Software Failure Modes Effects Analysis Overview

Copyright, Ann Marie Neufelder, SoftRel, LLC, 2020
ann.neufelder@missionreadysoftware.com
www.missionreadysoftware.com
• This presentation is copy protected and licensed to one person who has registered for the class
• You may not
  • Use any part of this presentation in a derivative work including but not limited to presentations, conferences, published articles or books, theses, etc.
  • Convert this presentation to any other format other than PDF.
• Violators will be prosecuted to the fullest extent of the law
Additional information

• “Effective Application of Software Failure Modes Effects Analysis” book.
• The SFMEA toolkit contains a complete list of software failure modes and root causes.
• Has been developing software and managing software engineers since 1984
• Has been applying software failure modes effects analysis since 1986 on complex software intensive engineering systems
• Has conducted numerous software FMEAs in medical, defense, space, energy and electronics industries
• Has reviewed numerous software FMEAs for large organizations acquiring software intensive systems
• Has read more than 200,000 software problem reports and assessed the failure mode and root cause of each
• Has identified more than 400 failure mode/root cause pairs for software
• Wrote the SFMEA webinar for NASA
• Has seen the good, bad and ugly with regards to effective software FMEAs
• Chair-person for the IEEE 1633 Recommended Practices for Software Reliability, 2016
• Published current industry guidance for software failure modes effects analysis as referenced by IEEE 1633 and SAE TAHB0009
• Invented the only industry accepted method to predict the number of software failures before the code is even written

7/25/2020
Software Failure Modes Effects Analysis

Introduction
1.0 Introduction

Software is increasing in size

• The increase in size of F16A to F35 is just one example[1]
• With increased size comes increased complexity and increased failures due to software

SIZE IN SLOC OF FIGHTER AIRCRAFT SINCE 1974

Key benefits of Software FMEAs

- Addressing one failure mode could mean eliminating several failures
  - i.e. if the software engineers failed to consider what the software shall do when there is a hardware fault, this effects all hardware interfaces which can have wide reaching effect

- The SFMEA output can be used for any of these purposes
  - Develop better software requirements and/or make requirements reviews more effective
    - Abnormal behavior/alternative flows that might be missing from the requirements or design specifications
    - Unwritten assumptions
  - Develop a better design and/or make design reviews more effective
    - Features that need fault handling design
    - Defects that easier to see when looking at the design or code but difficult to see during testing
  - Design a health management system (HMS) or Built In Test (BIT)
    - Identify defects that cannot be addressed by redundancy or other hardware controls
  - Develop a test plan that tests for more than the happy day scenarios
    - Identify tests that find failures as opposed to confirm requirements
    - Identify failures in systems that are difficult or expensive to test (i.e. spacecraft, missiles, medical equipment, etc.)
  - Develop software/supporting documentation that minimizes user error
How to NOT conduct a software FMEA

• Treating the software as it’s a black box – Early guidance on software FMEA recommended the black box approach which doesn’t provide value. The *functional* viewpoint has proven effectiveness.

• Assigning the analysis to someone who’s not a software engineer. Reliability engineers may facilitate but software engineers understand the failure modes.

• Assigning the software will work. Instead, one must assume that:
  • Specifications are missing something crucially important
  • Specifications are incorrect
  • Specifications are sufficient but design/code doesn’t address them

• Not having sufficient written specifications
  • SFMEAs are much more effective when conducted on use cases and design than high level requirements

• Trying to analyze everything
  • Most productive when they focus on riskiest software, riskiest functions, with most likely failure modes. More effective when preceded by a root cause analysis.

• Conducting the analysis too early or too late

• Spending too much time spent assessing likelihood.
  • SFMEAs are NOT used to calculate failure rates. Once root cause of failure mode is removed correctly, failure event doesn’t recur. That’s different than for hardware.
1.0 Introduction

At highest level - three things that can and will go wrong with software

- The sources of all software faults lie in the below three basic sources
- Software FMEA analyst must understand and consider all three sources
- It’s common for SFMEAs to assume that all *software* specifications are both complete and correct and that requirements based testing will find all defects
- The only thing that can be assumed is that the *customer* specification is correct

<table>
<thead>
<tr>
<th>Top level software fault</th>
<th>Why its not found prior to operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>The software specifications are missing crucially important details</td>
<td>These aren’t found in requirements based testing because only the written part is tested.</td>
</tr>
<tr>
<td>The software specifications are themselves faulty and hence the code is faulty.</td>
<td>These are found in requirements based testing because the code does what the stated requirement says</td>
</tr>
</tbody>
</table>
| The software engineer may not always write the code as per the written specifications | • These faults can happen with very large systems and/or insufficient traceability.  
• These can get through requirements based testing undetected if the code works with some inputs but not others |
1.0 Introduction

Common software root causes summarized

- Faulty functionality
  - Software does the wrong thing
  - Software fails to do the right thing
  - Software feature is overengineered

- Faulty error handling - Inability to detect and correctly recover from
  - Computational faults
  - Faults in the processor
  - Memory faults
  - Environment faults (Operating system)
  - Hardware faults
  - Power faults
  - Power not checked on startup or power down
  - Faulty I/O

- Faulty sequences/state
  - Missing safe state and/or return to safe state
  - Prohibited state transitions allowed

- Faulty timing
  - Race conditions
  - Timeouts are too big or too small
  - Software takes too long to execute and misses timing window

- Faulty data handling
  - Data and interface conflicts
  - Insufficient handling of invalid data

- Faulty algorithms
  - Crucially important algorithm isn’t specified
  - Algorithm is specified incorrectly

- Faulty logic
  - Control flow defects
  - Faulty comparison operators

- Faulty usability
  - Insufficient positive feedback of safety and mission critical commands
  - Critical alerts aren’t obvious

- Faulty processing
  - Software behaves erratically or can’t start after a loss of power or user abort

- Endurance/peak load
  - Safety and mission are degraded when system remains on for extended period of time
  - Operational status isn’t monitored
1.0 Introduction

FAQ—Which of these failure modes is most common? Most severe effect?

• Severity depends on the feature that has the failure mode
  • All software failure modes can result in catastrophic failure and all can result in a non-critical failure.
  • If a mission critical feature has one of these failure modes the effect will generally be severe, however, non-mission critical features encountering these failure modes may still have a severe consequence.

