



# governmentattic.org

*"Rummaging in the government's attic"*

Description of document: Two National Aeronautics and Space Administration (NASA) Studies on Software Anomalies, 2010-2011

Requested date: 24-October-2020

Release date: 18-December-2020

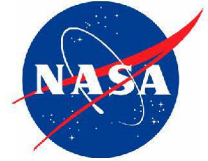
Posted date: 04-January-2021

Source of document: FOIA request  
LaRC FOIA Public Liaison  
NASA Langley Research Center  
MS 151  
Hampton, VA 23681  
Email: [larc-dl-foia@mail.nasa.gov](mailto:larc-dl-foia@mail.nasa.gov)

The governmentattic.org web site ("the site") is a First Amendment free speech web site and is noncommercial and free to the public. The site and materials made available on the site, such as this file, are for reference only. The governmentattic.org web site and its principals have made every effort to make this information as complete and as accurate as possible, however, there may be mistakes and omissions, both typographical and in content. The governmentattic.org web site and its principals shall have neither liability nor responsibility to any person or entity with respect to any loss or damage caused, or alleged to have been caused, directly or indirectly, by the information provided on the governmentattic.org web site or in this file. The public records published on the site were obtained from government agencies using proper legal channels. Each document is identified as to the source. Any concerns about the contents of the site should be directed to the agency originating the document in question. GovernmentAttic.org is not responsible for the contents of documents published on the website.

National Aeronautics and Space Administration

**Langley Research Center**  
100 NASA Road  
Hampton, VA 23681-2199



December 18, 2020

Reply to Attn. of: Office of Communications

REF: NASA's FOIA Case Number 21-LaRC-F-00064

This is in response to your Freedom of Information Act (FOIA) request to the National Aeronautics and Space Administration (NASA) dated October 24, 2020, and received in our office on October 26, 2020. Your request was assigned FOIA Case Number 21-LaRC-F-00064 and was for:

A copy of a deliverable report from Lockheed Martin provided to NASA Langley Research Center in Delivery Order NNL09AD66T and Delivery Order NNL07AB06T of parent contract NNL06AA08B, which were awarded in approximately 2014-2015.

Specifically, I would like to receive the deliverable report(s) on this topic: TASK 1: ABSTRACTION OF CLASSES OF SOFTWARE ANOMALIES. A. CATALOG HISTORICAL AIRCRAFT SOFTWARE ANOMALIES TO INCLUDE REPRESENTATIVE ANOMALIES UNCOVERED DURING PRE-DEPLOYMENT VERIFICATION AND VALIDATION ACTIVITIES AS WELL AS THOSE DISCOVERED POST-DEPLOYMENT. B. DEVELOP ABSTRACTIONS OF THE CATALOGED ANOMALIES ALONG WITH A TAXONOMY OF SOFTWARE FAILURES IN AIRCRAFT SYSTEMS. NASA'S AVIATION SAFETY (AVSAFE) PROGRAM'S INTEGRATED VEHICLE HEALTH MANAGEMENT (IVHM) PROJECT IS PURSUING FOUNDATIONAL RESEARCH IN THE DEVELOPMENT OF TECHNOLOGIES FOR AUTOMATED DETECTION, DIAGNOSIS, PROGNOSTICS, AND MITIGATION OF ADVERSE EVENTS DUE TO AIRCRAFT SOFTWARE. THIS EFFORT IS BEING CONDUCTED UNDER THE SUB-ELEMENT ENTITLED SOFTWARE HEALTH MANAGEMENT IN VERSION 2.0 OF THE IVHM TECHNICAL PLAN.

The NASA's Langley Office of Procurement conducted a search within their database for a copy of a deliverable report from Lockheed Martin using the search terms Delivery Order NNL09AD66T and Delivery Order NNL07AB06T of parent contract NNL06AA08B, that was

awarded in approximately 2014-2015. That search identified 2 reports consisting of 113 pages in response to your request. We have reviewed these responsive records under the FOIA to determine whether they may be accessed under the FOIA's provisions.

Based on that review, this office is providing the following:

113 pages are being released in full (RIF).

You may contact the NASA's Chief FOIA Public Liaison, Stephanie Fox, via telephone at 202-358-1553 or via e-mail at [stephanie.k.fox@NASA.gov](mailto:stephanie.k.fox@NASA.gov) to obtain further assistance or seek dispute resolution services for any aspect of your request. You may also send correspondence to Ms. Fox at the following address:

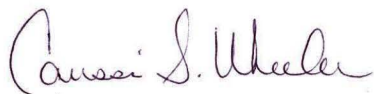
National Aeronautics and Space Administration (NASA)  
Freedom of Information Act Unit  
NASA Headquarters  
Attn: Stephanie K. Fox  
Chief FOIA Public Liaison  
300 E Street, S.W., 5P32  
Washington D.C. 20546  
Fax: 202-358-4332

Additionally, you may contact the Office of Government Information Services (OGIS) at the National Archives and Records Administration to inquire about the FOIA dispute resolution services it offers. The contact information for OGIS is:

Office of Government Information Services  
National Archives and Records Administration  
8601 Adelphi Road-OGIS  
College Park, Maryland 20740-6001  
Email: [ogis@nara.gov](mailto:ogis@nara.gov)  
[Telephone: 202-741-5770](tel:202-741-5770)  
[Toll free: 1-877-684-6448](tel:1-877-684-6448)  
[Fax: 202-741-5769](tel:202-741-5769)

If you have any questions, please contact me electronically at [carissa.s.wheeler@nasa.gov](mailto:carissa.s.wheeler@nasa.gov) and provide the above-referenced tracking number. Fees for processing your request were less than \$50 and are not being charged in accordance with 14 CFR §1206.503.

Sincerely,



Carissa S. Wheeler  
FOIA Public Liaison Officer

Enclosures

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](mailto: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 .....	7
Introduction .....	8
Approach .....	8
Preparing the Data .....	8
Classification Details .....	9
Creating the Baseline .....	9
Creating the Failure Taxonomy .....	9
Analysis Results .....	10
Failure Classes .....	10
Algorithm .....	10
Bus Interface .....	11
Configuration Management (CM) .....	11
Compiler Error .....	12
Data Definition .....	12
Data Handling .....	12
Documentation .....	13
Hardware .....	13
Input-Output (I/O) System .....	13
Implementation .....	14
Inter-process Communication .....	14
Performance .....	14
Self-Test .....	15
System Integration .....	15
Tools .....	16
User/Pilot .....	16
Error Analysis .....	17
Background .....	17



The Risk Priority Number .....	17
Detailed Class Analysis.....	19
RPN Component Analysis .....	20
Bus Interface Error Class Profile .....	23
Configuration Management Error Class Profile .....	24
Data Definition Error Class Profile .....	24
Data Handling Error Class Profile .....	25
Inter-Process Communication Error Class Profile .....	25
Input/Output System Error Class Profile .....	26
Self-Test Error Profile.....	26
System Integration Error Class Profile .....	27
Root Failure Cause and Effect Relationship Analysis .....	27
Background .....	27
Ground Rules .....	28
Overview of Root Failure Cause and Effect Relationship Chart.....	28
Documentation and External Problems Category .....	29
Requirements Category .....	30
Configuration Management Category .....	31
Algorithm Category.....	31
System Integration / Communication Category .....	34
Self-Test Category.....	36
Application of Data Analysis Results to Evaluating Future Technologies .....	36
References .....	39

## 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 from 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<sup>1</sup>.

## 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<sup>2</sup>. We recognize 4 distinct I/O System error types.

<sup>1</sup> See Error Analysis – Rankings by Occurrence.

<sup>2</sup> 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<sup>3</sup>, 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<sup>4</sup>, 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:

$O := \text{Relative Frequency of Occurance}$

$S := \text{Severity of Error}$

$D := \text{Phase}_{\text{Detected}} - \text{Phase}_{\text{Injected}}$

### 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.

<sup>3</sup> i.e. quantitative or probabilistic risk assessment

<sup>4</sup> See [http://en.wikipedia.org/wiki/Failure\\_mode\\_and\\_effects\\_analysis](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 Introduction Phase	Defect Detection 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.

