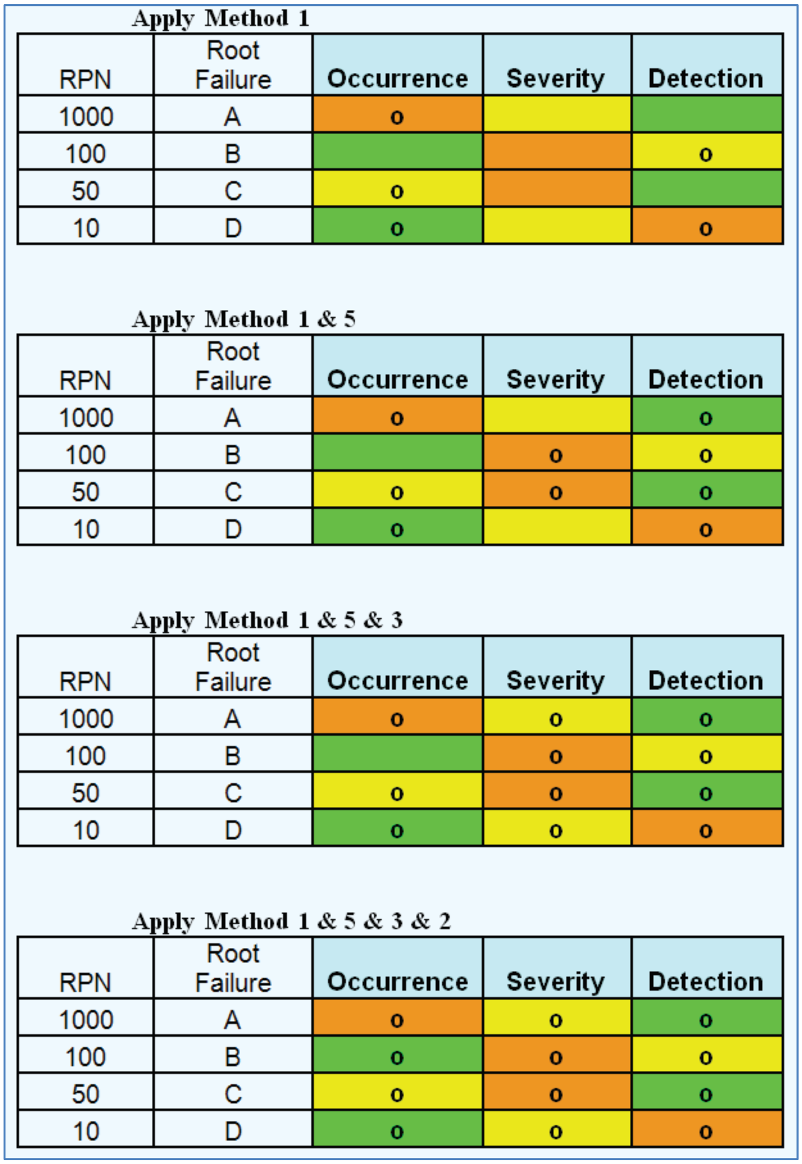
		Occurrence					Severity					Detection				
RPN	Root Failure	method 1	method 2	method 3	method 4	method 5	method 1	method 2	method 3	method 4	method 5	method 1	method 2	method 3	method 4	method 5
1000	A	🟧 o	🟧	🟧	🟧	🟧	🟨	🟨 o	🟨 o	🟨	🟨	🟩	🟩	🟩	🟩 o	🟩 o
100	B	🟩	🟩 o	🟩	🟩	🟩	🟧	🟧	🟧	🟧	🟧 o	🟨 o	🟨 o	🟨	🟨	🟨 o
50	C	🟨 o	🟨	🟨	🟨 o	🟨	🟧	🟧	🟧 o	🟧	🟧 o	🟩	🟩 o	🟩	🟩 o	🟩 o
10	D	🟩 o	🟩	🟩	🟩	🟩	🟨	🟨 o	🟨 o	🟨	🟨	🟧 o	🟧	🟧	🟧 o	🟧

Figure 24 – Related Root Failure Categories

3. From Step 2, come up with different combination of Technologies/Methods to use and evaluate them. From Table 2, we can draw conclusions that “method 1” is the most effective for Software Health management method. However, it does not cover all the issues. Figure 23 provides some additional example tables that show how many problems that can be covered with different combinations of Technologies/Methods.