• Likelihood depends on the application type and the development practices

<table>
<thead>
<tr>
<th>Failure mode</th>
<th>When it’s more likely</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faulty functionality</td>
<td>Common if the software requirements are too high level</td>
</tr>
<tr>
<td>Faulty error handling</td>
<td>Most common for nearly every organization because software engineers almost never consider failure space</td>
</tr>
<tr>
<td>Faulty sequences/state</td>
<td>More likely if the software engineers don’t do detailed state design at the lower levels and highest level</td>
</tr>
<tr>
<td>Faulty timing</td>
<td>More likely if there aren’t timing diagrams</td>
</tr>
<tr>
<td>Faulty data handling</td>
<td>More likely if there aren’t well defined software interface design documents and data definitions</td>
</tr>
<tr>
<td>Faulty algorithms</td>
<td>Applies to mathematically intensive systems</td>
</tr>
<tr>
<td>Faulty logic</td>
<td>Common for all software systems and is due to insignificant logic design at the lower levels</td>
</tr>
<tr>
<td>Faulty usability</td>
<td>Common if the user has to make quick decisions that are important</td>
</tr>
<tr>
<td>Faulty processing</td>
<td>Most common for systems that may encounter interrupted power supply</td>
</tr>
<tr>
<td>Endurance issues</td>
<td>Most visible for systems that do have an uninterrupted power supply</td>
</tr>
<tr>
<td>Peak loading issues</td>
<td>Applies to systems that can be used by more than one end user at the same time</td>
</tr>
</tbody>
</table>
A few real world failure events due to these root causes

• Faulty error handling – Apollo 11 lunar landing, ARIANE5, Quantas flight 72, Solar Heliospheric Observatory spacecraft, Denver Airport, NASA Spirit Rover (too many files on drive not detected)

• Faulty data definition - Ariane5 explosion 16/64 bit mismatch, Mars Climate Orbiter Metric/English mismatch, Mars Global Surveyor, 1985 SDIO mismatch, TITANIV wrong constant defined

• Faulty logic – AT&T Mid Atlantic outage in 1991

• Faulty timing - SCUD missile attack Patriot missile system, 2003 Northeast blackout
  • Race condition - Therac 25

• Peak load conditions - Affordable Health Care site launch

• Faulty usability
  • Too easy for humans to make mistakes – AFATDS friendly fire, PANAMA city over-radiation
  • Insufficient positive feedback of safety and mission critical commands – 2007 GE over-radiation

The above illustrates that history keeps repeating itself because people assume root causes from other industries/applications are somehow not applicable.

Lesson to be learned – the root causes are applicable to all software. It’s the hazards/effects that result from the root causes that are unique
1.0 Introduction

Real example of faulty data and faulty error handling

Loss of $370 million payload

On June 4\textsuperscript{th}, 1996 launch vehicle veered off course and self-destructed 37 seconds into the flight, at an altitude of 4km and a distance of 1km from the launch pad.

Guidance system shut down and passed control to an identical redundant unit which had the identical software defect.

Unnecessary course correction.

Primary defect--Faulty data - The guidance system’s computer attempted to convert the sideways velocity of the rocket from a 64 bit format to a 16 bit format. The result was an overflow.

Related defect- Faulty error handling - The guidance software had not been designed to trap for overflows because it was thought that the overflow could never happen.

Related defect- One size fits all error handling. When the overflow occurred the computer reboot by design. It should handled the fault differently.
Whether the life cycle model is waterfall, incremental, spiral, etc. The best time to do the particular SFMEA is shown above. With incremental models, there may be more than one iteration.
Prepare Software FMEA

Define scope

For each use case, use case steps, SW requirement, interfaces, detailed design, user manuals, Installation scripts ... (as applicable based on selected view point)

Analyze applicable failure modes

Identify root causes(s) for each failure mode

Analyze failure modes and root causes

Identify what can go wrong

Select View points

Gather artifacts

Identify riskiest functions

Identify resources

Set ground rules

Define likelihood and severity

Select template and tools

Tailor the SFMEA

Identify equivalent failure modes

Identify local/subsystem/system failure effects

Identify controls

Identify corrective actions

Revise RPN

Identify compensating provisions

Mitigate

Identify riskiest functions

Identify consequences

Identify severity and likelihood

Identify corrective actions

Revise RPN

Identify riskiest functions

Identify consequences

Identify severity and likelihood

Identify controls

Identify compensating provisions

Mitigate

Identify equivalent failure modes

Generate a Critical Items List (CIL)

Generate CIL

Step 1

Prepare the Software FMEA
2.0 Prepare for the SFMEA

To perform an effective SFMEA - first understand what can and usually does go wrong

1. The software engineer will not write code that’s not specified.

2. The software test engineers may test the written requirements but won’t test what’s not in writing.

3. The software engineer may not always write the code as per the stated requirements or specifications.
   - It cannot be assumed that even when requirements are complete and unambiguous that the code will be written correctly.

4. The software engineer may “guess” if presented with more than one option for implementing a requirement.
   - The wrong guess can result in code that causes a system failure.

5. It cannot be assumed that “thorough testing” will uncover all software failures prior to operation.

6. The failure modes from a past similar system are likely to be relevant for the current system.
Example of common power checking fault

This is a specification pertaining to the initialization of a system:

*The software shall check for voltages within* \(<x>\) *and* \(<y>\)

• There are no other statements regarding the voltages.
• The software development plan indicates that there will be “requirements” based testing. There is no indication of stress testing or fault injection testing.

This is what’s wrong...

• The software specification doesn’t say what the software shall do if the voltages are out of range.
• The software specification doesn’t explicitly say that the software does not continue if voltages are out of range.
  • Technically the specification is passed if the software checks the voltages regardless of what it does when the voltages are out of range.
• Hence there’s no reason to believe that there will be code to handle what happens if the voltages aren’t within \(<x>\) and \(<y>\).
• Since there is only requirements based testing it is very likely that voltages out of range won’t be tested.

This is what can happen if the voltages are out of range...

• The initialization completely stops (this is called a dead state)
• The initialization proceeds to the next state when it should not (this is called an inadvertent state transition)
Example of common data logging fault

This is the specification for the logging feature for a mission and safety critical system:

1) The software shall log all warnings, failures and successful missions.
2) At least 8 hours of operation shall be captured
3) Logging to an SD card shall be supported in addition to logging to the computer drive

This is what you know about the software organization and software itself

1) Logging function will be called from nearly every use case since nearly every use case checks for warnings, failures and successes
2) Testing will cover the requirements. But no plans to cover stress testing, endurance testing, path testing, fault insertion testing.
3) Software engineers have discretion to test their code as they see fit.
4) There is a coding standard but there is no enforcement of it through automated tools and code reviews only cover a fraction of the code
2.0 Prepare for the SFMEA

Example

• These are the faults that can/will fall through the cracks
  • No checking of read/write errors, file open, file exist errors which are common
  • No rollover of log files once drive is full (may be beyond 8 hours)
  • No checking of SD card (not present, not working)
  • Logging when heavy usage versus light or normal usage (might take less than 8 hours to fill drive if heavy usage)
  • Is both SD card and drive to be written to or does user select?

• This is why these faults aren’t found prior to operation
  • No one is required to explicitly test these faults/scenarios
  • No one is required to review the code for this fault checking
  • No one is required to test beyond 8 hours of operation

• This is the effect if any of these faults happens
  • Entire system is down because it crashes on nearly every function once drive is full, SD card removed, file is open or read/write errors

• When conducting a SFMEA one cannot assume that best practices will be followed unless there is a means to guarantee that
Example of how specification can be incorrectly coded

- Even if previously discussed issues are specified - these things can happen with the logging function due to coding mistakes
  - logs the wrong status
  - fails to log any status
  - takes too long to log the status (log file has stale timestamps)
  - logs corrupt data
  - deletes the entire log file
  - fails to acquire 8 hours of operation under any scenario
## Identify most relevant viewpoint for a particular software version