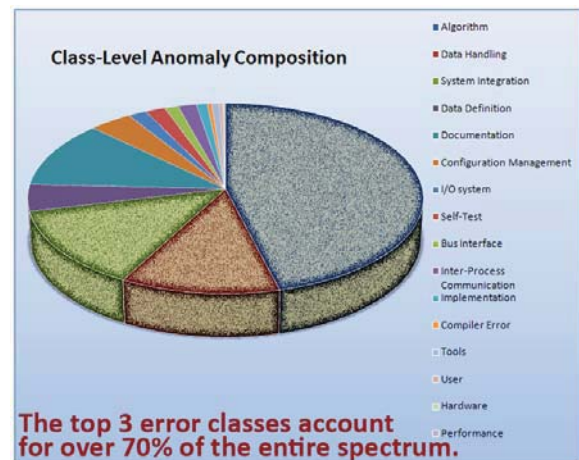


Figure 1 - Class-Level Analysis

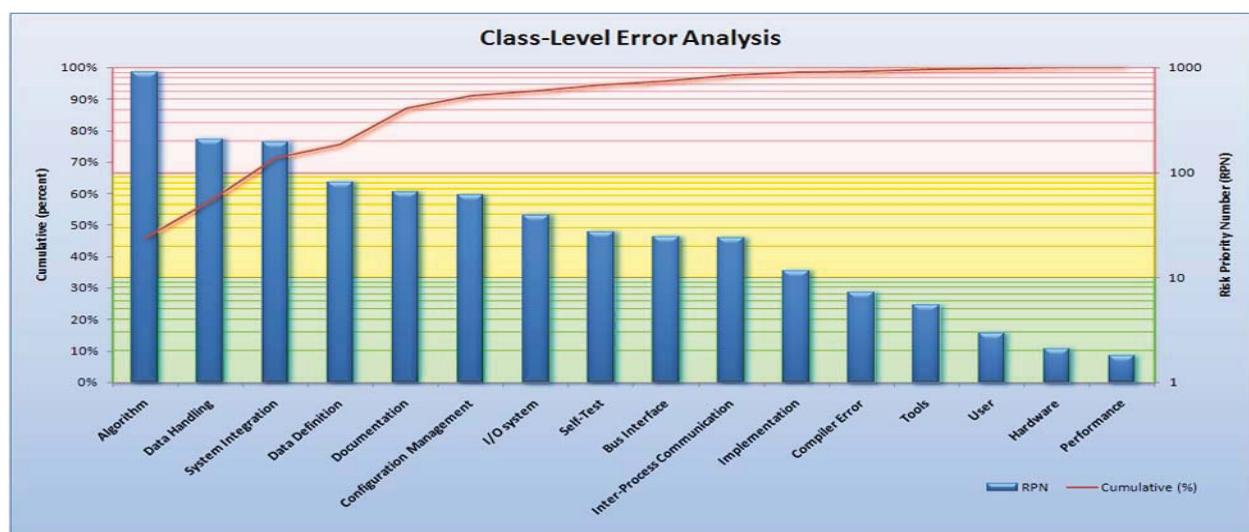


Figure 2 – Class-Level Error Profile

## 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

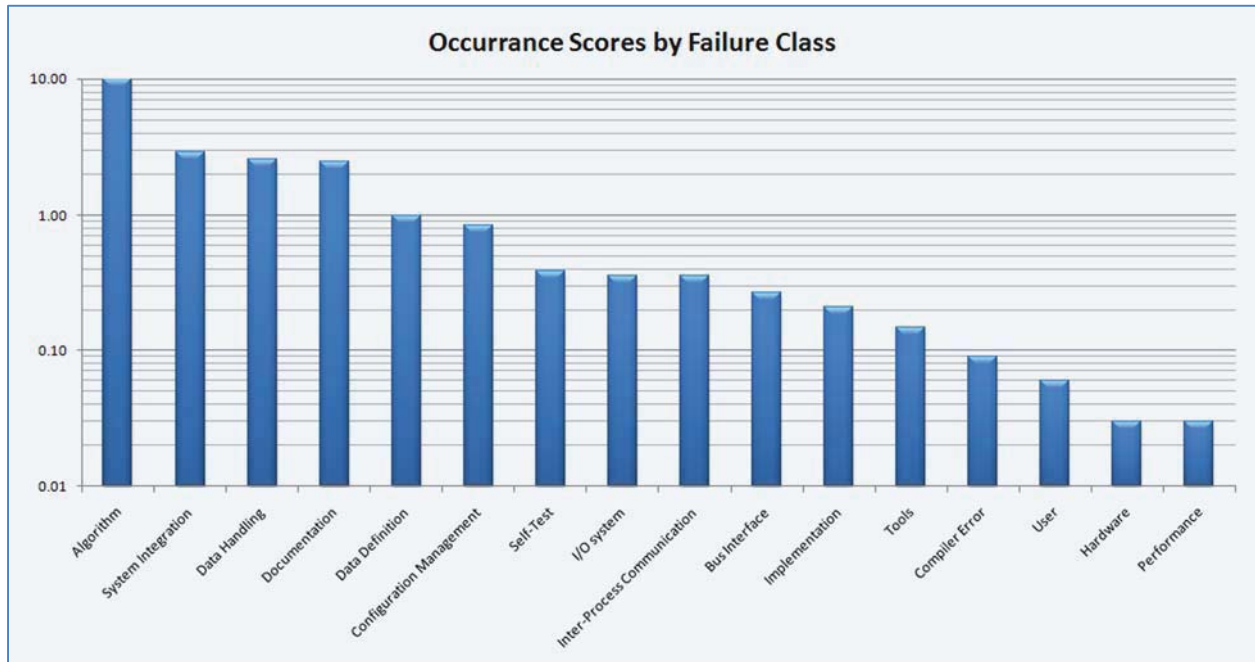


Figure 3 – Occurrence Dimension

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

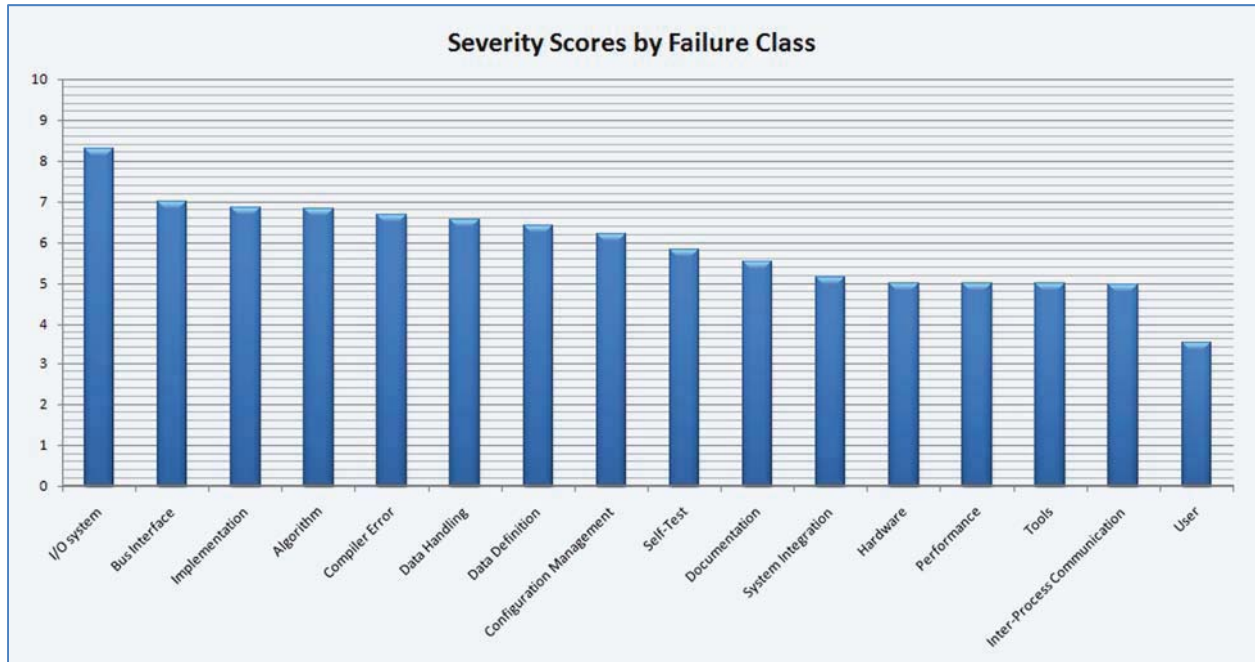


Figure 4 – Severity Dimension

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

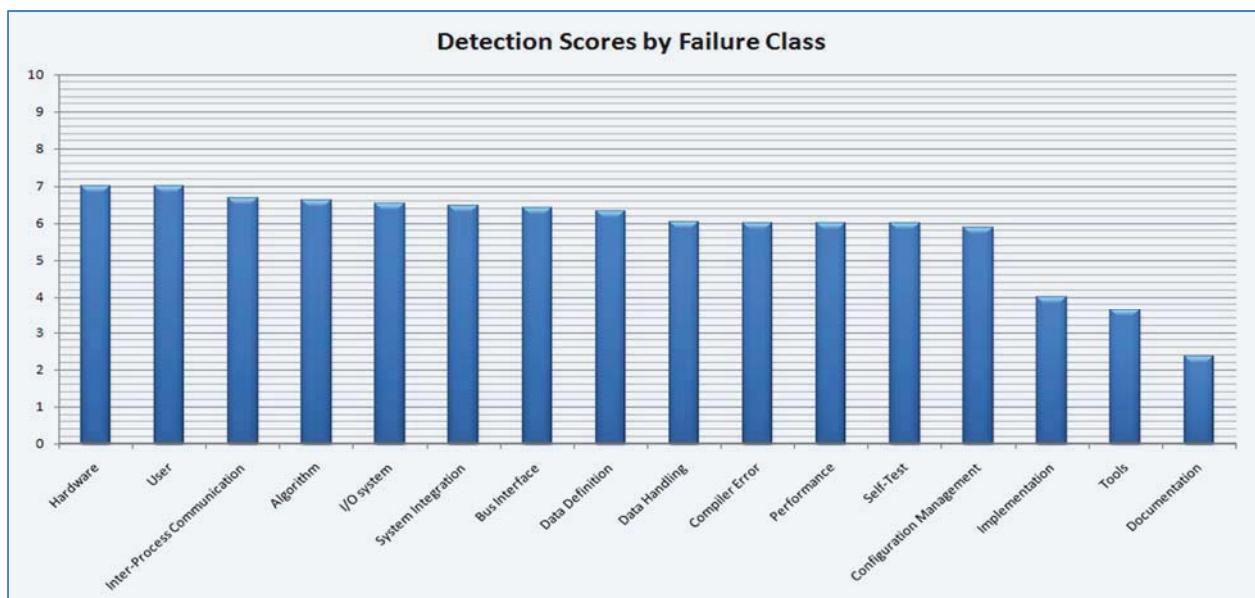


Figure 5 – Detection Dimension



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

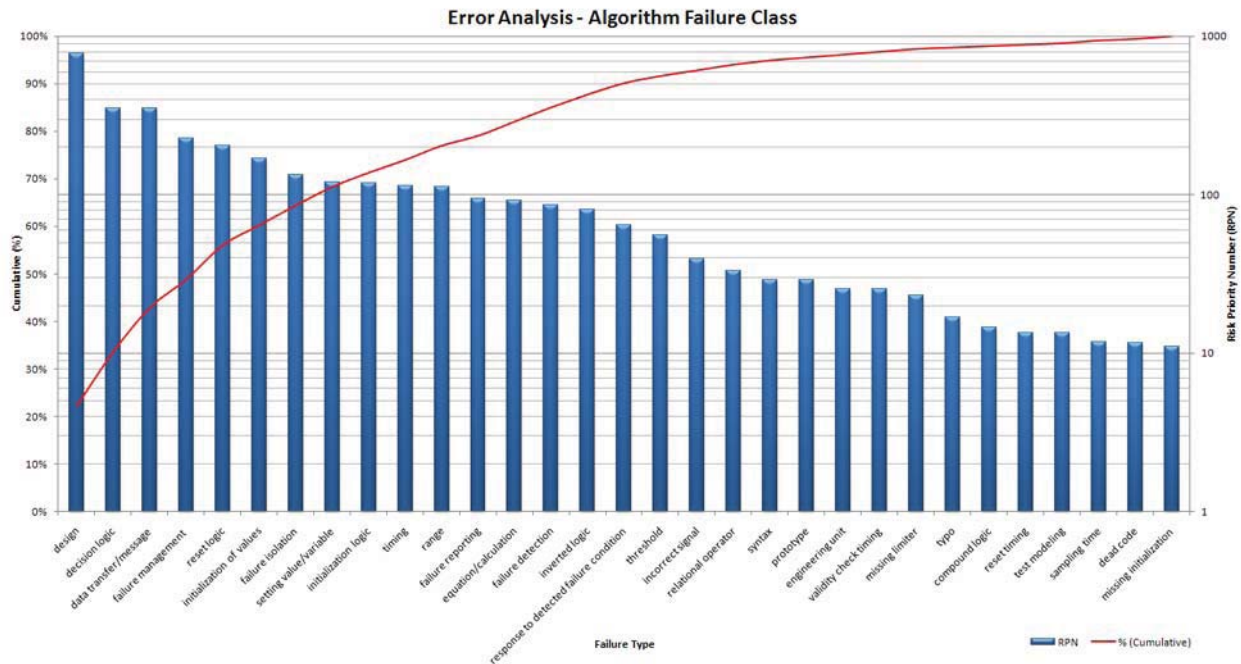


Figure 6 - Algorithm Error Profile

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.