Individuals that are developing methods or tools for software health management and using currently available methods or tools can benefit from this kind of practice.

For the Developer of methods or tools for software health management, this practice can be their assessment, and it will help users identify what kind of methods they are going to use for their project.



Apply Method 1

RPN	Root Failure	Occurrence	Severity	Detection
1000	A	🟧 o	🟨	🟩
100	B	🟩	🟧	🟨 o
50	C	🟨 o	🟧	🟩
10	D	🟩 o	🟨	🟧 o

Apply Method 1 & 5

RPN	Root Failure	Occurrence	Severity	Detection
1000	A	🟧 o	🟨	🟩 o
100	B	🟩	🟧 o	🟨 o
50	C	🟨 o	🟧 o	🟩 o
10	D	🟩 o	🟨	🟧 o

Apply Method 1 & 5 & 3

RPN	Root Failure	Occurrence	Severity	Detection
1000	A	🟧 o	🟨 o	🟩 o
100	B	🟩	🟧 o	🟨 o
50	C	🟨 o	🟧 o	🟩 o
10	D	🟩 o	🟨 o	🟧 o

Apply Method 1 & 5 & 3 & 2

RPN	Root Failure	Occurrence	Severity	Detection
1000	A	🟧 o	🟨 o	🟩 o
100	B	🟩 o	🟧 o	🟨 o
50	C	🟨 o	🟧 o	🟩 o
10	D	🟩 o	🟨 o	🟧 o

Figure 25 – Combining Technologies and Methods

Here are some software development technologies which are of interest in the literature and research:

- Automated Verification Management
- Formal Requirements Specifications
- Requirements and Traceability Analysis
- Formal Methods
- Computer-Aided System Engineering
- V&V Run-Time Design
- Rigorous Analysis for Test Reduction
- Requirements and Design Abstraction
- Probabilistic/Statistical Test
- Testing Metrics

It would be valuable to examine some of these technologies with the new information obtained from this study. Selection of the emerging technologies to be evaluated should be guided by the "lessons learned" in research efforts such as VVIACS (Validation & Verification of Intelligent and Adaptive Control Systems), CerTA FCS CPI (Certification Techniques for Advanced Flight Critical Systems – Challenge Problem Integration), and MCAR (Mixed Criticality Architecture Requirements). Several technologies including Auto-Code, Auto-Test, Rapid Prototyping, System Model-Based, and Simulation-Based Design are mature enough to already be established with recognized benefits.

Future research should include analysis of some additional programs to reflect a larger variety of software development processes.

REFERENCES

[1] Goddard, P.L., "Software FMEA Techniques", *Proceedings of the Annual Reliability and Maintainability Symposium*, January 2000.

[2] Goddard, P.L., "Validating the Safety of Embedded Real-Time Control Systems using FMEA", *Proceedings of the Annual Reliability and Maintainability Symposium*, January 1993.

[3] Jackson, D., Thomas, M., and Millett, L., Eds. *Software for Dependable Systems: Sufficient Evidence?* National Research Council. National Academies Press, 2007.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
01- 05 - 2011

2. REPORT TYPE
Contractor Report

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE
Concept Development for Software Health Management

5a. CONTRACT NUMBER
NNL06AA08B

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Riecks, Jung; Storm, Walter; Hollingsworth, Mark

5d. PROJECT NUMBER

5e. TASK NUMBER
NNL07AB06T

5f. WORK UNIT NUMBER
645846.02.07.07

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/CR-2011-217150

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category 06
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES
Langley Technical Monitor: Eric G. Cooper

14. ABSTRACT

This report documents the work performed by Lockheed Martin Aeronautics (LM Aero) under NASA contract NNL06AA08B, delivery order NNL07AB06T. The Concept Development for Software Health Management (CDSHM) program was a NASA funded effort sponsored by the Integrated Vehicle Health Management Project, one of the four pillars of the NASA Aviation Safety Program. The CD-SHM program focused on defining a structured approach to software health management (SHM) through the development of a comprehensive failure taxonomy that is used to characterize the fundamental failure modes of safety-critical software.

15. SUBJECT TERMS
Software, failure, health management, safety-critical, taxonomy

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	40	(443) 757-5802