<table>
<thead>
<tr>
<th>FMEA</th>
<th>When this viewpoint is relevant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>Any new system or any time there is a new or updated feature/use case/requirements/design.</td>
</tr>
<tr>
<td>Interface</td>
<td>Anytime there is complex hardware and software interfaces or software to software interfaces.</td>
</tr>
<tr>
<td>Detailed</td>
<td>Almost any type of system is applicable. Most useful for mathematically intensive functions.</td>
</tr>
<tr>
<td>Maintenance</td>
<td>An older legacy system which is prone to errors whenever changes are made.</td>
</tr>
<tr>
<td>Usability</td>
<td>Anytime user misuse can impact the overall system reliability.</td>
</tr>
<tr>
<td>Serviceability</td>
<td>The software is mass distributed and remotely installed/updated as opposed to installed in a factory, authorized service personnel or controlled environment.</td>
</tr>
</tbody>
</table>
Do not analyze what the software *is* – analyze what the software *does*

- Hardware FMECA focuses on the configuration items. Software FMECA is less effective with that approach.
- *Focus should be on what the software is required to do and not the CSCI unit itself.*
- A CSCI typically performs dozens, hundreds or even thousands of functions so the below is too open ended

<table>
<thead>
<tr>
<th>LRU</th>
<th>Failure mode</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executive CSCI</td>
<td>CSCI fails to execute</td>
<td>Doesn’t address states, timing, missing functionality, wrong data, faulty error handling, etc.</td>
</tr>
<tr>
<td>Executive CSCI</td>
<td>CSCI fails to perform required function</td>
<td>CSCI performs far too many features and functions. List each feature and what can go wrong instead.</td>
</tr>
</tbody>
</table>
2.0 Prepare for the SFMEA

Common Viewpoint Mistake

Total failures rarely result from total failure of software to execute

• Common but ineffective approach - *Analyze every software CSCI against total failure of that CSCI to execute*

• Hardware engineers assume that since hardware fails when it wears out or breaks that software failures when it doesn’t execute at all
  • Sounds good-but history shows that the most serious hazards happen when the software is *running*
    • Your book has many real world examples
  • If the software doesn’t execute at all that’s likely to be identified in testing hence this is akin to a “Captain Obvious” failure mode

• The concept of a partial failure causing a total system failure is often difficult for hardware/systems engineers to grasp
2.0 Prepare for the SFMEA
Common Viewpoint Mistake
Focusing at too high or too low level a level of abstraction

Analyzing one line of code at a time has potential to miss the design and requirements related faults. Won’t work for code that’s “missing”.

Not enough coverage across the software and not enough coverage of design or software only requirements.
**2.0 Prepare for the SFMEA**

**Common Viewpoint Mistake**

Don’t mix functional and process root causes

- **Functional root cause** – defect in the software requirements, specifications, design, code, interface, usability, etc.
- **Process root cause** – the reason why the software product root causes weren’t detected or mitigated prior to system failure
- **Example** – Ariane 5 disaster 1996

<table>
<thead>
<tr>
<th>Functional root causes</th>
<th>Process related root causes</th>
</tr>
</thead>
</table>
| Data conversion from 64 to 16 bit                   | • Faulty assumption that since code didn’t change from ARIANE4 then it is OK for ARIANE5. In fact preparation sequence and workload were different from ARIANE5 than ARIANE4. ARIANE 5 could handle a heavier payload.  
  • Insufficient review of existing design against new ARIANE5 environment. |
| Conversion error unprotected (not handled)          | • Faulty assumption that since overflow didn’t happen on ARIANE4 it is “impossible” for ARIANE5.                                                                 |
|                                                     | • Since there were no controls for overflow “impossible” was wrong assessment.                                                                             |
| One size fits all defect exception handling -       | • No requirement for ARIANE5 for the protection of the unhandled conversion.                                                                                 |
| Software shut down when unhandled events detected   | • Since only requirements are tested, no tests new environment or for overflow were run in simulation.                                                     |
Contrary to popular myth it’s not feasible to conduct a software FMEA on all of the software. Even small applications have thousands of lines of code and countless combinations of inputs.

Some options:

1. **Broad and shallow**
   - Choose one or two known common failure modes and apply them throughout the mission and/or safety critical software specifications/use cases.
   - Example: There have been many problems with alternative flows and handling of faulty hardware, computations, input/output, etc. Apply “missing fault handling” to every mission and safety critical use case.

2. **Deep and narrow**
   - Choose the most critical function or use case and analyze it against every applicable software failure mode.
   - Example: A missile must be ejected so as to not hit the platform that launched it. The specifications related to the initial ejection are analyzed against faulty state, sequences, timing, data, error handling.
Software FMEA is not a “one person” activity. Inputs from software and systems engineering are required. Reliability engineers who haven’t developed software cannot perform the analysis effectively by themselves.
Prepare Software FMEA

Define scope

Tailor the SFMEA

Identify what can go wrong
Identify riskiest functions
Identify resources
Set ground rules
Define likelihood and severity
Select template and tools

For each use case, use case steps, SW specifications, interfaces, detailed design, user manuals, Installation scripts ... (as applicable based on selected view point)

Analyze applicable failure modes
Identify root causes(s) for each failure mode

Analyze failure modes and root causes

Identify local/subsystem/system failure effects
Identify controls
Identify compensating provisions
Identify corrective actions
Revise RPN

Identify consequences
Identify severity and likelihood
Mitigate

Identify equivalent failure modes
Generate a Critical Items List (CIL)

Generate CIL

Step 2

Analyze software failure modes and root causes
### 3.0 Analyze failure modes and root causes

#### Template

| Failure Mode No. | Software Item Under Consideration | Potential Failure Mode | Potential Root Cause | Potential Effect(s) of Failure | Effect Level (E) | Detection Method(s) | Occurrence Level of Failure Mode (O) | Detection Level of Failure Mode (D) | Risk Priority Number (RPN) | Software CTQ (Design Details) | CTQ Rationale | Recommended Action(s) | Responsible Individual(s) / Function(s) | Target Completion Date | Action(s) Taken | Residual Effect Level (E) | Residual Occurrence Level (O) | Residual Detection Level (D) | Residual Risk Priority Number (RPN) | System Hazard ID(s) |
|------------------|----------------------------------|------------------------|----------------------|-------------------------------|------------------|---------------------|----------------------------------------|--------------------------------------|---------------------------|--------------------------------|------------------------|----------------------|--------------------------------|---------------------------------|----------------|--------------------------|-----------------------------|-----------------------------|-----------------------------|-----------------------------|----------------|

© Softrel, LLC 2020
## 3.0 Analyze failure modes

### Summary of views, sub-views and failure modes/root causes

<table>
<thead>
<tr>
<th>Viewpoint</th>
<th>Sub-view</th>
<th>Failure modes/root causes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>What can go wrong with most or all of the specifications?</td>
<td>Missing crucially important details, overengineered, under-engineered, Missing error handling, one size fits all error handling, wrong error recovery, no recovery, missing essential timing requirements, missing fault state, missing transitions to fault state, prohibited states allowed, implied dead states</td>
</tr>
<tr>
<td></td>
<td>What can go wrong with a feature?</td>
<td></td>
</tr>
<tr>
<td></td>
<td>What can go wrong with a specific specification?</td>
<td></td>
</tr>
<tr>
<td>Interface</td>
<td>What can go wrong between two components?</td>
<td>Conflicting units of measure, type, size. Unable to handle corrupt, missing or null data.</td>
</tr>
<tr>
<td>Detailed, maintenance</td>
<td>What can go wrong with the detailed design for a module. What can go wrong when the module is changed after a baseline.</td>
<td>Algorithms themselves are faulty or are implemented faulty. Faulty logic, faulty sequences, faulty data definition, faulty error handling, faulty I/O</td>
</tr>
<tr>
<td>Usability</td>
<td>What can go wrong with the user?</td>
<td>Unnecessary fill in the blanks, faulty assumption user is always there or looking at software, overly cumbersome</td>
</tr>
</tbody>
</table>
3.0 Analyze failure modes