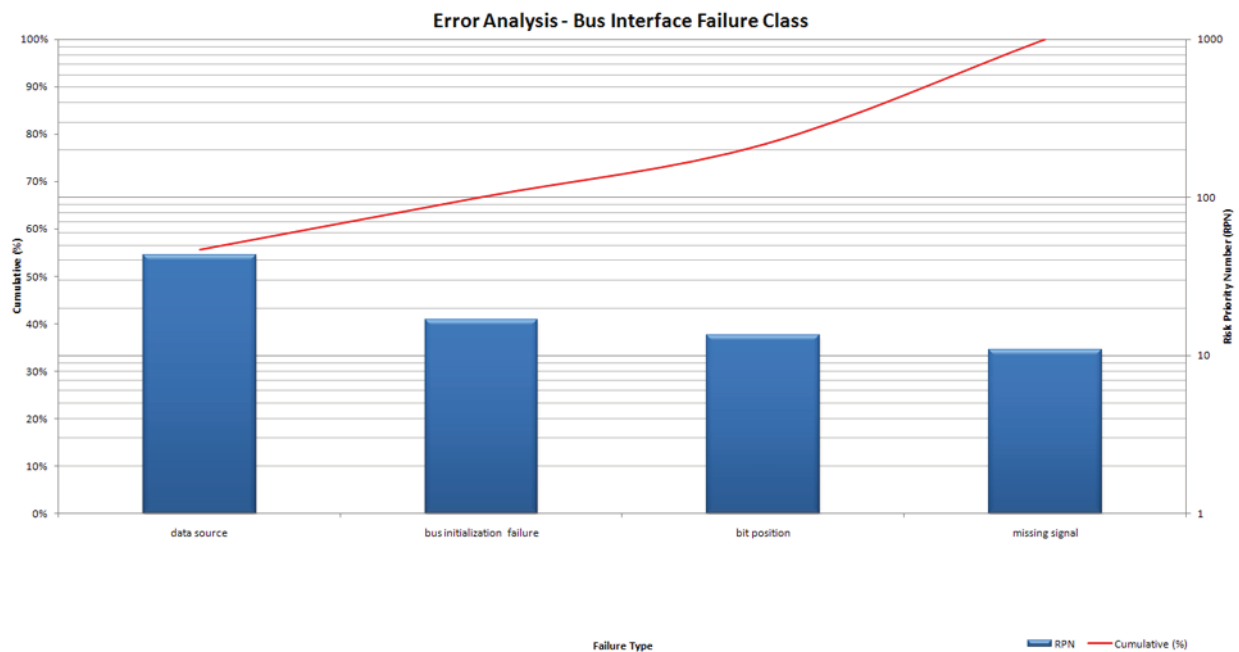


Figure 7 – Bus Interface Error Profile

## 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.

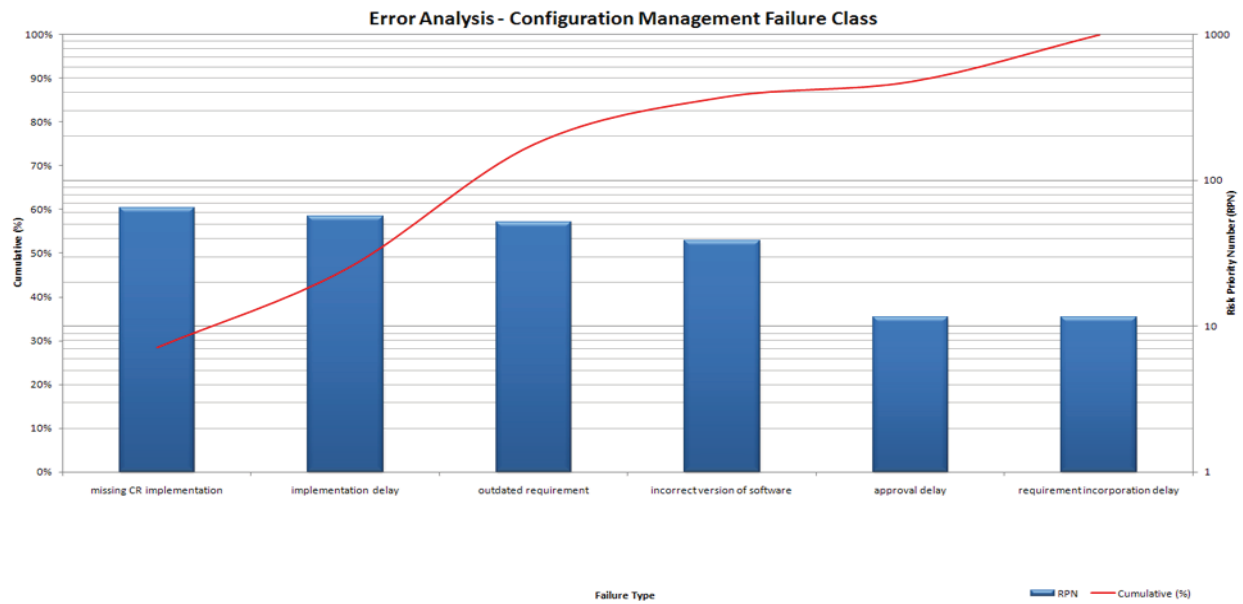


Figure 8 – Configuration Management Error Profile

## 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.

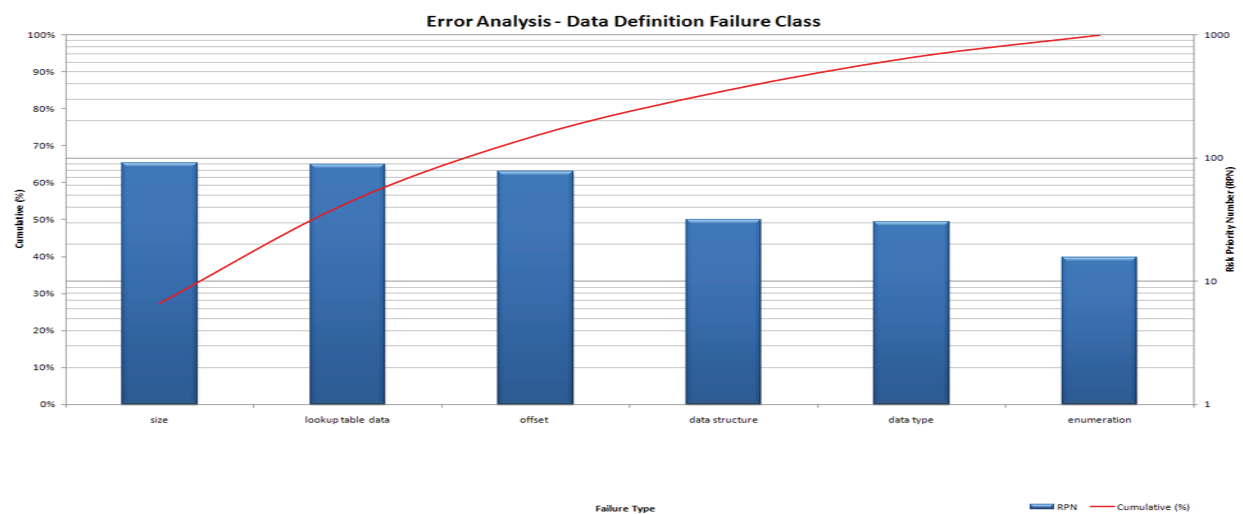


Figure 9– Data Definition Error Profile

## 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.

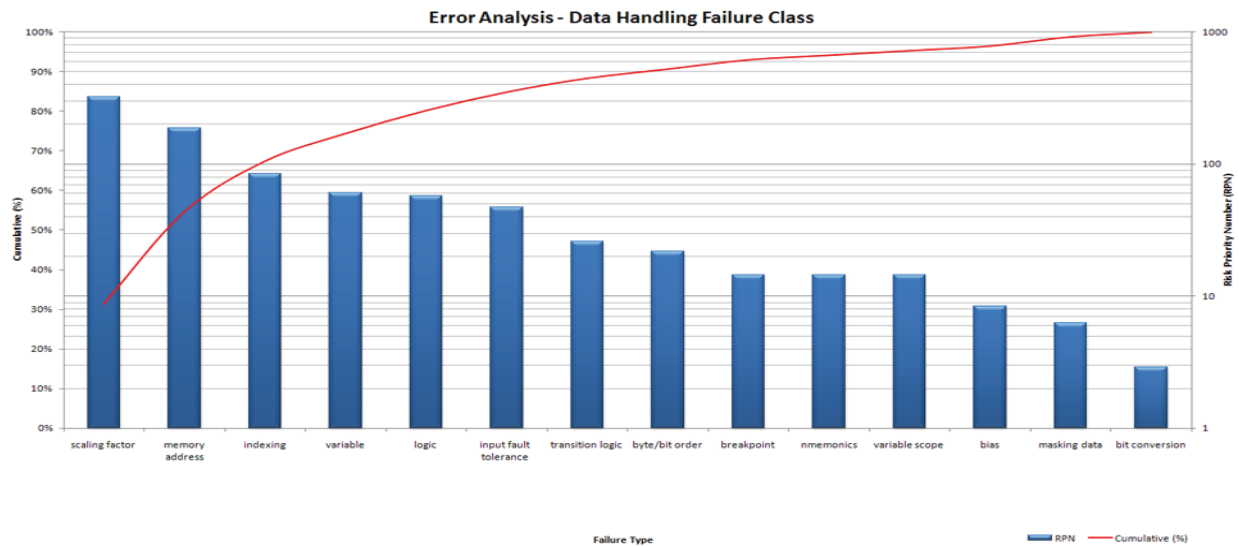


Figure 10 – Data Handling Error Profile

## 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.

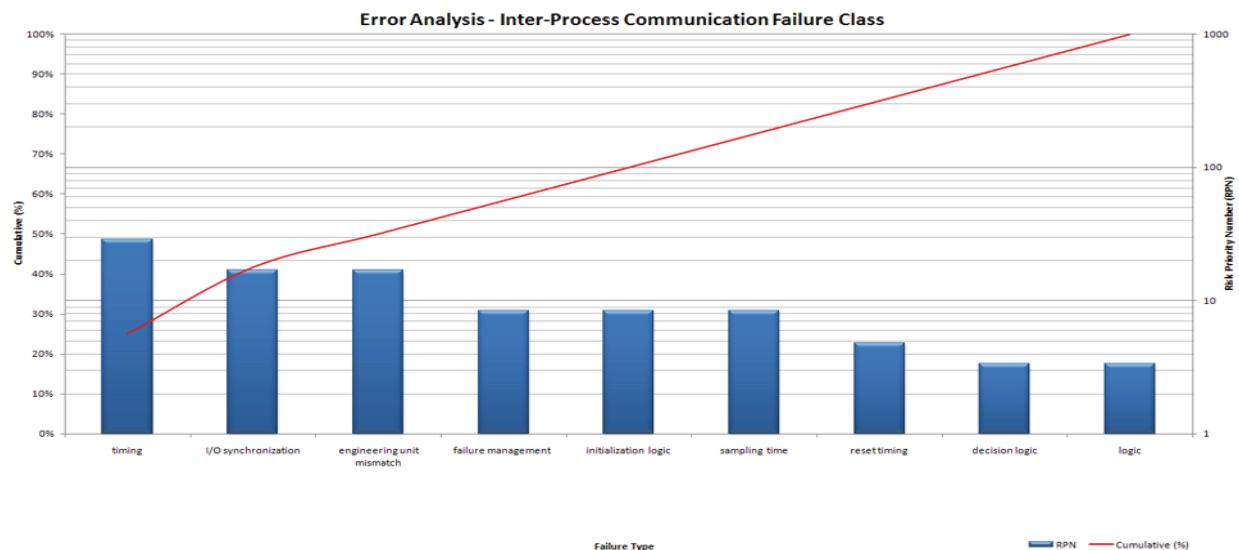


Figure 11 – IPC Error Profile

## 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.

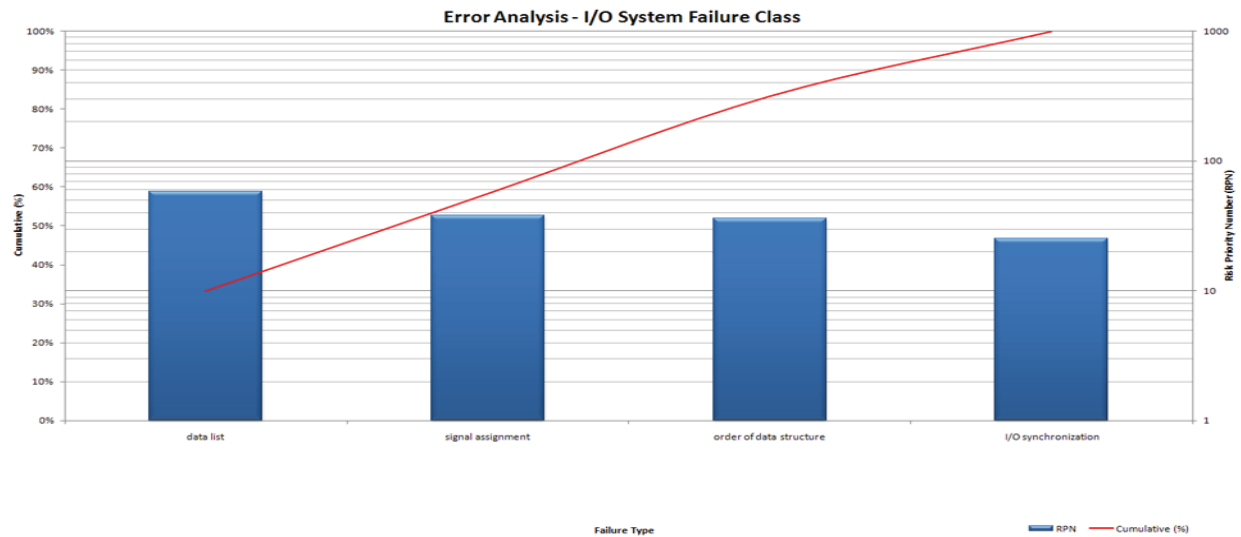


Figure 12 – I/O System Error Profile

## SELF-TEST ERROR PROFILE

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

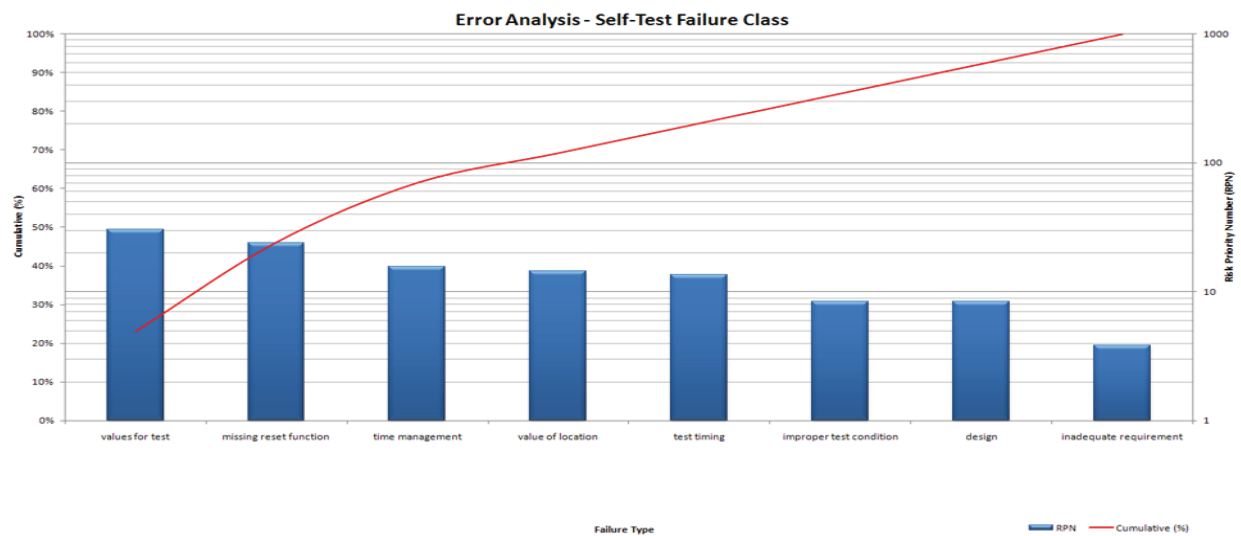


Figure 13 – Self-Test Error Profile

## 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.

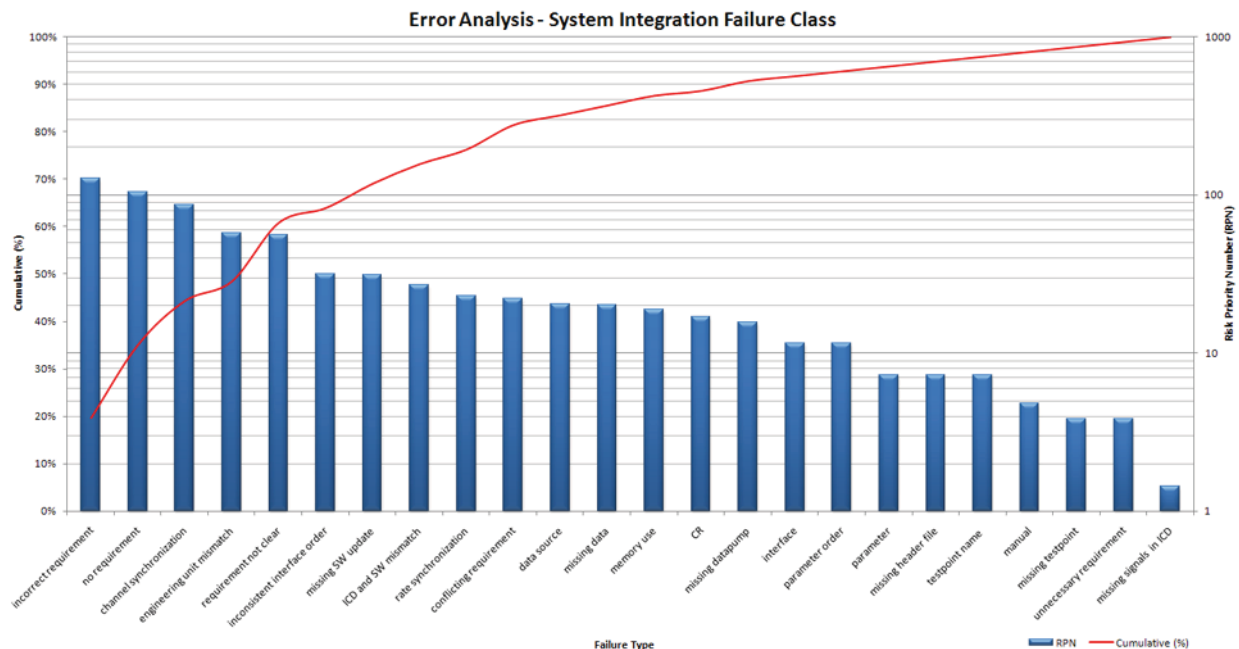


Figure 14 – System Integration Error Profile

## 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.

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

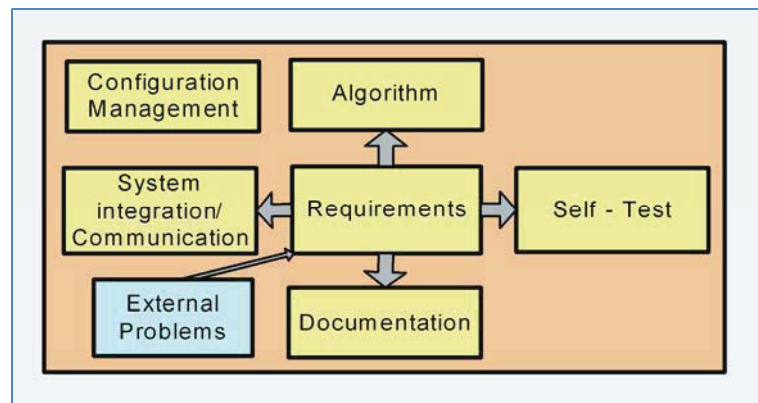


Figure 15 – Related Root Failure Categories

## 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.

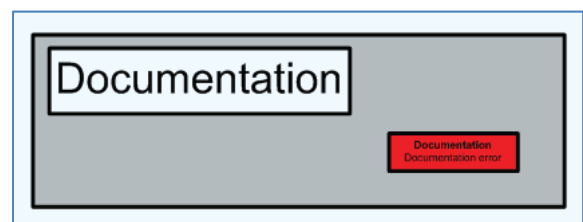


Figure 16 – Documentation Category



Figure 17 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

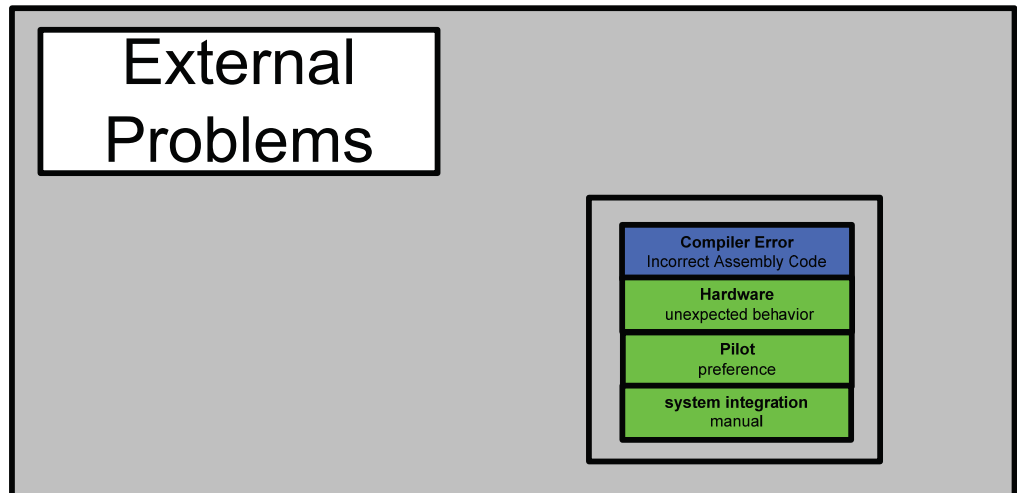


Figure 17 – External Problems Category

application code. They are primarily 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 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

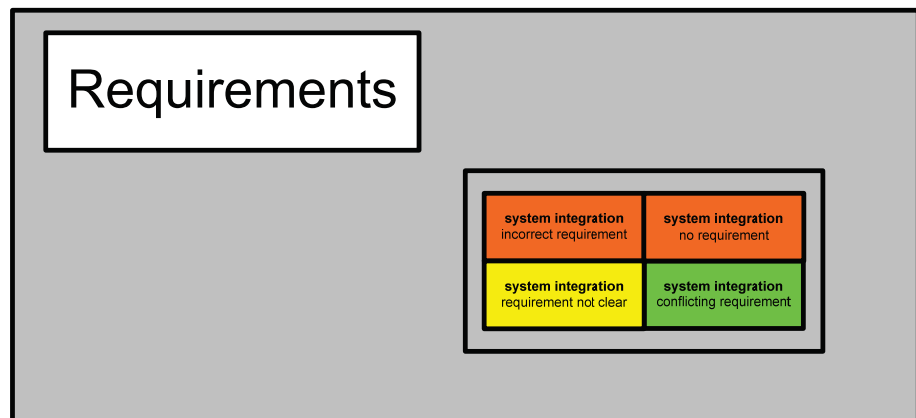


Figure 18 – Requirements Category

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

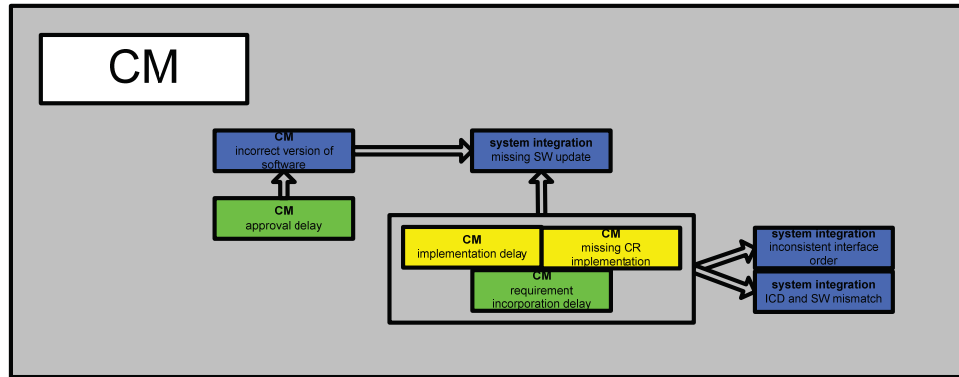


Figure 19 – Configuration Management Category

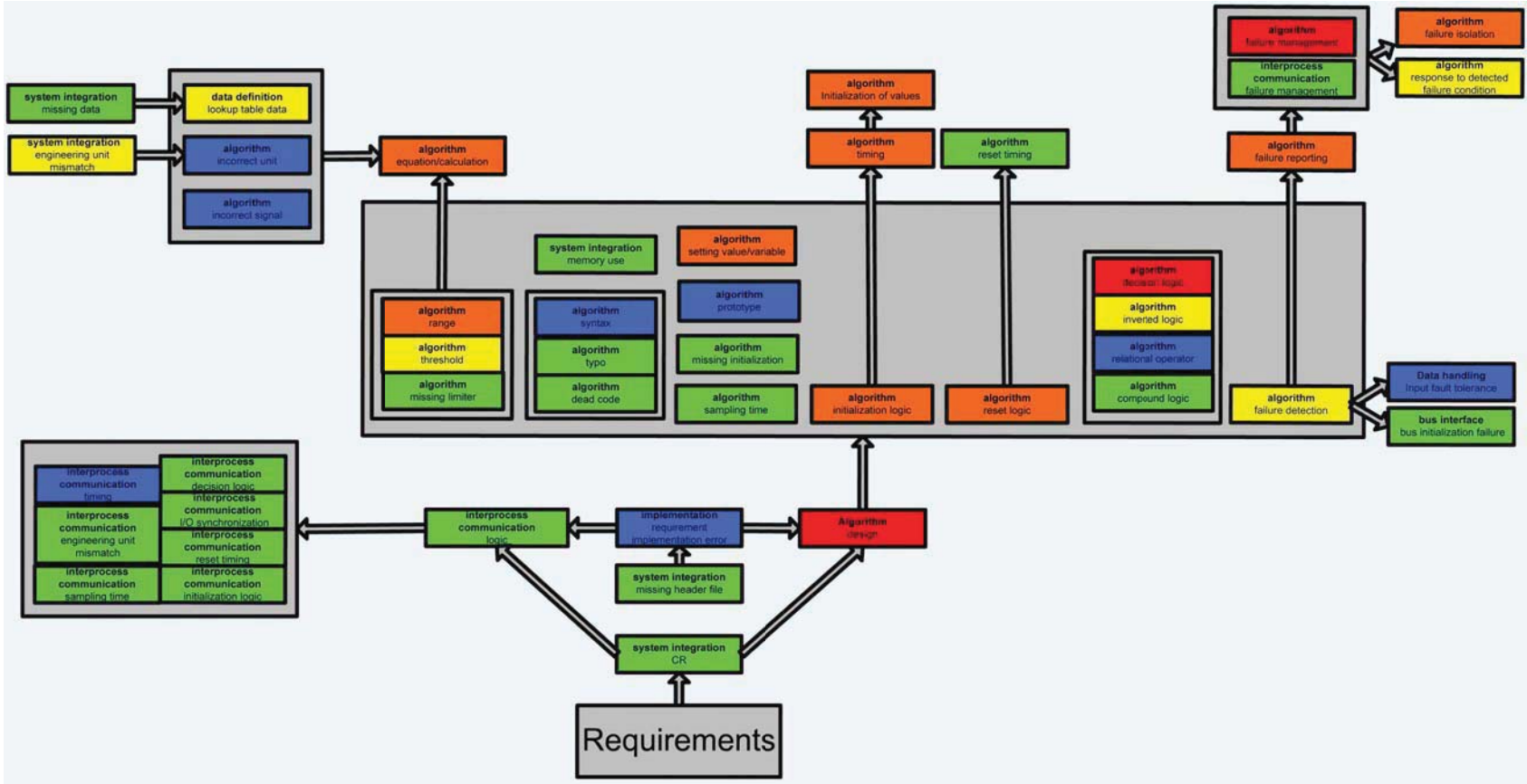
management” failure types. The two yellow blocks, “CM: implementation delay” and “CM: missing CR implementation” are grouped together with the green “CM: requirement incorporation delay” to collect the problems with delays in already approved changes. This collection relates to several “system integration” 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.

## 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 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.

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.

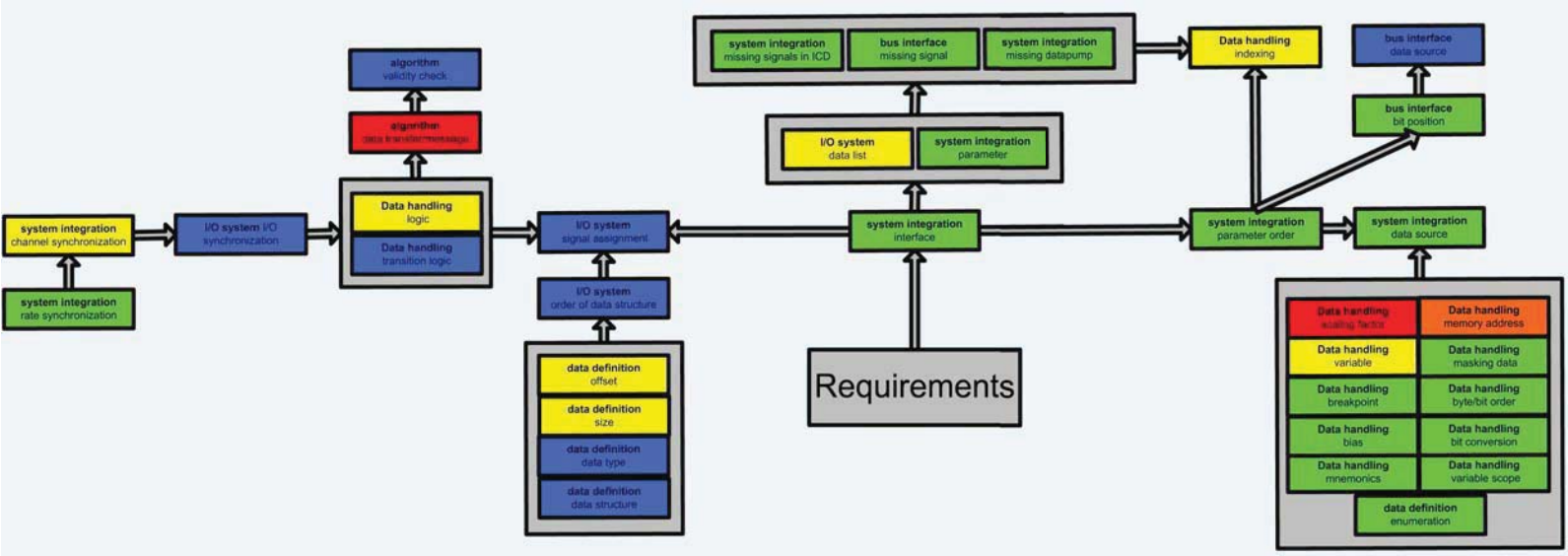


Figure 21 – System Integration / Communication Category

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.

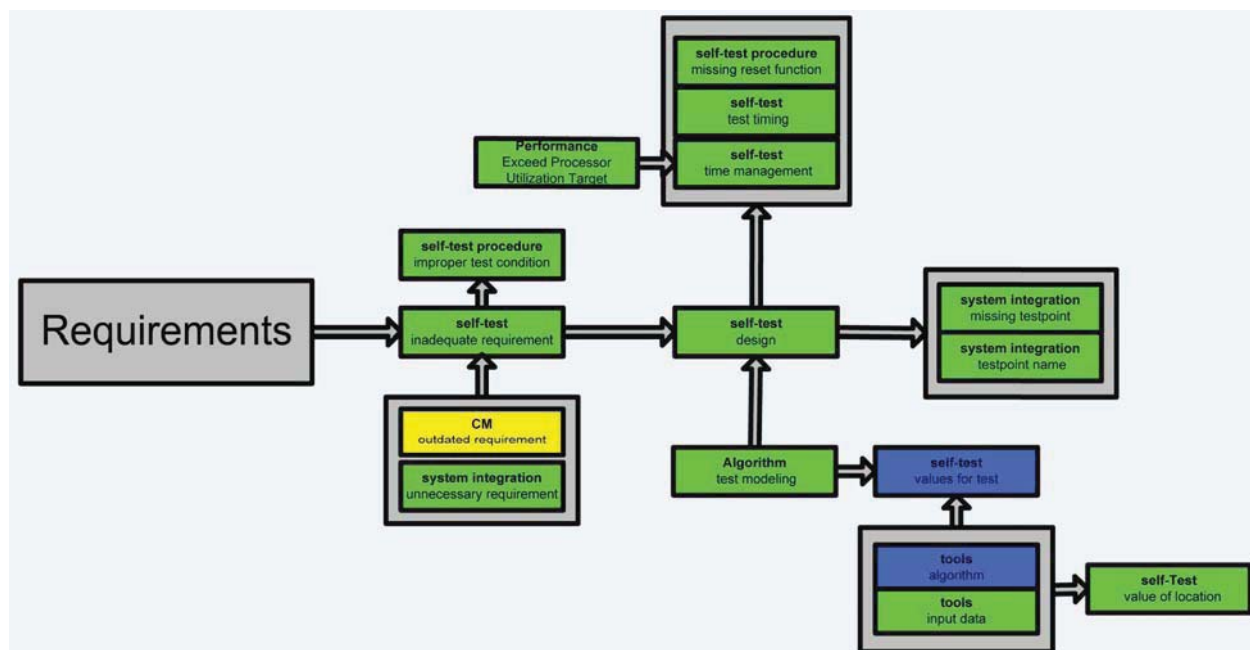


Figure 22 – Self-Test Category

## 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.

- Select technologies/ methods that you want to examine.
- 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 = 2<sup>nd</sup> dominant factor, and Green = the least dominant factor)
- 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)

RPN	Root Failure	Occurrence					Severity					Detection				
		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	
<p>The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.</p> <p><b>PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.</b></p>						
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)	
U	U	U	UU	40	19b. TELEPHONE NUMBER (Include area code) (443) 757-5802	

NASA/CR-2009-000000



# Software Anomaly Taxonomy Validation (SWAT-V)

*Greg Cotton*

*Lockheed Martin Aeronautics Company, Fort Worth, TX 76101*

July 2010

## 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](mailto: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-2009-000000



# Software Anomaly Taxonomy Validation (SWAT-V)

*Greg Cotton*

*Lockheed Martin Aeronautics Company, Fort Worth, TX 76101*

National Aeronautics and  
Space Administration

Langley Research Center Prepared for Langley Research Center  
Hampton, Virginia 23681-2199 under Contract NNL06AA08B

July 2010

## Table of Contents

Table of Contents .....	2
Nomenclature .....	3
Abstract .....	4
Foreword .....	4
1.0 Introduction .....	5
2.0 Background .....	6
3.0 Methods, Assumptions, and Procedures .....	8
3.1 Mariana Auto-Classification Tool .....	8
MarianaPrep .....	8
MarianaExec .....	9
MarianaPredict .....	9
3.2 Configure Mariana On Computer Asset .....	9
3.3 Prototype Classification Tool using Small Data Set .....	10
4.0 Results and Discussions .....	15
4.1 Apply Tool to Original Data Set .....	15
4.2 Validate Terminology and Results .....	22
4.3 Expand Data Set to Other Databases .....	23
5.0 Conclusions .....	27
6.0 References .....	28
7.0 Appendix A .....	28
Appendix A Contents .....	A-3

## **Nomenclature**

CD-SHM: Concept Development for Software Health Management

CSV: Comma Separated Value

FCSR: Flight Critical Systems Research

FMEA: Failure Modes and Effects Analysis

IVHM: Integrated Vehicle Health Management

SPAR: Software Product Anomaly Report

SWAT-V: Software Anomaly Taxonomy Validation

SHM: Software Health Management



## **Abstract**

Under a previous NASA-sponsored effort, Lockheed Martin created taxonomy for software anomalies that quantified the scope, magnitude, and types of fundamental software errors. This taxonomy was created through a manual process whereby several subject matter experts read through and classified a relatively small number of individual software anomaly reports into fundamental error types. Typical software anomaly databases, are, however, far too large and subject matter experts are too valuable to inspect every record manually to develop a failure taxonomy. In an effort to streamline this process, Lockheed Martin investigated the application of auto-classification algorithms for deriving a failure taxonomy for flight critical software anomalies. Because software anomalies are generally documented in free form text, auto-classification tools capable of directly examining this text without significant manual supervision would be very helpful in understanding software health trends. Indeed, auto-classification tools may be the only practical method of “mining” the vast amount of available data for software anomaly trends and failure modes. This report documents the results of Lockheed’s investigation of NASA’s Mariana free form text auto-classification tool for deriving a failure taxonomy from software anomaly reports. The investigation included a partial validation of the previously created taxonomy by using patterns of software failures derived from much larger datasets.

## **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: NNL09AD66T. The work documented herein was performed from September, 2009 through July, 2010.

## 1.0 Introduction

Integrated Vehicle Health Management (IVHM) is one of the four pillars of the NASA Aviation Safety Program. The Integrated Vehicle Health Management project is pursuing foundational research in the development of technologies for automated detection, diagnostics, and mitigation of adverse events due to aircraft software. IVHM includes a Software Health Management (SHM) effort to explore software health in the context of system level dependability cases (a central recommendation of a recent National Academies report).

The Software Anomaly Taxonomy Validation (SWAT-V) program was a NASA-funded effort sponsored by IVHM. This current report documents the work performed by Lockheed Martin Aeronautics (LM Aero) under NASA contract NNL06AA08B, delivery order NNL09AD66T. SWAT-V is a follow-on to a previous Flight Critical Systems Research (FCSR) project “Concept Development for Software Health Management” (CD-SHM), also sponsored by IVHM (task NNL07AB06T). CD-SHM focused on defining a structured approach to software health management through the development of a comprehensive failure taxonomy used to characterize the fundamental failure modes of safety-critical software. For readers who are not familiar with the CD-SHM project, that report is included (as Appendix A) to provide more context for SWAT-V results.

Under this new effort, Lockheed investigated the use of auto-classification algorithms to replicate and validate the software failure taxonomy developed under CD-SHM. The investigation included application of the classification parameters and toolset to large datasets that would otherwise be unsuitable for inspection by subject matter experts. The resulting taxonomy of anomalies will serve as a candidate for software health management frameworks. For this effort Lockheed developed the classification algorithms using NASA’s Mariana free form text tool.

## 2.0 Background

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.

CD-SHM cataloged historical aircraft software anomalies using text-based problem report archives from selected advanced flight-critical software development programs. Anomalies were uncovered during verification and validation activities throughout the software development lifecycle. Subject experts derived a failure taxonomy for aircraft system software by inspecting the data records. CD-SHM quantified the scope, magnitude and types of fundamental software errors that manifest themselves throughout the development of advanced flight-critical software. It employed a two phase approach: 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. CD-SHM mined data from the development of flight-critical software systems for 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. The resulting database served as the source for error classification and comprehensive taxonomy development. The analysis of the CD-SHM database was performed manually; enlisting several subject matter experts to read through and classify 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.

The raw data sources for the CD-SHM common database 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. Analysis required many meetings with program experts 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. It took considerable effort to identify the anomaly found, the phase in which it was introduced and its severity. This information is the foundation of the CD-SHM study and the basis for recommendations.

The process used in CD-SHM is not practical for analyzing typical databases, as the databases of software anomalies are far too large and subject matter experts are too valuable to inspect every record manually. Auto-classification tools may be the only practical method of “mining” the vast amount of available data for software anomaly trends and failure modes. This new tasking applied the Mariana auto-classification tool which was originally developed to assist NASA in categorizing aircraft accident reports. Under this new tasking, Lockheed sought to apply Mariana to the problem of deriving a software anomaly taxonomy from problem reports for purposes of demonstrating that free-form problem reports

could be efficiently classified and also to validate the taxonomy developed under the previous CD-SHM effort.

## 3.0 Methods, Assumptions, and Procedures

### 3.1 Mariana Auto-Classification Tool

NASA Ames provided the MARIANA auto-classification software toolset. MARIANA was developed to help categorize aircraft accident reports, which are also more or less free form text records. MARIANA is available from NASA's DASHLINK web site at the following address:

<https://c3.ndc.nasa.gov/dl/algorithm/mariana/>

Mariana\_v0.8c\_par.zip updated 6/22/2010 is the file archive used for SWAT-V.

MARIANA works by inspecting a text file and a category file and automatically developing the algorithms which map between them. The development of the algorithms is referred to as "training". The algorithms form the basis of models which MARIANA can use to automatically categorize new records. MARIANA consists of three specific tools: MarianaPrep, MarianaExec, and MarianaPredict which are described in turn.

#### *MarianaPrep*

MarianaPrep allows three input files: a text file, a category file in Comma Separated Value (CSV) format, and a thesaurus file (optional). The text file is just a string of words separated by spaces with individual records separated by line breaks. The category file is a matrix with a column of fields for each possible category and a row of fields for each record. The value in the field is "1" if the record is in a category, "-1" if the record is not in the category, and "0" if it is not known if the record is in a category. The number of records (lines) must be the same for both the text and category files.

	A	B	C	D	E	F	G	H	I
1	1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	-1	-1	-1	-1	-1	-1	-1	-1
3	1	-1	-1	-1	-1	-1	-1	-1	-1
4	1	-1	-1	-1	-1	-1	-1	-1	-1
5	1	-1	-1	-1	-1	-1	-1	-1	-1
6	1	-1	-1	-1	-1	-1	-1	-1	-1

Record #1

Member of Category 1

Not Member of Category 5

Example CSV Category File (Partial)

The optional thesaurus is a simple text file which the user may edit. It allows specifying synonyms. While MARIANA is smart enough to know that plurals are equivalent (i.e. “user” has the same meaning as “users”), it does not know that “busses” is plural for “buss”. By default, it also considers “invent” as distinct from “invented”. Another feature in thesaurus called “Stop words” is a list of words to be ignored (such as “a” or “the”). Thesaurus allows the user a means to modify MARIANA’s behavior for a particular context.

MarianaPrep combines the input files to compute statistics. MarianaPrep counts the occurrences of each unique word in each record for each category. It also considers words which are not found in a category. It produces a separate Zip archive file for each category containing the computed statistics.

### ***MarianaExec***

MarianaExec is the tool which “trains” the models. It examines each Zip archive created by MarianaPrep in turn and creates a mathematical model for each category. It does this by randomly dividing the records into two groups; the group used to create a hypothetical model and the group used to test this model. It carries out this process 5 times (each time using a different random group of records) and chooses the model which performs the best. It even provides statistics on how well it performed (between 0.5 = 50% ; a coin toss and 1.0 = 100%; perfect).

### ***MarianaPredict***

MarianaPredict allows automatic classification of new data against the models created for each category. A text file (similar to the one used for MarianaPrep) contains a string of words separated by spaces with individual records separated by line breaks. This can be evaluated against each of the model files produced by MarianaExec. If the record is determined to be in the category of that model, MarianaPredict returns a positive number. If the record is not in the category, it returns a negative number.

## **3.2 Configure Mariana On Computer Asset**

MARIANA is written mainly in Java in a LINUX environment (there are a few system-specific modules written in C). For this effort the software was ported to a Windows® Vista® environment comprised of an HP® desktop computer with an Intel Core™ 2 Duo CPU running at 2.83 GHz with 3.48 GB of RAM..

Only a few lines of C-code required changes to compile in Windows®. They mostly referred to the system time used to generate random numbers. In “helperRoutines.cpp” “getpid” was revised to “\_getpid” and the following was inserted at about line 104:

```
void gettimeofday(struct timeval* t,void* timezone)
{ struct _timeb timebuffer;
  _ftime( &timebuffer );
```

```
t->tv_sec=timebuffer.time;
t->tv_usec=1000*timebuffer.millitm;
}
```

In “helperRoutines.h” some additional include statements were needed in place of sys/time:

```
//#include <sys/time.h>
#include <winsock.h>
#include <sys/timeb.h>
#include <sys/types.h>
#include <process.h>
```

After compiling, the new library must be renamed “MarianaFunctions.dll” and moved to the “bin” directory.

### 3.3 Prototype Classification Tool using Small Data Set

The 8 category Software Product Anomaly Report (SPAR) data developed to de-bug MARIANA was used to experiment with various “tuning” techniques. To improve the confidence of the classifications, the thesaurus was expanded from 2 lines to 575 lines.

A PERL script was developed to pre-process the SPAR data. This provided an expedited method of changing the data from standard spreadsheet format into the CSV and text files required by MarianaPrep. Three fields were combined to produce the richest possible text file describing the software anomalies. The script also scrubbed the data by deleting proper names, pronouns, etc. which are not relevant to the cause or effect of software anomalies. Since some SPAR authors used negative descriptions, the script file deleted the space following the word “not” so the phrase “not a timing problem” becomes “nottiming problem”.

The actual pre-processing PERL script file is included below:

```

$file = 'spar_data.csv';

@delete_words = split(" ", <<END);

a about again also although an and any anything anytime anyway apparent apparently appear appears are around as
assume assumed assumes assumption at be became because become becomes becoming been being believe believed
beleiving besides bring bringing but by came can cannot can't cant cause causes causing come comes coming
comma could do does doing done during follow followed following follows for from furthermore get gets getting go
going had happened has have having him however if in into is it its just kind leave leaves leaving like likely look
looked looking looks make maked makes making may mr much must nonetheless note of only other others
otherwise or our overall per please possible possibly possibility pretty quick quicker quickly quite required saw says
see seeing seem seemed seems seen sees shall shortly should significant significantly similarly simply since slight
slightly so some somehow something sometime sometimes somewhat soon sooner specific specifically still such
suggest suggestion suggestions supposed than that their them the then there thereby therefore these they this though
thus to upon use uses was we went were what whatever when whenever where whereas whether which while why
will with would you your =

richard larry dave bruce ted

END

$delete_words = join('|', @delete_words);

open( CSV, $file);
binmode CSV;
while( (read CSV, $more_text, 10000) ){ $text .= $more_text};
close CSV;
$text =~ s/"/ /sg;
while( $text =~ /(("[^"]*)/s ){
    $before=$`;
    $field=$1;
    $after=$';
    #print "field=$field\n";
    $field =~ s/,/ /g;
    $field =~ s/\x0d/ /g;
    $field =~ s/\n/ /g;
    $field =~ s/"//g;
    $text = $before . $field . $after;
    #print "field=$field\n";
    #exit 0;
}
#$text_new .= $after;
#$text = $text_new;

@spars=split(/\x0d/, $text);
$num=$#spars-1; #not counting header line
print "Spars found=$num\n";
foreach $spar (@spars){
    $spar =~ s/\x0d//g;
    $spar =~ s/\n//g;

```



```

$text2 .= "$spar\n";
}
$text = lc $text2;
# GET HEADER LINE
$text =~ s/(.*)\n//; # get all of first line (and remove it)
$header = uc $1;
$header =~ s/(.*)/$1/;
#print "header=$header\n";
# GET COLUMN LABELS
@labels = split(/,/,$header);
$num_cols=$#labels +1;
#print "num_cols=$num_cols\n";
$col=1;
foreach $label (@labels){
# print "$col => $label\n";
$col_index{$label} = $col-1;
$col++;
}

# SCAN DATA FOR CAUSE_CLASSIFICATION CATEGORIES
$cat_col=0;
while( $text =~ /(.)+/g ){
$spar=$1;
#print "spar=$spar\n";
@columns = split(/,/,$spar);
$cause= lc $columns[$col_index{CAUSE_CLASSIFICATION}];
if( !$found{$cause} ){
$found{$cause} = 'y';
$cat_col{$cause} = $cat_col;
$cat_col++;
push(@cat_columns, $cause);
}
}
$num_cats=$cat_col;
$cat_header = join(',', @cat_columns);
# READ EACH SPAR LINE
$line_num=0;
open(OUT_ALL, ">all_categories.csv");
print OUT_ALL "LINE_NUMBER,SPAR_ID,CAUSE_CLASSIFICATION,$cat_header,ROOT_CAUSE\n";
open(OUT_TRAIN_CATS, ">categories.csv");
open(OUT_TRAIN_SPARS, ">root_cause.txt");
while( $text =~ /(.)+/g ){
$spar=$1;
#print "spar=$spar\n";
$line_num++;
@columns = split(/,/,$spar);
$cause= lc $columns[$col_index{CAUSE_CLASSIFICATION}];
$anom=$columns[$col_index{ROOT_CAUSE}];
$combined_cause = $columns[$col_index{ANOMALY_BEHAVIOR}] . ' ' . $anom . ' ' .
$columns[$col_index{TITLE}];

```

```

# remove underscore and non ascii characters
$combined_cause =~ s/_/ /g;
$combined_cause =~ s/[\x0a\x20-\x7f]/ /g;
# remove non-informative words
$combined_cause =~ s/\b($delete_words)\b/ /gi;
# remove spaces after the word "not"
$combined_cause =~ s/\bnot\s+/not/g;
#$combined_cause =~ s/\d+/ /g;
$combined_cause =~ s/+/ /g;
$spar_num=$columns[$col_index{CH_DOC_ID}];
#print "cause=$cause cat_col=$cat_col{$cause}\n";
foreach $cat_num (0..$num_cats-1){
  if( $cat_num eq $cat_col{$cause} ){
    $cat_flag[$cat_num] = 1;
  }else{
    $cat_flag[$cat_num] = -1;
  }
}
$cats= join(', ', @cat_flag);
# print "Spar num=$num cause=$cause anom=$combined_cause\n";
print OUT_TRAIN_CATS "$cats\n";
print OUT_TRAIN_SPARS "$combined_cause\n";
print OUT_ALL "$line_num,$spar_num,$cause, $cats, $combined_cause\n";
}
close OUT_ALL;
close OUT_TRAIN_CATS;
close OUT_TRAIN_SPARS;
exit 0;

```

#### Pre-processing PERL Script

These tuning efforts appeared very successful in providing MARIANA the best possible chance of making correct classifications. The classifications reported better than 90% confidence on average when run through MarianaExec.