Efficiency is key

• Each of the viewpoints has templates which facilitate analysis at that viewpoint with most relevant and likely failure modes and root causes

  HIGHLY recommended – Convert text based specifications to flow, sequence, state transition, timing diagrams when analyzed

• MUCH easier to see the failure modes and root causes when there is a diagram or table

• In this presentation the functional viewpoint is presented
3.0 Analyze failure modes

More pictures – more efficiency

A very common specification fault

- This is a typical text based software design
  - The software shall Initialize
  - The software shall execute <Process A>
  - The software shall execute <Process B>
  - The software shall execute <Process C>
- In writing there doesn’t appear to be anything wrong
- However, when it’s illustrated as a flow it’s clear that some things are missing from specification
  - Initialization may not be successful
  - Process A may not be successful
  - Process B may not be successful
  - Process C may not be successful
  - Do previous processes have to be successful for next one to execute?
  - What should software do if any of these tasks are not successful?
3.0 Analyze failure modes

More pictures – more efficiency

Example of dead-end error handling

- Text based software design specification
  - The software shall Initialize.
  - If initialization is unsuccessful the software will make X attempts to initialize.
  - If initialization not successful after x attempts the software shall log an error.
  - The software shall execute <Process A>. If not successful it shall log an error.
  - The software shall execute <Process B>. If not successful it shall log an error.
  - The software shall execute <Process C>. If not successful it shall log an error.

- These are the problems which are easily to see in diagram form
  - Errors are logged but there is no action defined beyond that
  - As written the software will simply stop at the first error event
  - This is a very common problem within and outside of defense industry
  - It’s due to very little thinking about the failure scenarios
  - This is also an example of “one size fits all” error handling.
    - If error – log event.
    - Not applicable for all errors.
3.0 Analyze failure modes

More pictures – more efficiency

Example of order dependence in fault handling

- Text based software design specification
  - The software shall Initialize.
  - If initialization is unsuccessful the software will make X attempts to initialize.
  - If initialization not successful after x attempts the software shall log an error.
  - The software shall execute <Process A>. If not successful it shall log an error.
  - The software shall execute <Process B>. If not successful it shall log an error.
  - The software shall execute <Process C>. If not successful it shall log an error.

- These are the problems which are easier to see in a diagram
  - In this example, order doesn’t matter for process A, B or C.
  - If not stated as such, the code might be written to require sequential processing which might result in timing requirements not being met.
  - Similarly the reverse can happen if the specification is not clear.
The “Not operational” state is the missing safe state. The transitions to it are the safe state returns. If the system cannot revert back when the faulted state is entered that’s an example of no fallback or recovery. (In this example it should transition back to initialization)
3.0 Analyze failure modes
Status light failure analysis example

Example specifications

A system has a set of 3 lights modeled after a traffic light.

This status light is attached to mission critical equipment in a human-less factory. It is imperative that when the system is in a faulted state that the status is set to allow persons watching from above to send out a service technician before the fault causes a loss of product or equipment.

These are the specifications for the software:

- **SRS #1** - The software shall display a red light if at least one system failure has been detected.
- **SRS #2** - The software shall display a yellow light if there are no system failures and at least one system warning has been detected.
- **SRS #3** - The software shall display a green light if there are no system failures and no system warnings detected.
- **SRS #25** - The software shall detect all system failures as per appendix B within 2 seconds of the onset of the system failure
- **SRS #26** - The software shall detect all system warnings as per appendix B within 2 seconds of the onset of the system warning

**Informative** – Physically the light tower also has a blue light in addition to red, yellow, green. The blue light isn’t used for this feature but it used by other software features. There are 4 total light bulbs.

**Functional FMEA** – top level color display

**Detailed FMEA** – all alarms in appendix B

**Interface FMEA** – interfaces from each device to light tower
### Define Failure Definition Scoring Criteria (FDSC)

- First identify the severity of all known hazards

<table>
<thead>
<tr>
<th>Severity</th>
<th>Events</th>
<th>Immediate effect</th>
<th>Company effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catastrophic</td>
<td>There is a system failure but green light is on or no light at all</td>
<td>No service person is sent to equipment to fix system failure</td>
<td>Loss of product and/or loss of equipment. Potential loss of productivity for entire factory.</td>
</tr>
<tr>
<td>Critical</td>
<td>There is a system failure but yellow light is on</td>
<td>A service person is sent to the equipment but not as quickly as if the light displays red</td>
<td>Loss of product for several minutes.</td>
</tr>
<tr>
<td>Critical</td>
<td>There is a system warning but green light is on</td>
<td>A service person is not sent to the equipment</td>
<td>There will eventually be a failure that requires immediate attention</td>
</tr>
<tr>
<td>Moderate</td>
<td>There is no failure or warning but red light is on</td>
<td>A service person is sent to this equipment when not needed</td>
<td>Major inconvenience if it happens at all</td>
</tr>
<tr>
<td>Moderate</td>
<td>There is a system warning but red light is on</td>
<td>A service person is sent to this equipment sooner than need be</td>
<td>Major inconvenience if it happens regularly.</td>
</tr>
<tr>
<td>Moderate</td>
<td>All of the lights are on, or more than one light is on</td>
<td>A service person is sent to the equipment and doesn’t know what’s wrong</td>
<td>It can take longer to service. Major inconvenience if it happens regularly.</td>
</tr>
</tbody>
</table>

### 3.0 Analyze failure modes

**Status Light failure analysis example**
3.0 Analyze failure modes
Status light failure analysis example

Identify potential faults caused by implementation

• Brainstorm what can go wrong with implementation with light example regardless of whether the specifications are correct
  • More than one light is on at a time
  • The lights blink
  • No light is displayed
  • The software displays a red light when no fault has occurred
  • The software displays a yellow light when no warning has occurred
  • The software displays a green light when there is a warning or failure
  • The software displays the wrong light color when there are multiple failures
  • The software displays the wrong light color when there are multiple warnings
  • The software displays green after one failure is corrected even though there are multiple failures
  • The software displays green after one warning is corrected even though there are multiple warnings
  • The software takes a long time to change the light color when the status changes

• Place the above failure modes and root causes on the SFMEA template as implementation faults

• Next, let’s analyze the software design specifications for faults in the specifications themselves
3.0 Analyze failure modes

Status light failure analysis example

Analyze the specifications for most relevant software failure mode

• This is a stateful system, hence we know that faulty state management is relevant

• Brainstorm all of the ways that this system can be problematic with regards to sequences and statement management.

• Since this is a state-ful system draw a state transition table or state diagram (see next slide)

• What’s not in the design specifications?