To demonstrate how this PERL script massages data, the following example shows SPAR data before and after it is pre-processed for MARIANA (note the fields in the spread sheet contain more data than can be displayed in the fixed cell size shown):

Three Fields Combined To Produce Text			
CAUSE_CLASSIFICATION	ANOMALY_BEHAVIOR	ROOT_CAUSE	TITLE
Record #1 → algorithm	Unexpected EGI Failures	The EGI monitors v	Unexpected EGI Failure (MFL FCS056) During Aircraft C
algorithm	In mech, Action 2c for TAT S	Mechanization erro	TAT Selection – Action 2c
algorithm	The FCS MUX Input functio	Lack of detail and c	Executive Call for FCS MUX Input Function
algorithm	The TAT Range Check moni	Mechanization erro	TAT Range Check monitor PC logic
algorithm	The IMFP monitor uses a sir	Mechanization erro	IMFP monitor PC logic
algorithm	NVM data is dumped 34 fra	Cold start initializat	NVM dump on powerup
algorithm	Large transients on the actu	After a BIT Exit, co	Actuator Init. Function Not Providing Smooth Transition f
algorithm	With an out of range conditi	The TAT range che	TAS Out-of-Range Failure Condition Not Resetable to N
algorithm	Testing was performed stan	SRDCR-99 added	SPV Power Failure Reported After Cold Start Power Up
algorithm	The selected AOA signal is li	The AOA_Sel signa	AOA Invalid indication when AOA_Sel reaches 50 deg.
algorithm	While integrating SRDCR-99	The root cause of t	SPV Power Monitor does not Declare Failure if SPV Hi S
algorithm	After PowerUp Initialization,	Mech error	AVS MUX Init - Operational Status Register Should Be l
algorithm	Failed P15VDC or N15VDC	Eventhough the ch	Fail P15VDC or N15VDC internal cause all surfaces faile
algorithm	While performing integration	SRDCR-31 modifie	FCS Fails to Indicate MFL-112 (Probe Temp Low) When
algorithm	Throttle data (PLA) was trac	The use of the pin	AV mux data stale for one FADEC
algorithm	Section 3.3.1.2 of the MLV ii	The MLV SDD use	MLV logic using GCR not compatible with OFP
algorithm	In SPAR-1347, BIT fault tab	The problem starte	BIT Control: Missing Initialization in Mechanization
algorithm	When FCS_BIT_Request is	The mechanization	The FCS_BIT_In_Progress signal is set to "True" when l

Example Of Data Before Pre-Processing

unexpected egi failures (mfl fcs056 pfl fcs egi unav) occurred aircraft ground tests ofp fm02c. part 1 test: unexpected mfl fcs056 occurred steps 5 28 52. unexpected mfl egi035 failure occurred steps 28 52. part 2 egi monitors activated prematurely after powerup before egi powered began communicating flcc. monitors activated mlg tach sensors indicating wheel speed ~23 knots duration 1 second. tach inputs unexpected test procedure states aircraft stationary no time test aircraft moved. further investigation test procedure revealed brake control unit bit initiated pilot mlg tach inputs stimmed ~23 knots maximum 1 second. initiating bcu bit before egi powered nuisance egi failures. unexpected egi failure (mfl fcs056) aircraft ground test (ofp fm02)

Record #1 Text After Pre-Processing

Obviously repeating this procedure for every record would be very labor intensive manual effort. This is the reason we use a software script to automate the task.

## 4.0 Results and Discussions

### 4.1 Apply Tool to Original Data Set

CD-SHM partitioned 726 SPARs into 16 “classes” and 114 “fundamental types”. While one fundamental type (Documentation Error) had 83 members, 37 fundamental types had only a single example. Since it takes several examples to “train” the MARIANA tool, it was not feasible to use the CD-SHM fundamental types as the taxonomy categories. In addition, categories with a single example are not very useful for Software Health Management (no trends). For these reasons, the CD-SHM classification scheme was revisited.

The data was subsequently revised to a relaxed granularity of classification. Fundamental types (for example “Memory Address”) with many members were retained as categories. Other fundamental types were mapped to unique categories such that each category would contain at least 11 members. Remaining unique SPARs (for example the “Hardware” class with a single member) were grouped together as “Other”. After several iterations, a data set of 22 categories seemed to provide the best balance between functionality and usability:

CAT #	CATEGORY	INSTANCES
1	Algorithm	78
2	Configuration Management	40
3	Data Definition	33
4	Data Handling	45
5	Documentation	83
6	Equation / Calculation	19
7	Failure Detection	11
8	Failure Isolation	14
9	Failure Management	36
10	Incorrect Signal / Data	47
11	Initialization	42
12	Logic	54
13	Memory Address	20
14	No Requirement	15
15	Other	17
16	Range	11
17	Requirements	45
18	Reset Logic	24
19	Scaling Factor	27
20	Self-Test	11
21	Synchronization / Timing	37
22	System Integration	17
TOTAL		726

SPAR Categories

The CD-SHM fundamental types are mapped to SWAT-V categories as follows:

CAT #	CATEGORY	CD-SHM FUNDAMENTAL TYPES	INSTANCES	TOTAL
1	Algorithm	Algorithm - Design	74	78
		Algorithm - Syntax	3	
		Algorithm - Test Modeling	1	
2	Configuration Management	Configuration Management (all)	28	40
		System Integration - ICD And SW Mismatch	4	
		System Integration - Change Release	1	
		System Integration - Interface	1	
		System Integration - Manual	1	
		System Integration - Missing SW Update	5	
3	Data Definition	Data Definition (all)	33	33
4	Data Handling	Data Handling - Bias	1	45
		Data Handling - Bit Conversion	1	
		Data Handling - Breakpoint	2	
		Data Handling - Byte / Bit Order	2	
		Data Handling - Indexing	11	
		Data Handling - Logic	5	
		Data Handling - Masking Data	2	
		Data Handling - Input Fault Tolerance	4	
		Data Handling - Transition Logic	3	
		Data Handling - Variable	6	
		Data Handling - Variable Scope	1	
		I / O System - Data List	4	
		I / O System - Order Of Data Structure	3	
5	Documentation	Documentation (all)	83	83
6	Equation / Calculation	Algorithm - Equation / Calculation	10	19
		Algorithm - Engineering Unit	2	
		Algorithm - Missing Limiter	2	
		Interprocess Communication - Engineering Unit Mismatch	1	
		System Integration - Engineering Unit Mismatch	4	
7	Failure Detection	Algorithm - Failure Detection	10	11
		Algorithm - Threshold	1	
8	Failure Isolation	Algorithm - Failure Isolation	14	14
9	Failure Management	Algorithm - Failure Isolation	20	36
		Algorithm - Failure Reporting	7	
		Algorithm - Response To Detected Failure Condition	8	
		Interprocess Communication - Failure Management	1	

10	Incorrect Signal / Data	Algorithm - Data Transfer / Message	31	47
		Algorithm - Incorrect Signal	4	
		Algorithm - Typo	1	
		Bus Interface - Bit Position	1	
		Bus Interface - Data Source	5	
		Bus Interface - Missing Signal	2	
		I / O System - Signal Assignment	3	
11	Initialization	Algorithm - Initialization Logic	10	42
		Algorithm - Initialization Of Values	14	
		Algorithm - Missing Initialization	2	
		Algorithm - Setting Value / Variable	13	
		Bus Interface - Bus Initialization Failure	1	
		Data Handling - Mnemonics	1	
		Interprocess Communication - Initialization Logic	1	
12	Logic	Algorithm - Decision Logic	38	54
		Algorithm - Inverted Logic	9	
		Algorithm - Compound Logic	1	
		Algorithm - Relational Operator	4	
		Interprocess Communication - Decision Logic	1	
		Interprocess Communication - Logic	1	
13	Memory Address	Data Handling - Memory Address	20	20
14	No Requirement	System Integration - No Requirement	15	15
15	Other	Algorithm - Dead Code	1	17
		Algorithm - Prototype	2	
		Compiler Error - Incorrect Assembly Code	3	
		Hardware - Unexpected Behavior	1	
		Performance - Exceed Processor Utilization Target	1	
		Pilot - Preference	2	
		Self Test - Time Management	2	
		Tools - Input Data	1	
		Tools - Algorithm	4	
16	Range	Algorithm - Range	10	11
		Algorithm - Threshold	1	
17	Requirements	Implementation - Requirement Implementation Error	7	45
		System Integration - Conflicting Requirement	5	
		System Integration - Incorrect Requirement	19	
		System Integration - Requirement Not Clear	12	
		System Integration - Requirement	1	
		Self Test - Inadequate Requirement	1	
18	Reset Logic	Algorithm - Reset Logic	24	24
19	Scaling Factor	Data Handling - Scaling Factor	27	27

20	Self-Test	Algorithm - Threshold	2	11
		Self Test - Design	1	
		Self Test - Values For Test	3	
		Self Test - Value Of Location	1	
		Self Test Procedure - Improper Test Condition	1	
		Self Test Procedure - Missing Reset Function	3	
21	Synchronization / Timing	Algorithm - Timing	9	37
		Algorithm - Reset Timing	1	
		Algorithm - Sampling Time	2	
		Algorithm - Threshold	1	
		Algorithm - Validity Check Timing	2	
		I / O System - I / O Synchronization	2	
		Interprocess Communication - Timing	3	
		Interprocess Communication - I / O Synchronization	2	
		Interprocess Communication - Reset Timing	1	
		Interprocess Communication - Sampling Time	1	
		System Integration - Channel Synchronization	9	
		System Integration - Rate Synchronization	3	
		Self Test - Test Timing	1	
22	System Integration	System Integration - Inconsistent Interface Order	3	17
		System Integration - Data Source	2	
		System Integration - Memory Use	2	
		System Integration - Incorrect Parameter	1	
		System Integration - Missing Data	2	
		System Integration - Missing Datapump	2	
		System Integration - Missing Header File	1	
		System Integration - Missing Signals In ICD	1	
		System Integration - Missing Testpoint	1	
		System Integration - Parameter Order	1	
		System Integration - Testpoint Name	1	
TOTAL			726	726

Mapping of Categories to CD-SHM Fundamental Types

The data was pre-processed with the previously described PERL script. Another, simple PERL script was developed to call MarianaPrep successively for each category:

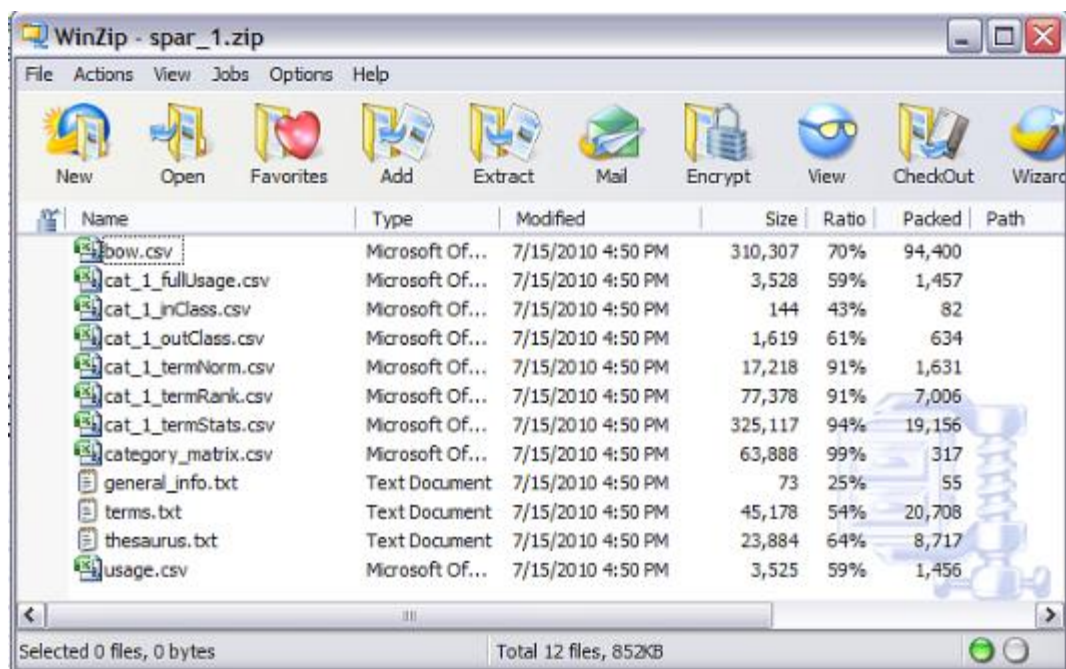
```
$num_runs=22;

print "num_runs=$num_runs\n\n";

foreach $i (1..$num_runs){
  $run_printout = `java -Xmx512M -Djava.library.path=.\\ MarianaPrep -h ..\\data\\thesaurus.test -c
  ..\\data\\categories.csv -o test\\spar_${i}.zip -cl $i -t ..\\data\\root_cause.txt`;
  print "Run $i output:\n$run_printout\n";
}
```

#### MarianaPrep Script

This resulted in the creation of 22 Zip archives each containing 12 files. The archives each look similar to the following:



#### Typical Zip Archive



MarianaExec was called on each of the Zip archives produced by MarianaPrep. The result is a separate model file for each of the 22 categories. A custom PERL script automated the successive calls for each category:

```
$num_runs=22;

print "num_runs=$num_runs\n\n";

foreach $i (1..$num_runs){
    $run_printout = `java -Xmx512M -Djava.library.path=.\\ Mariana -p test\\spar_${i}.zip -c $i -o test\\cat_${i}.model`;
    print "Run $i output:\n$run_printout\n";
}
```

#### MarianaExec Script

When MarianaExec runs, it provides statistics on the model's ability to correctly classify. The confidence statistics are computed for both a default MARIANA algorithm and an optimized algorithm. Since a record is either IN or NOT IN a category, a confidence of 50% represents random chance. The resulting average confidence of 93.6% (92.4% when using optimized algorithms) indicates MARIANA has high confidence in its classifications. The confidence statistics for each category are presented in the following table:

			CONFIDENCE	
CAT #	CATEGORY	INSTANCES	DEFAULT	OPTIMIZED
1	Algorithm	78	0.908425	0.907509
2	Configuration Management	40	0.990956	0.986434
3	Data Definition	33	0.931358	0.915462
4	Data Handling	45	0.822997	0.817829
5	Documentation	83	0.893231	0.913043
6	Equation / Calculation	19	0.974286	0.973333
7	Failure Detection	11	0.994413	0.955307
8	Failure Isolation	14	0.796610	0.806497
9	Failure Management	36	0.877193	0.843275
10	Incorrect Signal / Data	47	0.910750	0.910256
11	Initialization	42	0.859649	0.763158
12	Logic	54	0.944763	0.944046
13	Memory Address	20	0.997207	0.997207
14	No Requirement	15	0.966292	0.966292
15	Other	17	0.980226	0.905367
16	Range	11	0.983146	0.983146
17	Requirements	45	0.991429	0.991429
18	Reset Logic	24	0.918095	0.899048
19	Scaling Factor	27	0.983580	0.983580
20	Self-Test	11	1.000000	1.000000
21	Synchronization / Timing	37	0.905848	0.905848
22	System Integration	17	0.952514	0.952514
TOTAL		726		
AVERAGE			0.935589	0.923663

#### Original SPAR Statistics

It is interesting the “Other” category has 98% (91% optimized) confidence even though the SPARs in this category only share the fact that they are not in any other class. The “Other” SPARs could have been eliminated from the set if they had low confidence of classification (leaving 21 categories). The relative high confidence seems to validate the decision to leave “Other” in the data. The existence of “Other” also insures any SPAR will have a valid available classification.

## 4.2 Validate Terminology and Results

The models produced by MarianaExec were then tested with MarianaPredict on the same data. With the high confidence reported by MarianaExec, it was expected that about 93% of the SPARs would be correctly classified. The results were rather different:

CAT #	CATEGORY	INSTANCES	CONFIDENCE		HITS, AB, ERRORS	BATTING AVERAGE
			DEFAULT	OPTIMIZED		
1	Algorithm	78	0.908425	0.907509	0/78/1	0.0000
2	Configuration Management	40	0.990956	0.986434	13/40/0	0.3250
3	Data Definition	33	0.931358	0.915462	0/33/0	0.0000
4	Data Handling	45	0.822997	0.817829	12/45/0	0.2667
5	Documentation	83	0.893231	0.913043	44/83/0	0.5301
6	Equation / Calculation	19	0.974286	0.973333	0/19/0	0.0000
7	Failure Detection	11	0.994413	0.955307	0/11/0	0.0000
8	Failure Isolation	14	0.796610	0.806497	0/14/0	0.0000
9	Failure Management	36	0.877193	0.843275	6/36/1	0.1667
10	Incorrect Signal / Data	47	0.910750	0.910256	19/47/0	0.4043
11	Initialization	42	0.859649	0.763158	9/42/0	0.2143
12	Logic	54	0.944763	0.944046	1/54/0	0.0185
13	Memory Address	20	0.997207	0.997207	2/20/0	0.1000
14	No Requirement	15	0.966292	0.966292	0/15/0	0.0000
15	Other	17	0.980226	0.905367	0/17/0	0.0000
16	Range	11	0.983146	0.983146	0/11/0	0.0000
17	Requirements	45	0.991429	0.991429	0/45/0	0.0000
18	Reset Logic	24	0.918095	0.899048	0/24/0	0.0000
19	Scaling Factor	27	0.983580	0.983580	0/27/0	0.0000
20	Self-Test	11	1.000000	1.000000	1/11/0	0.0909
21	Synchronization / Timing	37	0.905848	0.905848	7/37/0	0.1892
22	System Integration	17	0.952514	0.952514	0/17/0	0.0000
TOTAL		726				
AVERAGE			0.935589	0.923663	114/726/2	0.1570

“Batting Average” Of MarianaPredict

Of the 726 SPARs, only 114 were categorized correctly. The majority (610) were not assigned to any category. The good news is only two SPARs were categorized into the wrong class. The results varied considerably by category and apparently independently of the number of training examples.

These results lead us to hypothesize the possibility of potential improvements in the MARIANA tool for increased productivity. There may also be more optimum numbers of categories or choices of categories. Any such efforts would require coordination with NASA Ames to better understand the interdependencies and are beyond the scope of the current effort.

Even though the productivity is low, the MARIANA tool is still useful for classifying software anomalies. It takes very little effort to run MARIANA (after set up) and it quickly classified 16% of anomalies correctly. The error rate is a very low 0.3% (points to a possibility of adjusting the algorithms to make MARIANA more likely to make category calls, albeit with a higher risk of incorrect calls).

### 4.3 Expand Data Set to Other Databases

With the MARIANA tool configured and validated against the previous data, it was time to see how it performed with different data. We collected an additional 2896 SPARs from flight-critical software on other aircraft programs. This data followed the same general format, but was compiled by different engineers on different programs over a different time frame. It should therefore represent how generally the developed models might apply to any flight critical software.

The new data was prepared with the pre-processing script. MarianaPredict was called on this data against each of the 22 models produced from training with the previous 726 SPARs. Yet another PERL script was used to make the successive calls.

```
$num_runs=22;

print "num_runs=$num_runs\n\n";

foreach $i (1..$num_runs){
    $run_printout = `java -Xmx512M -Djava.library.path=.\\ MarianaPredict -m test\\Cat_${i}.model -t
    ..\\data\\root_cause.txt -o test\\SparCat${i}test.csv`;
    print "Run $i output:\n$run_printout\n";
}
```

MarianaPredict Script

This created a file for each of the 22 categories with a determination on whether each of the 2896 records was in that file. These files were merged together. Positive numbers indicate a record is in that category. Negative numbers indicate the record is not in the category. The actual Raw Data is a 22 x 2896 matrix which is too large to provide here. MarianaPredict performance on the expanded data is summarized below:

	Number of SPARs	Different Categories	Number of Placements
	0	>4	0
	3	4	12
	4	3	12
	47	2	94
	264	1	264
	2578	0	0
TOTAL	2896		
# Placed	318	>0	382
% Placed	11%		

#### Expanded Data Statistics

318 of 2896 SPARs were categorized which represents 11% of the records. Of these, 54 SPARs were placed in multiple categories. The remaining 264 SPARs placed in a single category represent 9% of the total. It should be noted that the categories are not mutually exclusive, so placing a SPAR in multiple categories may not be incorrect. Indeed, that may be a benefit from an automated taxonomy; it should place SPARs in ALL relevant categories, while human experts tend to place them in only the most appropriate category. A quick check of the SPARs placed in 3 or 4 categories shows that some of those placements were in error. Many of the placements in 2 categories seem reasonable.

This placement performance is reasonably consistent with MarianaPredict on the original data (16% placed). A similar 0.3% error rate would equal 9 SPARs incorrectly assigned. While we did not submit the results to a detailed review by subject matter experts, a top level estimate is this is approximately the number of errors MarianaPredict produced on the expanded data.

As noted previously, it may be possible to tweak the MARIANA algorithms to force more record classifications if we accept a higher error rate. For the purposes of software anomaly taxonomy, this holds some promise. Taxonomy errors may be less important than failures to classify.

The results on the expanded data correlate well with the original data:

CAT #	CATEGORY	INSTANCES	PREVIOUS BATTING AVERAGE
1	Algorithm	0	0
2	Configuration Management	86	0.325
3	Data Definition	0	0
4	Data Handling	97	0.2667
5	Documentation	153	0.5301
6	Equation / Calculation	0	0
7	Failure Detection	0	0
8	Failure Isolation	0	0
9	Failure Management	0	0.1667
10	Incorrect Signal / Data	11	0.4043
11	Initialization	0	0.2143
12	Logic	0	0.0185
13	Memory Address	0	0.1
14	No Requirement	0	0
15	Other	0	0
16	Range	0	0
17	Requirements	0	0
18	Reset Logic	0	0
19	Scaling Factor	0	0
20	Self-Test	0	0.0909
21	Synchronization / Timing	35	0.1892
22	System Integration	0	0
TOTAL		382	

Mariana Predict Performance On Expanded Data  
Compared To Original Data

The categories with the most placements match the training data. Notice the number called in a category is highest for those categories which were previously correct the most often. This implies the

models for those categories are useful on a larger data set. Conversely, the models associated with Scaling Factor (for example) do not appear to be effective on either data set.

The implication is some of the categories are not modeled well. This could be due to the MARIANA algorithms or the category choices themselves. A reasonable effort was already expended on choosing the categories and any further improvements really require coordination with the MARIANA experts.

## 5.0 Conclusions

The potential for auto-classification algorithms to generate a failure taxonomy from software problem reports appears promising. Because software anomalies are generally documented in free form text, auto-classification tools capable of directly examining this text without significant manual supervision would be very helpful in understanding software health trends as the databases of software anomalies are far too large and subject matter experts are too valuable to inspect every record manually to develop a failure taxonomy. Auto-classification tools may be the only practical method of “mining” the vast amount of available data for software anomaly trends and failure modes

The software failure taxonomy developed under CD-SHM was partially validated by the reproduction of the identified patterns of software failures over a much larger dataset. This provides evidence that at least some of the identified failure types are fundamental to flight critical software development. Overall results showed potential for auto-classification algorithms to derive a failure taxonomy, and the Mariana tool was moderately successful in classifying the anomalies identified under previous tasking. The use of auto-classification algorithms may facilitate development of an even stronger software failure taxonomy incorporating the concept of software failures belonging to more than one category. Findings also suggest that the taxonomy developed under CD-SHM may benefit from additional tuning to reduce ambiguity and offer less subjective boundaries.



## **6.0 References**

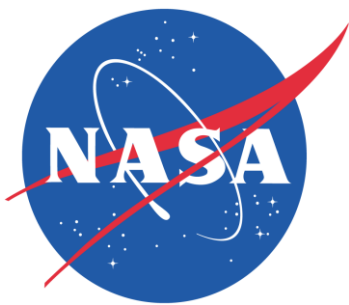
Windows® and Vista® are registered trademarks of the Microsoft Corporation.

HP® is a registered trademark of the Hewlett-Packard Corporation

Core™ 2 Duo is a trademark of the Intel Corporation

## **7.0 Appendix A**

The full text of the Concept Development for Software Health Management (CD-SHM) final report from August 4<sup>th</sup>, 2009 is included in its entirety as an appendix to this report. It has been re-formatted to be more compatible with this document.





FZM-9673-06  
04-August-2009

# **Concept Development for Software Health Management (CD-SHM)**

## **Final Report**

Contract: NNL06AA08B

Delivery Order: NNL07AB06T

CDRL A003

Prepared and Approved for Release by:

A handwritten signature in blue ink, appearing to read "Walter A. Storm".

Walter A. Storm – Program Manager

LOCKHEED MARTIN AERONAUTICS COMPANY

DISTRIBUTION STATEMENT A. Approved for Public Release. Ref: AER200907017

Copyright © 2009 by Lockheed Martin Corporation.  
All rights reserved.

## Contents

Foreword .....	5
Introduction .....	6
Approach      6	
Preparing the Data .....	6
Classification Details .....	6
Creating the Baseline .....	7
Creating the Failure Taxonomy .....	7
Analysis Results .....	8
Failure Classes 8	
Algorithm .....	8
Bus Interface .....	9
Configuration Management (CM) .....	9
Compiler Error .....	10
Data Definition .....	10
Data Handling .....	10
Documentation .....	11
Hardware .....	12
Input-Output (I/O) System .....	12
Implementation .....	12
Inter-process Communication .....	13
Performance .....	13
Self-Test .....	13
System Integration .....	14
Tools .....	15
User/Pilot .....	15
Error Analysis 15	
Background .....	16
The Risk Priority Number .....	16
Detailed Class Analysis .....	18
RPN Component Analysis .....	19
Bus Interface Error Class Profile .....	23
Configuration Management Error Class Profile .....	24

Data Definition Error Class Profile .....	24
Data Handling Error Class Profile .....	25
Inter-Process Communication Error Class Profile .....	25
Input/Output System Error Class Profile .....	26
Self-Test Error Profile .....	26
System Integration Error Class Profile .....	27
Root Failure Cause and Effect Relationship Analysis	27
Background.....	27
Ground Rules .....	28
Overview of Root Failure Cause and Effect Relationship Chart .....	28
Documentation and External Problems Category .....	29
Requirements Category .....	30
Configuration Management Category .....	31
Algorithm Category .....	31
System Integration / Communication Category .....	35
Self-Test Category .....	39
Application of Data Analysis Results to Evaluating Future Technologies .....	39
References.....	43

## **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:

- Delete all the blank sections
- Delete unimportant sections for this project. (i.e. User ID, date,...etc)
- Delete 'cancelled' or 'analysis' in status
- Delete 'external', 'duplicate', 'not a problem', 'suspended' in final resolution
- Delete 'No' in confirmed problem
- 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.

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.

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.

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 from 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
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

Algorithm Failure Class (Cont'd)	
Failure Type	Definition
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
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
Data Handling Failure Class(Cont'd)	
Failure Type	Definition
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<sup>1</sup>.

---

<sup>1</sup> See Error Analysis – Rankings by Occurrence.

## ***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<sup>2</sup>. We recognize 4 distinct I/O System error types.

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.

---

<sup>2</sup> See Error Analysis – Rankings by Severity.

### ***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)
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
System Integration Failure Class (Cont'd)	
Failure Type	Definition
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<sup>3</sup>, 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<sup>4</sup>, 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:

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

## ***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.

---

<sup>3</sup> i.e. quantitative or probabilistic risk assessment

<sup>4</sup> See [http://en.wikipedia.org/wiki/Failure\\_mode\\_and\\_effects\\_analysis](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 Introduction Phase	Defect Detection 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

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

elements from the entire taxonomy whose RPN is greater than 100.

### *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 1, at right.

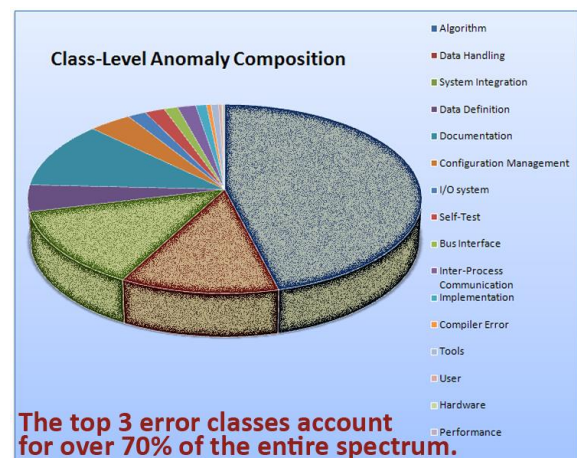


Figure 1 - Class-Level Analysis



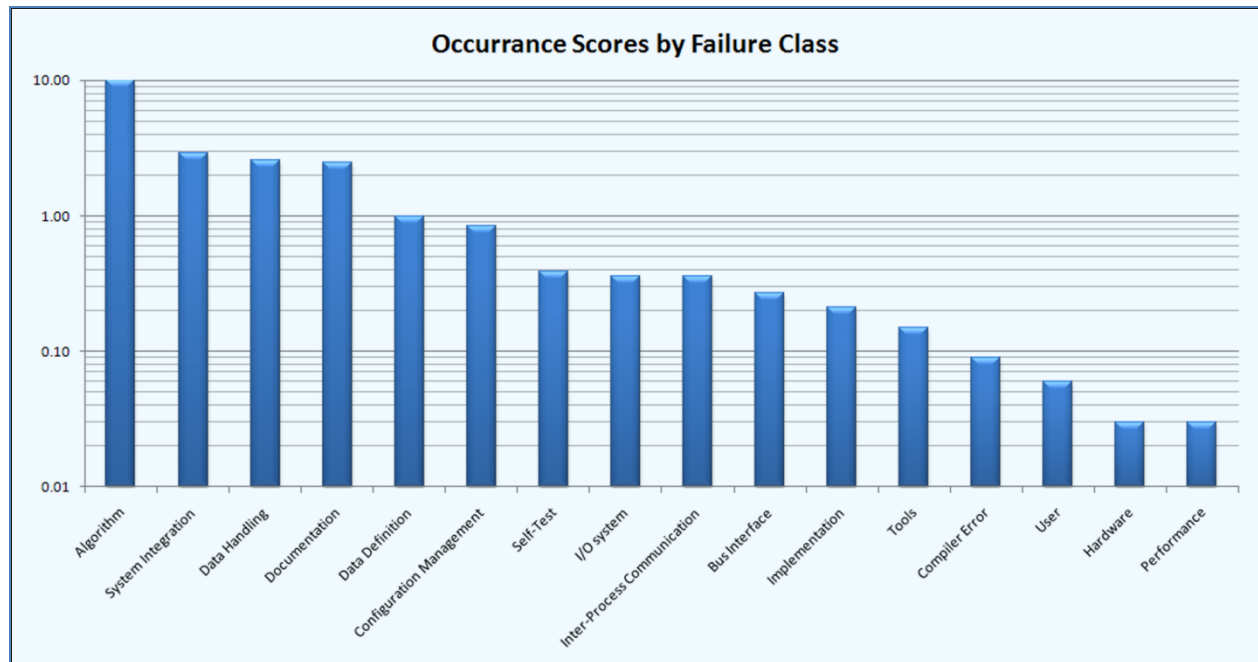
**Figure 2 – Class-Level Error Profile**

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 2 below.

### ***RPN Component Analysis***

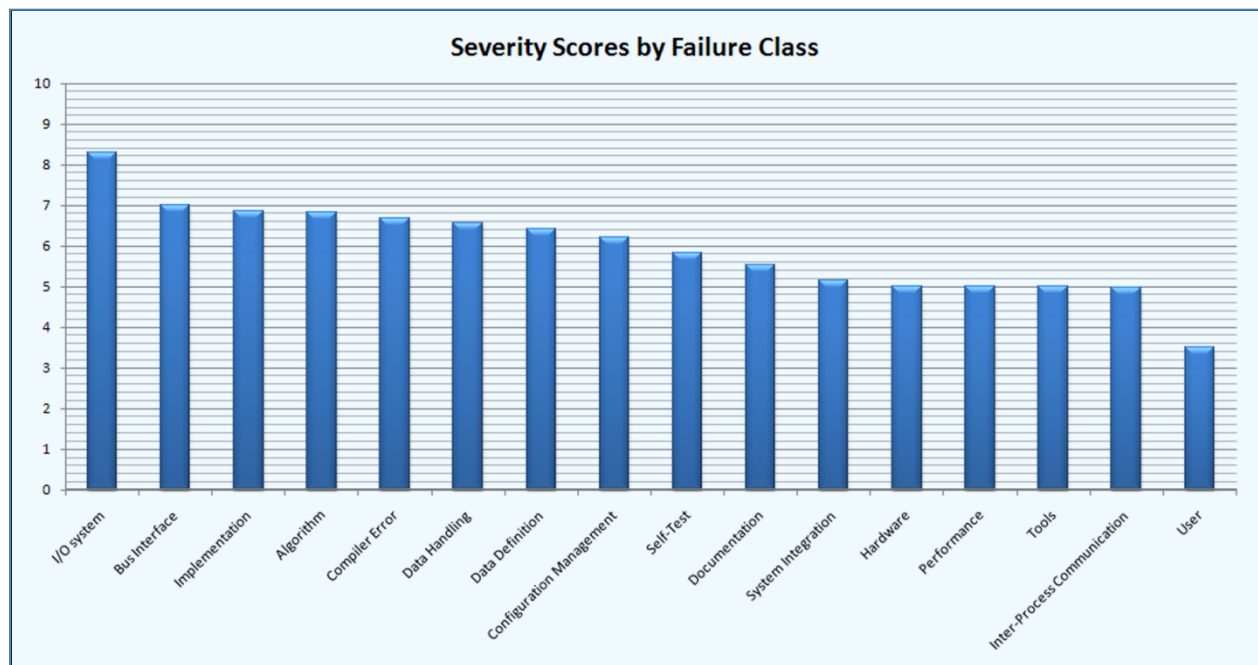
At this point, we discuss the individual parameters of RPN for the failure class analysis. The most dominant discriminator for RPN analysis is the occurrence parameter. There is some distinct differentiation between severity and detection as well, but not nearly as drastic as occurrence. The following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

### *Occurrence Parameter*

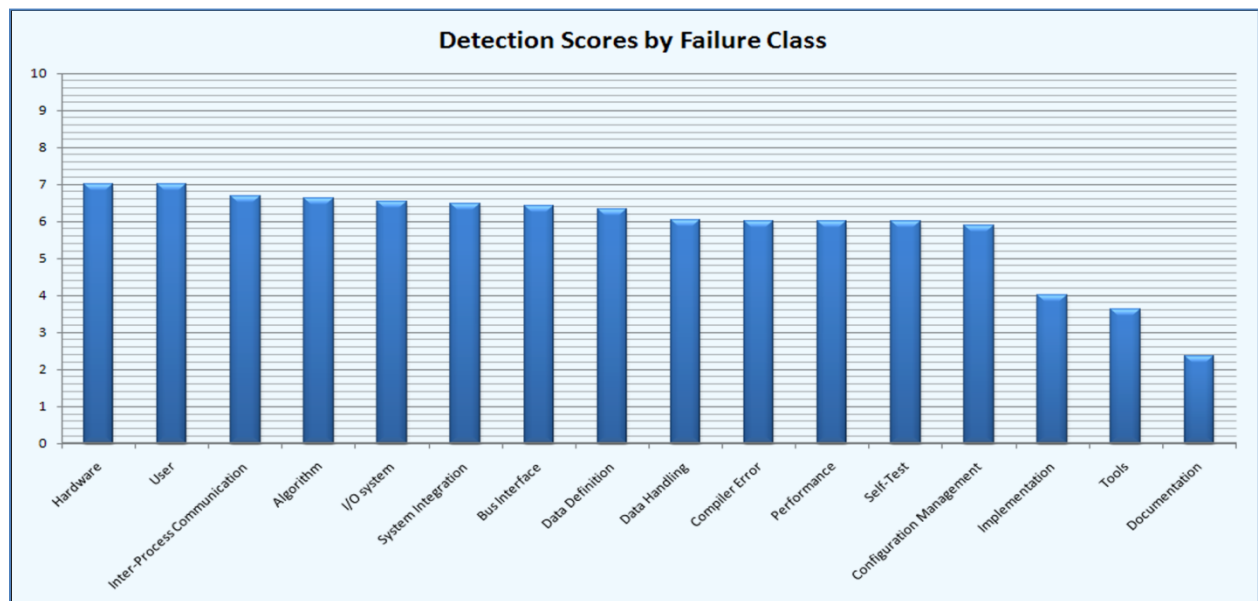


**Figure 3 – Occurrence Dimension**

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.