• Refer to the below list of possible root causes

• Add any additional root causes that you identify in brainstorming

<table>
<thead>
<tr>
<th>Failure mode</th>
<th>Generic root cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faulty sequences</td>
<td>Required operations are specified in the wrong order</td>
</tr>
<tr>
<td></td>
<td>Required state transitions are incorrect or missing</td>
</tr>
<tr>
<td></td>
<td>Specification implies a dead state</td>
</tr>
<tr>
<td></td>
<td>Specification implies an orphan state</td>
</tr>
<tr>
<td></td>
<td>Prohibited transitions aren’t explicitly specified</td>
</tr>
</tbody>
</table>
3.0 Analyze failure modes
Status light failure analysis example

Visualize the specifications for greater efficiency
• This is the required system state diagram for the status light

Many of the faults are related to jammed or misaligned material. A common maintenance action is to unjam the material. This can be done without rebooting the equipment. Other maintenance actions may require a reboot of hardware. If the reboot time exceeds 15 minutes that will affect the factory production.
3.0 Analyze failure modes
Status light failure analysis example

Visualize the specifications for greater efficiency

- The state diagram on the previous page is the required state for the status light
- The written specifications are converted to a state diagram and shown on bottom right
  - The specifications specify how the lights are set, but don’t explicitly cover what happens when the warning or failure is resolved
  - Since software engineers are obligated to trace their code directly to a written specification, it can’t be assumed that their code will clear the lights when the warning or failures are cleared.
  - Notice that there is a timing requirement for the setting of lights when an error is detected but not for the resetting of the lights once error is cleared.

- This is an example of when a picture is most efficient for finding gaps in specifications
## 3.0 Analyze failure modes

### Status light failure analysis example

**Brainstorm specific root cause for faulty sequences/state transitions**

- Brainstorm this generic template against the illustration
- Identify specific root causes for the status light

<table>
<thead>
<tr>
<th>Failure mode</th>
<th>Generic root cause</th>
<th>Specific root causes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faulty sequences/state transition</td>
<td>Required operations are specified in the wrong order</td>
<td>Doesn’t appear to be relevant from state diagram</td>
</tr>
<tr>
<td></td>
<td>Required state transitions are incorrect or missing</td>
<td>The state transition from yellow to green is missing. The specifications don’t state what lights are displayed – if any – when in the initializing mode</td>
</tr>
<tr>
<td></td>
<td>Specification implies a dead state</td>
<td>Once the red light is on, it’s always on until reboot of software</td>
</tr>
<tr>
<td></td>
<td>Orphan state</td>
<td>See below</td>
</tr>
<tr>
<td></td>
<td>Prohibited transition isn’t explicitly specified.</td>
<td>In this case the blue light is required to be an “orphan”. A transition for the blue light isn’t prohibited.</td>
</tr>
</tbody>
</table>
### 3.0 Analyze failure modes

**Status light failure analysis example**

Place brainstormed root causes on SFMEA template

<table>
<thead>
<tr>
<th>No.</th>
<th>Software item under consideration</th>
<th>Design requirement (Requirement ID Tag)</th>
<th>Related Design requirement (Requirement ID Tag)</th>
<th>Software item functionality</th>
<th>Failure mode</th>
<th>Generic Root Cause</th>
<th>Potential root cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Status light tower</td>
<td>SRS 1 - The software shall display a red light if at least one system failure has been detected.</td>
<td>SRS 25 The software shall detect all system failures as per appendix B within 2 seconds of the onset of the system failure</td>
<td>The red light is on when the system is in a failed state</td>
<td>Faulty Sequences/State Transitions</td>
<td>Specifications imply a dead state</td>
<td>Dead state when transitioning from red to yellow- (there are no specifications for transitioning from failed state to warning state)</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>SRS 2 - The software shall display a yellow light if there are no system failures and at least one system warning has been detected.</td>
<td>SRS 26 The software shall detect all system warnings as per appendix B within 2 seconds of the onset of the system warning</td>
<td>The yellow light is on when the system is in a degraded state</td>
<td>Missing state transition</td>
<td>Specifications imply a dead state</td>
<td>No transition from yellow to green- (there are no specifications for transitioning from warning state to operational state)</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>SRS 1 -The software shall display a red light if at least one system failure has been detected.</td>
<td>SRS 25 The software shall detect all system failures as per appendix B within 2 seconds of the onset of the system failure</td>
<td>The red light is on when the system is in a failed state</td>
<td>Specifications imply a dead state</td>
<td>Dead state when transitioning from red to green- (there are no specifications for transitioning from failed state to clear state)</td>
<td></td>
</tr>
</tbody>
</table>
### Failure mode and root cause Section

<table>
<thead>
<tr>
<th>No.</th>
<th>Software under consideration</th>
<th>Design requirement (Requirement ID Tag)</th>
<th>Related Design requirement (Requirement ID Tag)</th>
<th>Software item functionality</th>
<th>Failure mode</th>
<th>Generic Root Cause</th>
<th>Potential root cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Status light tower</td>
<td>#1 The software shall display a red light if at least one system failure has been detected. #2 The software shall display a yellow light if there are no system failures and at least one system warning has been detected. #3 - The software shall display a green light if there are no system failures and no system warnings detected.</td>
<td>SRS_25 The software shall detect all system failures as per appendix B within 2 seconds of the onset of the system failure. SRS_26 The software shall detect all system warnings as per appendix B within 2 seconds of the onset of the system warning.</td>
<td>The red light is on when the system is in a failed state. The yellow light is on when the system is in a degraded state. The red light is on when the system is in a failed state.</td>
<td>There is no specification to explicitly prohibit changing the blue light.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.0 Analyze failure modes
Status light example

**Brainstorm faulty sequences/state transitions**

- **4 Status light tower**
  - **#1** The software shall display a red light if at least one system failure has been detected.
  - **#2** The software shall display a yellow light if there are no system failures and at least one system warning has been detected.
  - **#3** The software shall display a green light if there are no system failures and no system warnings detected.
  - **SRS_25** The software shall detect all system failures as per appendix B within 2 seconds of the onset of the system failure.
  - **SRS_26** The software shall detect all system warnings as per appendix B within 2 seconds of the onset of the system warning.
  - **Faulty Sequences/State Transitions**
    - There is no specification for what light, if any, is on when initializing at startup.
Prepare Software FMEA

Define scope

Identify local/subsystem/system failure effects
Identify compensating provisions
Identify equivalent failure modes

Identify preventive measures
Identify corrective actions
Revise RPN

Identify severity and likelihood
Identify correcting actions
Mitigate

Gather artifacts
Select ground rules
Set ground rules

Define likelihood and severity

Tailor the SFMEA

Select template and tools

For each use case, use case steps, SW specifications, interfaces, detailed design, user manuals, Installation scripts ...
(as applicable based on selected view point)
Identify what can go wrong
Identify riskiest functions
Identify resources

Analyze applicable failure modes

Identify boundary
Identify what can go wrong
Select View points
Gather artifacts

Identify severity and likelihood

Identify root causes(s) for each failure mode
Analyze failure modes and root causes

Identify consequences

Identify boundary
Identify what can go wrong
Select View points
Gather artifacts
Identify riskiest functions
Identify resources
Set ground rules
Define likelihood and severity
Select template and tools