**Figure 4 – Severity Dimension**  
**Severity Parameter**



**Figure 5 – Detection 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

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

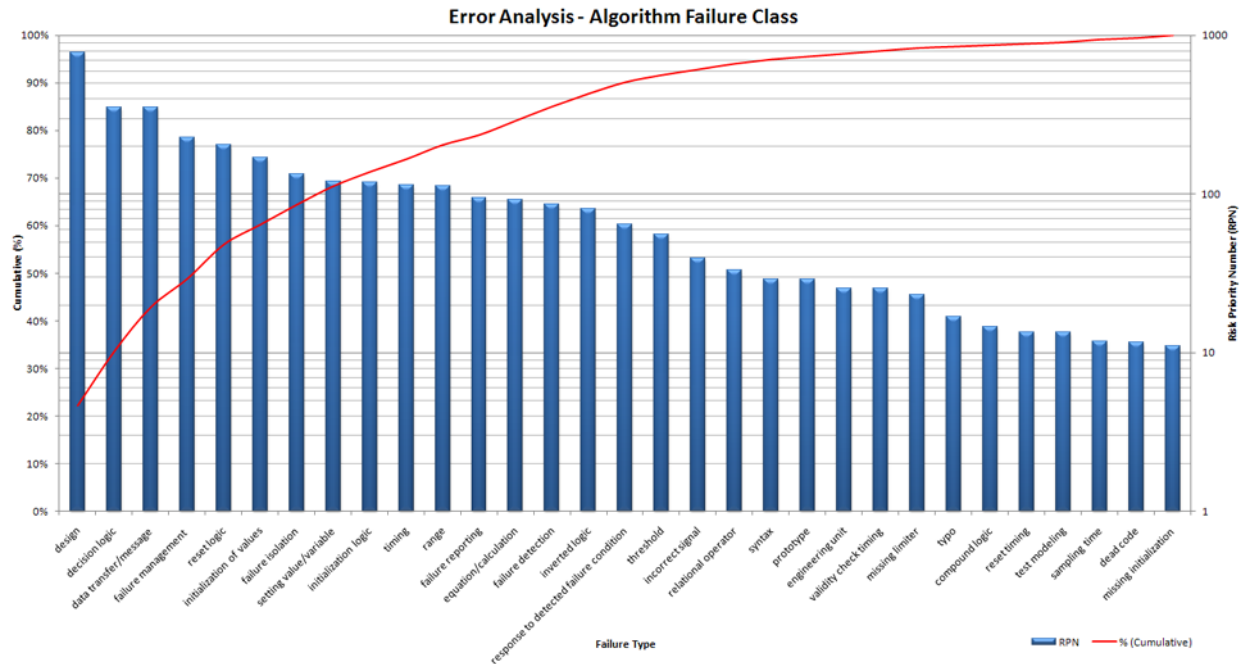


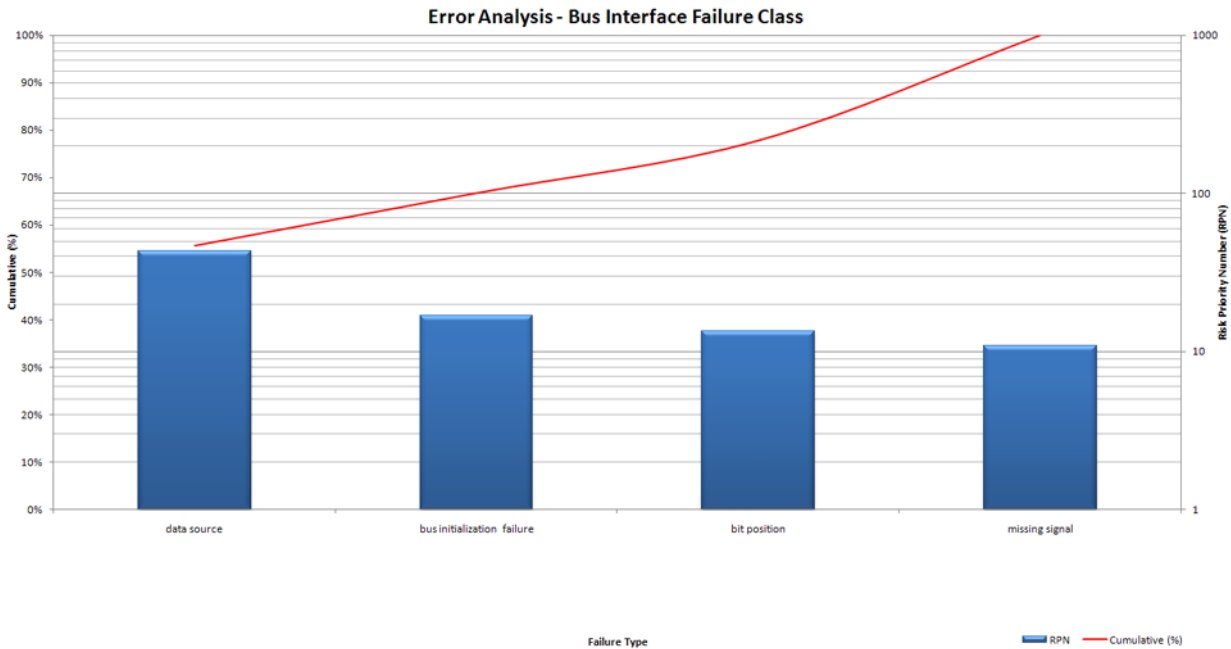
Figure 6 - Algorithm Error Profile

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***



**Figure 7 – Bus Interface Error 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.



### 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.

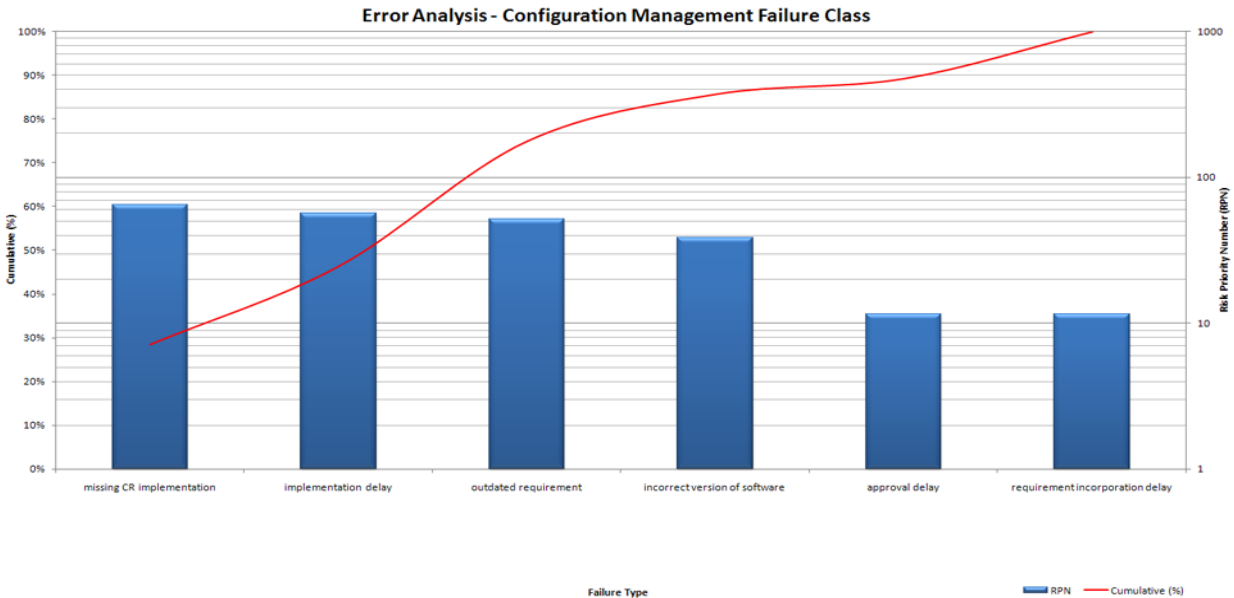


Figure 8 – Configuration Management Error Profile

### Data Definition Error Class Profile

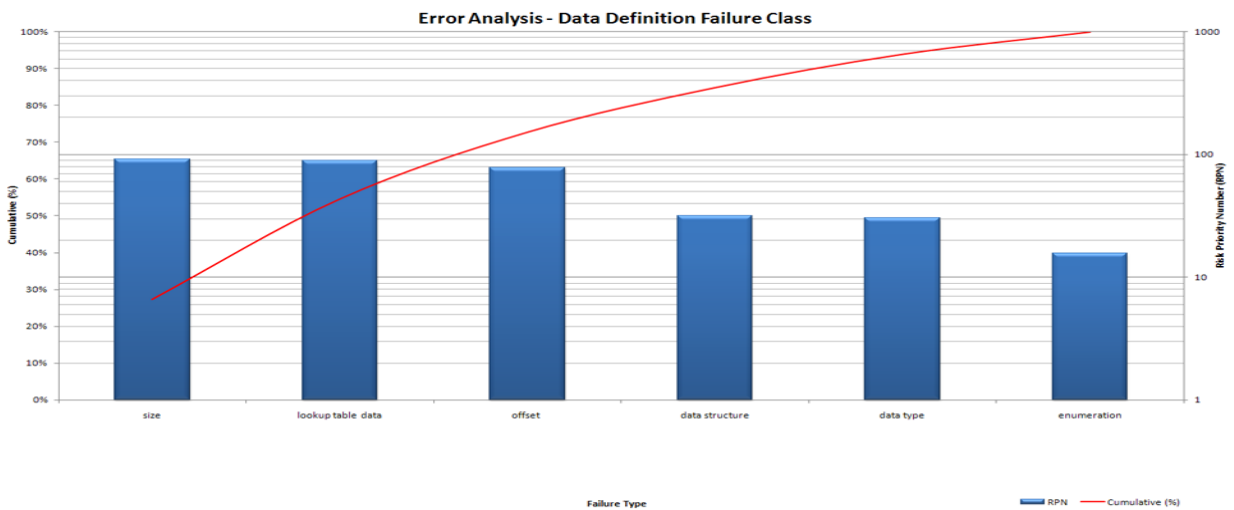


Figure 9– Data Definition Error Profile

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

### Data Handling Error Class Profile

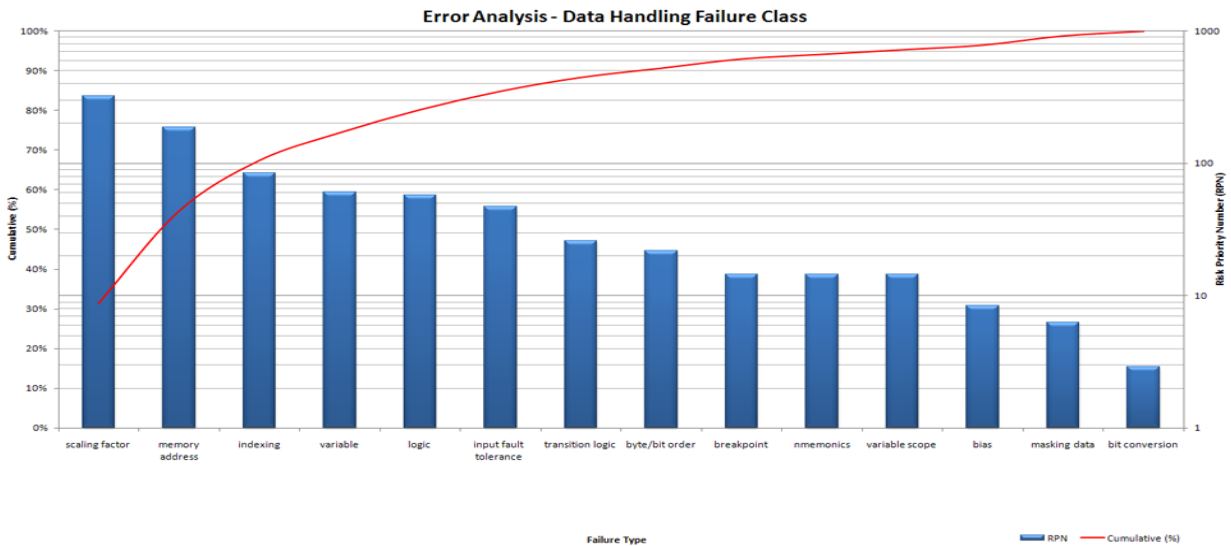


Figure 10 – Data Handling Error Profile

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.

### Inter-Process Communication Error Class Profile

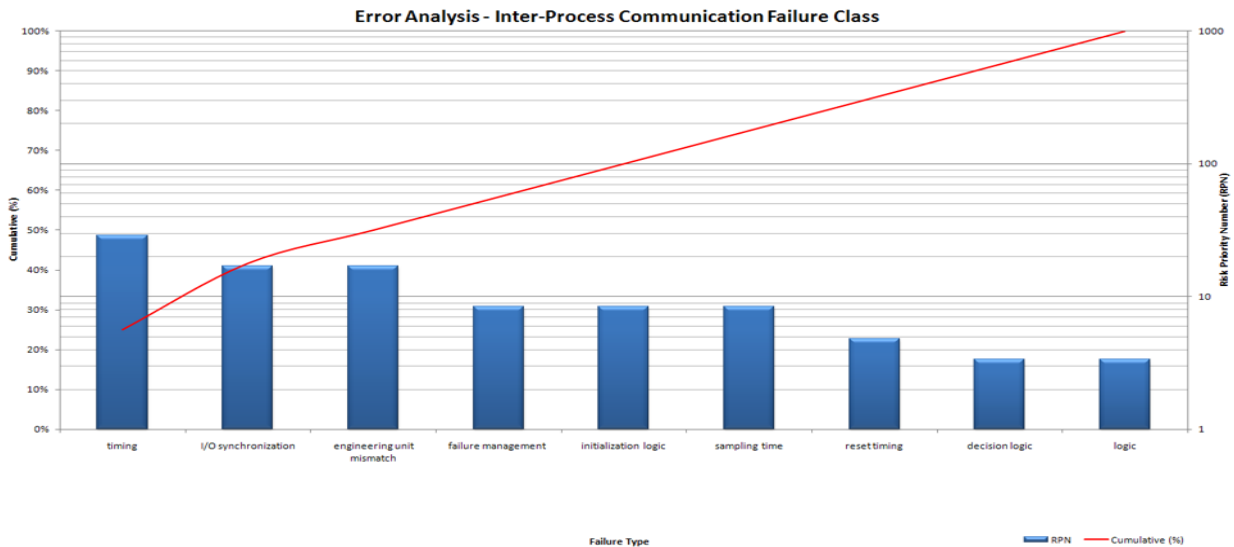
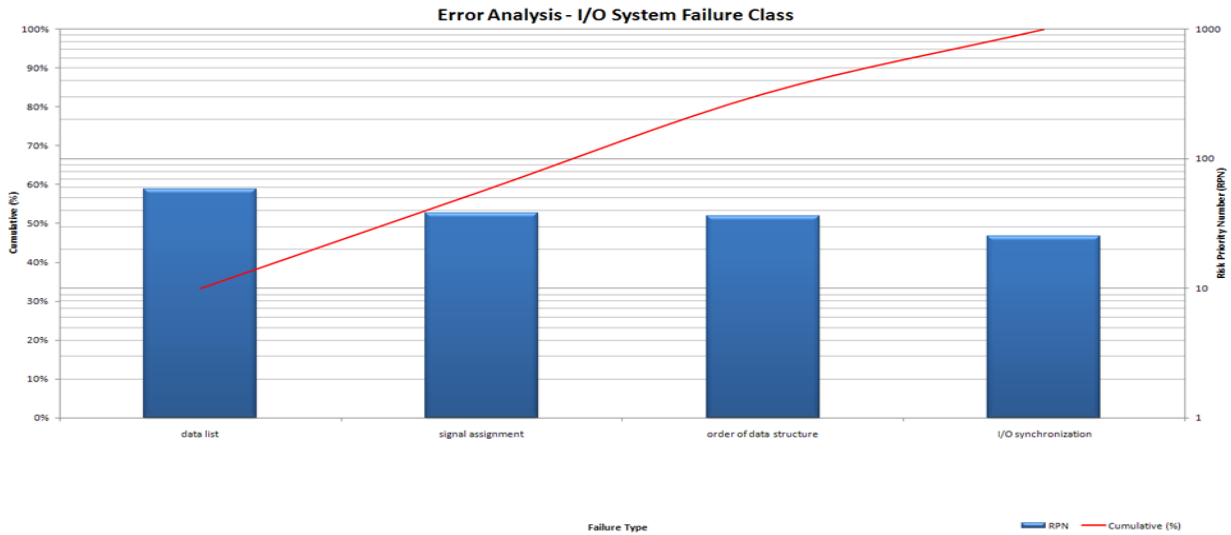


Figure 11 – IPC Error Profile

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.