Step 3
Identify Consequences
### 4.0 Analyze consequences

**Template**

<table>
<thead>
<tr>
<th>Failure Mode No.</th>
<th>Software Item Under Consideration</th>
<th>Software Item Functionality</th>
<th>Design Requirement (Requirement ID tag)</th>
<th>Potential Failure Mode</th>
<th>Potential Root Cause</th>
<th>Potential Effect(s) of Failure</th>
<th>Effect Level (E)</th>
<th>Detection Method(s)</th>
<th>Occurrence Level of Failure Mode (O)</th>
<th>Detection Level of Failure Mode (D)</th>
<th>Risk Priority Number (RPN)</th>
<th>Software CTQ (Design Details)</th>
<th>CTQ Rationale</th>
<th>Recommended Action(s)</th>
<th>Responsible Individual(s) / Function(s)</th>
<th>Target Completion Date</th>
<th>Action(s) Taken</th>
<th>Residual Effect Level (E)</th>
<th>Residual Occurrence Level (O)</th>
<th>Residual Detection Level (D)</th>
<th>Residual Risk Priority Number (RPN)</th>
<th>System Hazard ID(s)</th>
</tr>
</thead>
</table>

© Softrel, LLC 2020
### 4.0 Analyze consequences

Analyze each row of the SFMEA for effects using the defined FDSC

- Here is the initial hazards list for the status light.

<table>
<thead>
<tr>
<th>Severity</th>
<th>Events</th>
<th>Immediate effect</th>
<th>Company effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catastrophic</td>
<td>There is a system failure but green light is on or no light at all</td>
<td>No service person is sent to equipment to fix system failure</td>
<td>Loss of product and/or loss of equipment. Potential loss of productivity for entire factory.</td>
</tr>
<tr>
<td>Critical</td>
<td>There is a system failure but yellow light is on</td>
<td>A service person is sent to the equipment but not as quickly as if the light displays red</td>
<td>Loss of product for several minutes.</td>
</tr>
<tr>
<td>Critical</td>
<td>There is a system warning but green light is on or no light at all</td>
<td>A service person is not sent to the equipment</td>
<td>There will eventually be a failure that requires immediate attention</td>
</tr>
<tr>
<td>Major</td>
<td>There is no failure or warning but red light is on</td>
<td>A service person is sent to this equipment immediately when not needed</td>
<td>Major inconvenience if it happens at all</td>
</tr>
<tr>
<td>Minor</td>
<td>There is no system warning but yellow light is on</td>
<td>A service person is sent to this equipment when not needed</td>
<td>Major inconvenience if it happens regularly.</td>
</tr>
<tr>
<td>Minor</td>
<td>There is a system warning but red light is on</td>
<td>A service person is sent to this equipment sooner than needed</td>
<td>Major inconvenience if it happens regularly.</td>
</tr>
<tr>
<td>Minor</td>
<td>All of the lights are on, or more than one light is on</td>
<td>A service person is sent to the equipment and doesn’t know what’s wrong</td>
<td>It can take longer to service. Major inconvenience if it happens regularly.</td>
</tr>
</tbody>
</table>
### Status light failure analysis example

<table>
<thead>
<tr>
<th>No.</th>
<th>Design requirement (Requirement ID tag)</th>
<th>Potential root cause</th>
<th>Potential effects of failure</th>
<th>Potential effects of failure</th>
<th>Effect level (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The software shall display a red light if at least one system failure has been detected.</td>
<td>Dead state when transitioning from red to yellow- (there are no specifications for transitioning from failed state to warning state)</td>
<td>There is no failure or warning but red light is on</td>
<td>A service person is sent immediately to this equipment when not needed</td>
<td>Major</td>
</tr>
<tr>
<td>2</td>
<td>The software shall display a yellow light if there are no system failures and at least one system warning has been detected.</td>
<td>No transition from yellow to green- (there are no specifications for transitioning from warning state to operational state)</td>
<td>There is no system warning but yellow light is on</td>
<td>A service person is sent to this equipment when not needed</td>
<td>Minor</td>
</tr>
<tr>
<td>3</td>
<td>The software shall display a red light if at least one system failure has been detected.</td>
<td>Dead state when transitioning from red to green- (there are no specifications for transitioning from failed state to clear state)</td>
<td>There is no failure or warning but red light is on</td>
<td>A service person is sent immediately to this equipment when not needed</td>
<td>Major</td>
</tr>
</tbody>
</table>
# Identify severity for those that don’t map to initial hazards list

<table>
<thead>
<tr>
<th>No.</th>
<th>Design requirement (Requirement ID tag)</th>
<th>Potential root cause</th>
<th>Potential effects of failure</th>
<th>Potential effects of failure</th>
<th>Effect level (E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>The software shall display a red light if at least one system failure has been detected. The software shall display a yellow light if there are no system failures and at least one system warning has been detected.</td>
<td>There is no specification to explicitly prohibit changing the blue light.</td>
<td>Blue light is changed when it shouldn’t be</td>
<td>Process status isn’t known to factory. Potential for mis-processing.</td>
<td>Critical</td>
</tr>
<tr>
<td>5</td>
<td>The software shall display a red light if at least one system failure has been detected.</td>
<td>There is no specification for what color, if any, is displayed at the initial state.</td>
<td>If the equipment is in failed state, factory can’t see the status at all on startup</td>
<td>Delay in sending service person to equipment</td>
<td>Critical</td>
</tr>
</tbody>
</table>

These two hazards weren’t covered in the original FDSC.

- If the blue light isn’t correct it will effect whether the factory knows the state of the material being processed by the equipment. There could be an undetected misprocess as worst case.
- If there is no light when equipment starts up and there will be a delay in sending service person to equipment. That could lead to a backup in the factory.
4.0 Analyze consequences

Assess Likelihood

• Likelihood is assessed AFTER severity is assessed to ensure that catastrophic failure modes aren’t prematurely pruned from the analysis

• Likelihood is a function of four things
  • **Existence likelihood** – likelihood that the root cause exists in the software
  • **Manifestation likelihood** - How likely are the conditions that cause the root cause to manifest into a failure
  • Whether or not the failure mode/root cause is controlled
  • **How detectable the root cause** is during the development process

• Final likelihood for risk matrix =

  Existence Likelihood * Manifestation likelihood * Control

• Detectability will be assessed separately on the risk matrix
4.0 Analyze consequences

Assess likelihood that root cause exists in the software

1. First, determine if it’s known for sure that the failure mode/root cause does in fact exist in the specifications or code.
2. If it the specification or code is itself deficient then likelihood of existence is set to “high”
3. Otherwise, the below table is used to assess likelihood of existence

<table>
<thead>
<tr>
<th>Likelihood of existence</th>
<th>Affected software design</th>
<th>Past history</th>
<th>Domain expertise</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Very complex or problematic</td>
<td>Has happened in every past system or is known to be present.</td>
<td>No experience with this feature or product</td>
</tr>
<tr>
<td>Moderate</td>
<td>Average complexity</td>
<td>Has happened at least once in the past</td>
<td>Some experience with this feature or product</td>
</tr>
<tr>
<td>Low</td>
<td>Very simple, not problematic</td>
<td>Hasn’t happened in the past and there’s no reason to believe it will happen on this system.</td>
<td>Significant experience with this feature or product</td>
</tr>
</tbody>
</table>
1. Firstly, determine if the failure mode/root cause is contingent upon another failure – say a failure in the hardware. If the particular root cause will only manifest itself when there is another failure then it’s likelihood can be no worse than the likelihood of the related failure.

2. Secondly, determine if the failure mode/root cause could effect multiple installed sites. If the root cause affects a feature used by most customers then it’s frequency in operation is greater than a root cause in a feature not frequently used.

The manifestation likelihood is set to no worse than the likelihood of the root cause being related to another failure.
4.0 Analyze consequences

Identify controls

• Examples
  • BIT/Health monitoring software that monitors other software
  • Interlocks can ensure that if the software provides conflicting commands that the safest command is always executed.

• Review each row in the SFMEA, if there are any known controls for this failure mode and root cause, identify them.

<table>
<thead>
<tr>
<th>Controls</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Multiple controls for the root cause</td>
</tr>
<tr>
<td>Moderate</td>
<td>One control</td>
</tr>
<tr>
<td>Low</td>
<td>No controls</td>
</tr>
</tbody>
</table>
The detectability of the failure mode/root cause depends on the operating conditions required for it to be manifested. If the root cause is visible with any operating condition then it’s almost certain to be detected in testing. If the root cause is visible under an operating condition that cannot be reproduced in a test environment then it’s detectability is improbable.

### Analyze Detectability for Each SFMEA Row

<table>
<thead>
<tr>
<th>Likelihood</th>
<th>Detection level</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>5-IMPROBABLE</strong></td>
<td>The failure mode can’t be reproduced in a development or test environment.</td>
</tr>
<tr>
<td><strong>4-LOW PROBABILITY</strong></td>
<td>The failure mode is visible and detectable only with fault injection (faulty hardware, unexpected inputs, etc.)</td>
</tr>
<tr>
<td><strong>3-MODERATE PROBABILITY</strong></td>
<td>The failure mode is visible and detectable with off nominal testing (pressing cancel buttons, entering invalid data)</td>
</tr>
<tr>
<td><strong>2-HIGH PROBABILITY</strong></td>
<td>The failure mode is visible and detectable with requirements based testing</td>
</tr>
<tr>
<td><strong>1-ALMOST CERTAIN</strong></td>
<td>The failure mode is visible and detectable under any set of conditions</td>
</tr>
</tbody>
</table>
4.0 Analyze consequences

Apply the Risk Matrix

- Risk Matrices are typically company/project specific
- Here is an example

<table>
<thead>
<tr>
<th>Likelihood</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 – Catastrophic</td>
</tr>
<tr>
<td>5 – Almost certain</td>
<td>Mitigate</td>
</tr>
<tr>
<td></td>
<td>4 – Critical</td>
</tr>
<tr>
<td>4- High</td>
<td>Mitigate</td>
</tr>
<tr>
<td>3 – Moderate</td>
<td>Mitigate</td>
</tr>
<tr>
<td>2 - Low</td>
<td>Mitigation not required</td>
</tr>
<tr>
<td>1 – Improbable</td>
<td>Mitigation not required</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Severity</th>
<th>3 – Major</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 – Almost certain</td>
<td>Mitigate</td>
</tr>
<tr>
<td>4- High</td>
<td>Mitigate</td>
</tr>
<tr>
<td>3 – Moderate</td>
<td>Mitigate</td>
</tr>
<tr>
<td>2 - Low</td>
<td>Mitigation not required</td>
</tr>
<tr>
<td>1 – Improbable</td>
<td>Mitigation not required</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Severity</th>
<th>2 – Minor</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 – Almost certain</td>
<td>Mitigate</td>
</tr>
<tr>
<td>4- High</td>
<td>Mitigate</td>
</tr>
<tr>
<td>3 – Moderate</td>
<td>Mitigate</td>
</tr>
<tr>
<td>2 - Low</td>
<td>Mitigation not required</td>
</tr>
<tr>
<td>1 – Improbable</td>
<td>Mitigation not required</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Severity</th>
<th>1 - Negligible</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 – Almost certain</td>
<td>Mitigate</td>
</tr>
<tr>
<td>4- High</td>
<td>Mitigate</td>
</tr>
<tr>
<td>3 – Moderate</td>
<td>Mitigate</td>
</tr>
<tr>
<td>2 - Low</td>
<td>Mitigation not required</td>
</tr>
<tr>
<td>1 – Improbable</td>
<td>Mitigation not required</td>
</tr>
</tbody>
</table>

- Likelihood:
  - 5 – Almost certain: Almost certain
  - 4- High: High
  - 3 – Moderate: Moderate
  - 2 - Low: Low
  - 1 – Improbable: Improbable

- Severity:
  - 5 – Catastrophic: Catastrophic
  - 4 – Critical: Critical
  - 3 – Major: Major
  - 2 – Minor: Minor
  - 1 - Negligible: Negligible
## 4.0 Analyze consequences

**Status light failure analysis example**

### Map the software root causes to the known hazards

<table>
<thead>
<tr>
<th>No.</th>
<th>Design requirement ID Tag</th>
<th>Potential root cause</th>
<th>Potential effects of failure</th>
<th>Effect level (E)</th>
<th>Occurrence level of failure mode (O)</th>
<th>Detection level of failure (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#1</td>
<td>Dead state when transitioning from red to green - (there are no specifications for transitioning from failed state to clear state)</td>
<td>A service person is sent immediately to this equipment when not needed</td>
<td>3 - Major</td>
<td>5 - None</td>
<td>5 - This root cause is guaranteed because specification is incorrect</td>
</tr>
</tbody>
</table>

- Likelihood is determined by the specific root cause and the control for that root cause – not the effect.
- This specific root cause is guaranteed to effect the design because the specification is insufficient. So the existence likelihood = 5.
- This root cause is directly related to a hardware failure so the manifestation likelihood is assessed at 3.
- There is no control for the root cause.
- The average likelihood = Average(5, 5, 3) = 4.33
- Since only requirements based testing is planned, and this is a missing specification there is virtually no chance it will be found during testing so detectability is also assessed at 5. RPN = 3*4*5 = 60.
### 4.0 Analyze consequences
Status light failure analysis example

**Map the software root causes to the known hazards**

<table>
<thead>
<tr>
<th>No.</th>
<th>Design requirement ID tag</th>
<th>Potential root cause</th>
<th>Potential effects of failure</th>
<th>Effect level (E)</th>
<th>Occurrence level of failure mode (O)</th>
<th>Manifestation likelihood</th>
<th>Controls</th>
<th>Detection level of failure (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>SRS #1,#2,#3</td>
<td>There is no specification to explicitly prohibit changing the blue light.</td>
<td>Process status isn’t known to factory – Potential for misprocessing</td>
<td>4- Critical</td>
<td>5. None</td>
<td>5- Likely</td>
<td>5- Likely</td>
<td>5. This won’t be detected in testing because only requirements are tested</td>
</tr>
</tbody>
</table>

- The specification doesn’t prohibit the setting of the blue light by the status light feature. So, the question is whether the code has been designed to prevent this.
- If this code does allow the prohibited transition it won’t be found in testing because of exclusive requirements based testing.
- In the past, there has been state related problems.
- The software team has average experience with the application.
- This root cause is not related to a hardware failure.
- There is no control for the root cause. However, there are controls that can be mitigated.
- Hence likelihood is = Average(5,3,5) = 4.33 which is rounded down to 4.
- Risk Product Number (RPN) = 4*4*5= 80
Prepare Software FMEA