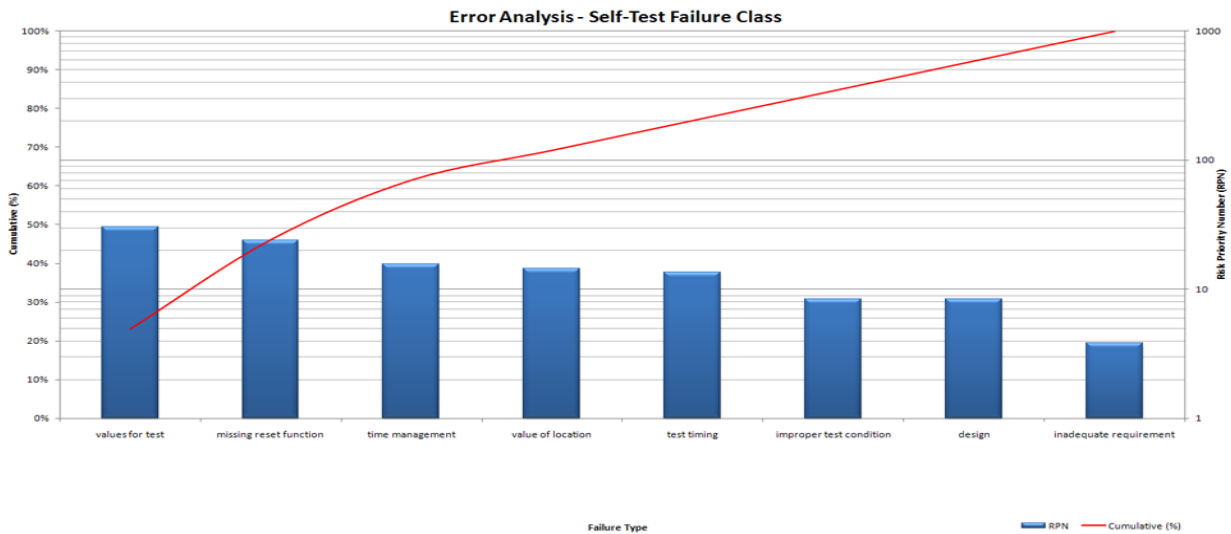
### ***Input/Output System Error Class Profile***

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***



**Figure 13 – Self-Test Error Profile**

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

### 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.

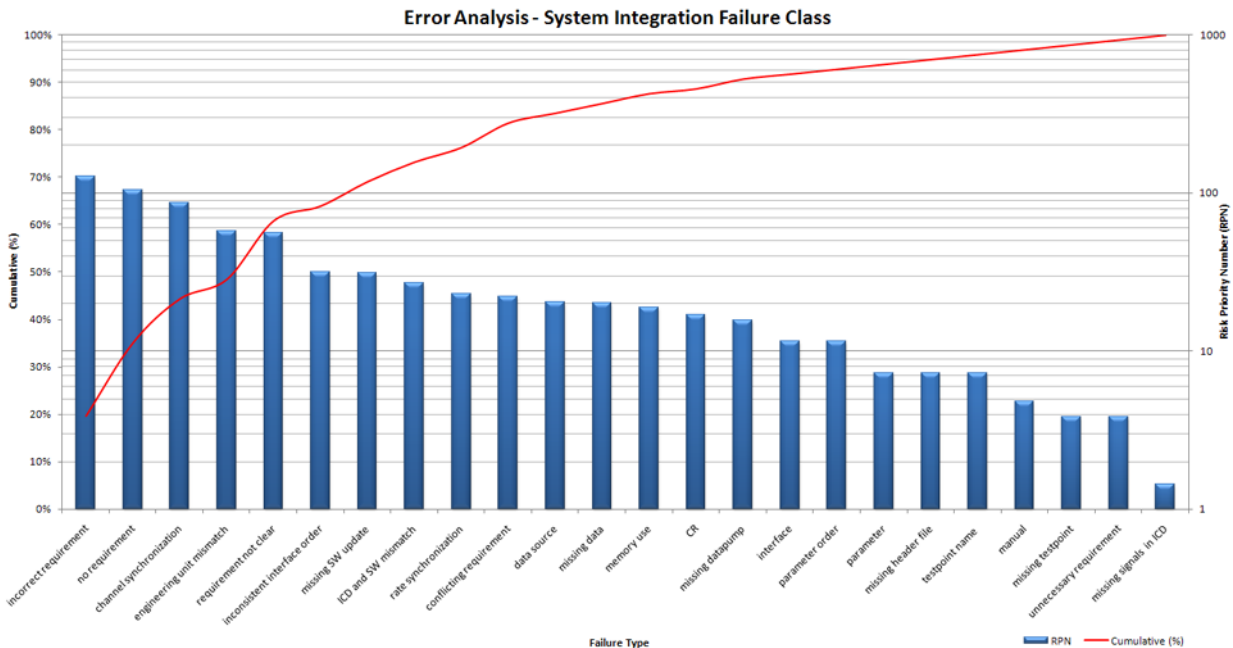


Figure 14 – System Integration Error Profile

### 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***

The relationships were not necessarily direct cause-effect relationships, but were rather a logical correlation between the two.

An error or confusion in one area might tend to imply an error or confusion in the related area.

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.

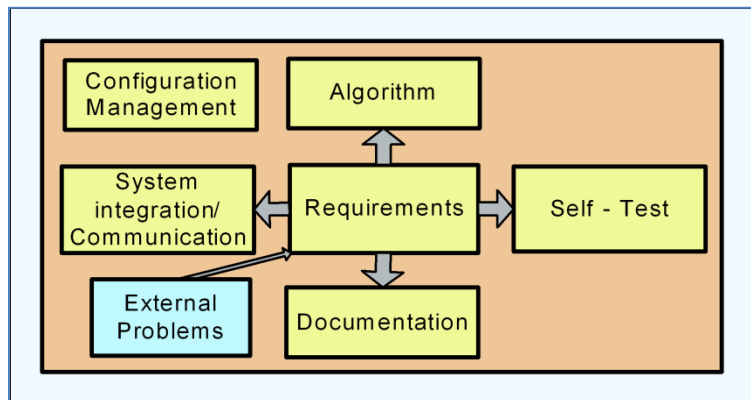
We color coded the 114 failure types to indicate their RPN percentile among the failure types by:

- 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, External Problems, Documentation, Algorithm, System Integration/Communication, and Self-Test.



**Figure 15 – Related Root Failure Categories**

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.

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

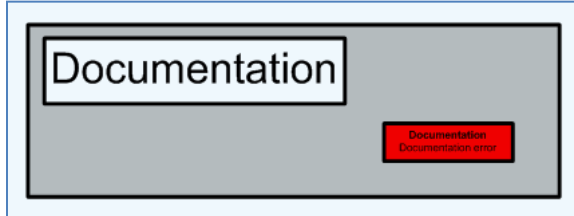
Documentation: 1 red

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 – Documentation 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.

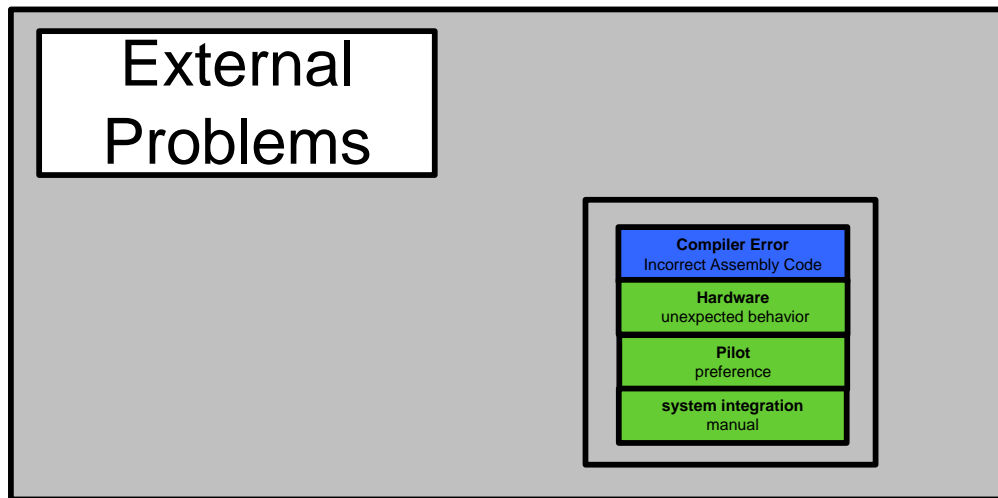


Figure 17 – External Problems Category

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 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

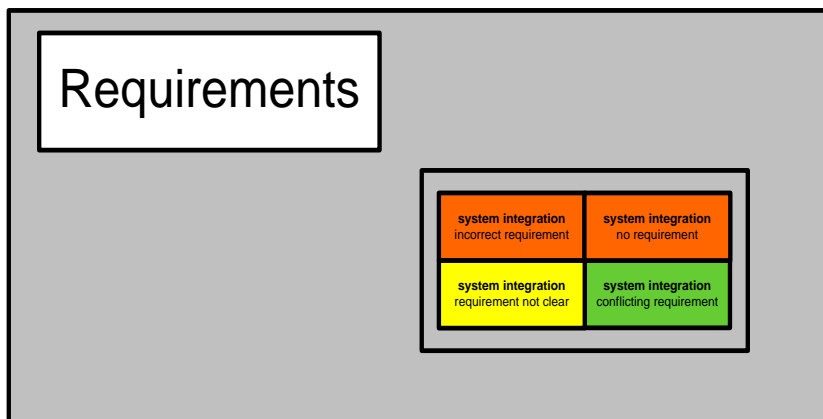
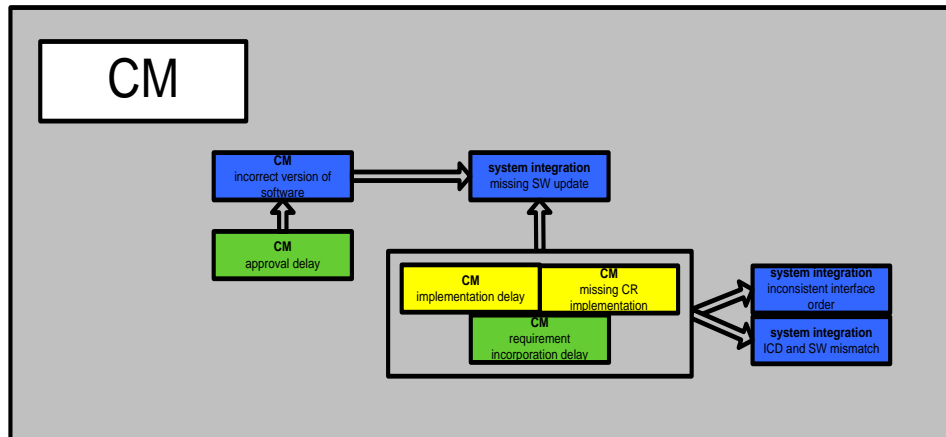


Figure 18 – Requirements Category

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 – 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

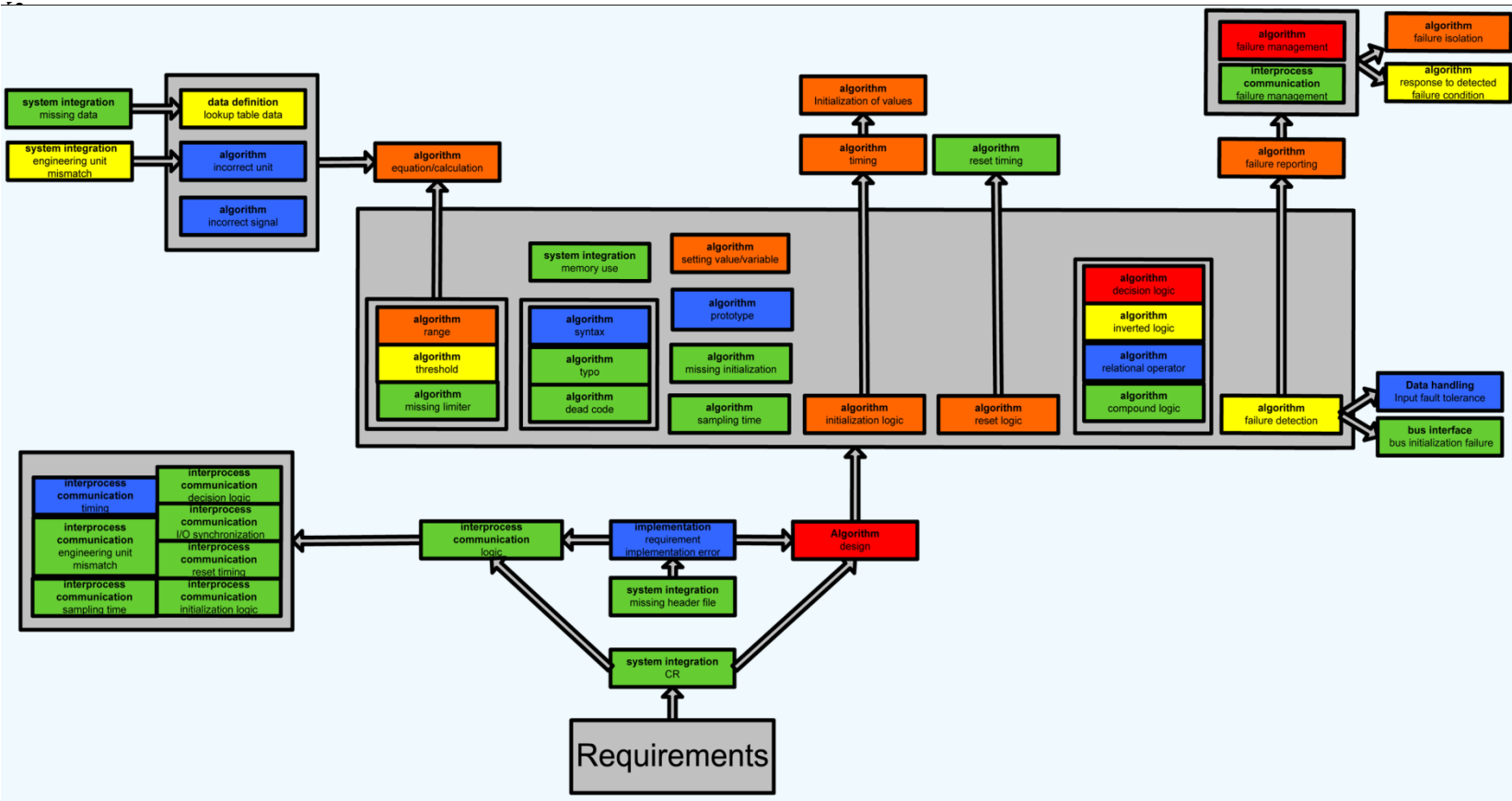
diagram because of their relationships with the “configuration management” failure types. The two yellow blocks, “CM: implementation delay” and “CM: missing CR implementation” are grouped together with the **green “CM: requirement incorporation delay” to collect the problems with delays in already approved 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.

### *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.



**Figure 20 – Algorithm Category**

***System Integration / Communication Category***

SWAT-V Appendix A

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

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

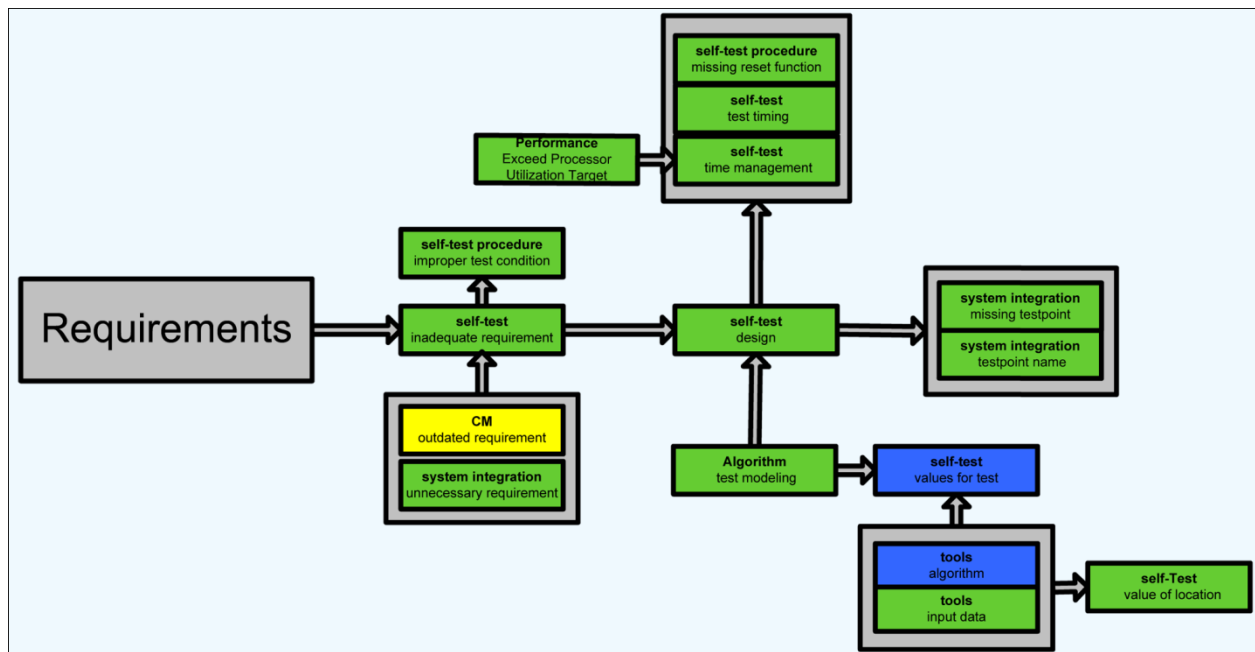


Figure 22 – Self-Test Category

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.

### Application of Data Analysis Results to Evaluating Future Technologies

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



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.

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.

Select technologies/ methods that you want to examine.

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)

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)

RPN	Root Failure	Occurrence					Severity					Detection				
		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.

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

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.

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.