Define scope

Prepare Software FMEA

- Identify boundary
- Identify what can go wrong
- Select View points
- Gather artifacts
- Identify riskiest functions
- Identify resources
- Set ground rules
- Define likelihood and severity
- Select template and tools

Tailor the SFMEA

Identify
- boundary
- resources
- equivalent failure modes
- consequences
- severity and likelihood
- corrective actions
- compensating provisions

Mitigate

Mitigate

For each use case, use case steps, SW specifications, interfaces, detailed design, user manuals, Installation scripts ... (as applicable based on selected view point)

Analyze applicable failure modes

Identify root causes(s) for each failure mode

Analyze failure modes and root causes

Identify local/subsystem/system failure effects

Identify controls

Identify severity and likelihood

Identify consequences

Identify riskiest functions

Identify corrective actions

Revise RPN

Identify compensating provisions

Identify equivalent failure modes

Generate a Critical Items List (CIL)

Generate CIL

Step 4

Identify Mitigation
### 5.0 Identify Mitigation

**Template**

| Failure Mode No. | Software Item Under Consideration | Software Item Functionality | Design Requirement (Requirement ID tag) | Potential Failure Mode | Potential Root Cause | Potential Effect(s) of Failure | Effect Level (E) | Detection Method(s) | Occurrence Level of Failure Mode (O) | Detection Level of Failure Mode (D) | Risk Priority Number (RPN) | Software CTQ (Design Details) | CTQ Rationale | Recommended Action(s) | Responsible Individual(s) / Function(s) | Target Completion Date | Action(s) Taken | Residual Effect Level (E) | Residual Occurrence Level (O) | Residual Detection Level (D) | Residual Risk Priority Number (RPN) | System Hazard ID(s) |
Identify Corrective Actions

- Corrective action is from the development standpoint and will depend on type of FMEA being performed
  - Functional FMEA - corrective action may include changing the specifications
  - Interface and detailed FMEAs - corrective action may include changing the design, code to correct the failure mode
  - Process FMEA - corrective action may be the execution of a particular practice or avoidance of a particular obstacle

- Examples of corrective actions that don’t apply to software
  - Replacing the software unit with an identical failed unit
  - Operator repair of the software unit on site
5.0 Identify Mitigation

Revise Risk Product Number (RPN)

• Adjust the RPN based on the assumption that the corrective action is employed
• Don’t override the original RPN assessment
• Likelihood will change if the problem is eliminated
• Severity will change only if the problem is downgraded
• Detectability will change if the failure mode is reviewed in design, code or fault injection test procedures are developed.
### 5.0 Identify Mitigation

**Status light failure analysis example**

<table>
<thead>
<tr>
<th>No.</th>
<th>Potential root cause</th>
<th>Potential effects of failure</th>
<th>Effect level</th>
<th>Detection level</th>
<th>Occurrence level</th>
<th>RPN</th>
<th>Recommended Action(s)</th>
<th>Residual RPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>There is no specification to explicitly prohibit changing the blue light.</td>
<td>Process status isn’t known to factory – Potential for mis-processing</td>
<td>4. Critical</td>
<td>5. Won’t be found in any test</td>
<td>4. Likely to highly likely</td>
<td>4<em>5</em>4=90</td>
<td>Add a specification statement prohibited any changes of blue light by light status code. Review code to ensure that status light never changes the blue light display. Monitor during endurance tests to ensure the blue light is never changed by the status light code – only the code that is allowed to change the blue light.</td>
<td>RPN =4= 4<em>1</em>1 since root cause is mitigated and the code review is in place to detect it</td>
</tr>
</tbody>
</table>

The corrective action is to add a specification for the prohibited state transition, review the code to ensure there isn’t a transition for the blue light in the status code (only the feature that is supposed to change the blue light). Prohibited transitions are difficult to test so it would need to be monitored over the testing period. The original RPN was 4*5*4. With the corrective actions, the root cause is mitigated so the likelihood = 1 and detectability = 1. The adjusted RPN is then 4*1*1=4.
Prepare Software FMEA

Define scope

Tailor the SFMEA

Identify boundary
Identify what can go wrong
Select view points
Gather artifacts
Identify riskiest functions

Identify resources
Set ground rules
Define likelihood and severity
Select template and tools

Identify
boundary

Identify
what can
go wrong

Select
view points

Gather
artifacts

Identify
riskiest
functions

Identify
resources

Set ground
rules

Define
likelihood
and severity

Select
template
and tools

For each use case, use case steps, SW specifications, interfaces, detailed design, user manuals, Installation scripts ... (as applicable based on selected view point)

Analyze applicable failure modes

Identify root causes(s) for each failure mode
Analyze failure modes and root causes

Identify local/subsystem/system failure effects

Identify preventive measures

Identify compensating provisions

Identify preventive measures

Identify corrective actions

Revise RPN

Identify severity and likelihood

Identify compensating provisions

Identify corrective actions

Revise RPN

Mitigate

Identify consequences

Identify consequences

Identify consequences

Generate a Critical Items List (CIL)

Generate CIL

Step 5

Generate a Critical Items List (CIL)
6.0 Generate a Critical Items List (CIL)

Identify Equivalent Failure Modes

• Two failure modes are equivalent if the effect, severity, likelihood and corrective action are the same
• During this step, identify the failure modes that are equivalent to each other so as to consolidate the corrective actions
• Don’t change the worksheet, just do the consolidation in a separate worksheet
6.0 Generate a Critical Items List (CIL)

Status light failure analysis example

These are the corrective actions to resolve every identified failure root cause of the status light

1. Add transitions from red to yellow, red to green, yellow to green to specification and test plan
2. Add prohibited transitions of blue light from status light code to specifications, code review and test plan.
3. Add specification for what lights should do upon initialization and test plan.

The corrective actions could all be made at one time without regard for RPN - or they could be implemented selectively to address only the most critical RPN root causes.

If the code hasn’t been written yet, it’s often most efficient to simply fix all defective requirements.

When the code is already written changing the defective specifications poses a bigger risk from both a schedule and software standpoint. Hence, the corrective actions are implemented based on risk.
<table>
<thead>
<tr>
<th>No.</th>
<th>Design requirement ID</th>
<th>Potential root cause</th>
<th>Potential effects of failure</th>
<th>Effect level (E)</th>
<th>Detection level (D)</th>
<th>Occurrence level (O)</th>
<th>RPN</th>
<th>Recommended Action(s)</th>
<th>Residual occurrence level (O)</th>
<th>Residual effect level (E)</th>
<th>Residual RPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The software shall display a red light if at least one system failure has been detected. The software shall display a yellow light if there are no system failures and at least one system warning has been detected.</td>
<td>There is no specification to explicitly prohibit changing the blue light. Potential for mis-processing.</td>
<td>Process status isn't known to factory.</td>
<td>4. Critical</td>
<td>5. Won't be found in any test</td>
<td>4-Likely to highly likely</td>
<td>90</td>
<td>Add specification for prohibited state transition, review the code to ensure there isn't one, monitor during testing</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>The software shall display a red light if at least one system failure has been detected.</td>
<td>There is no specification for what color, if any, is displayed at the initial state.</td>
<td>Delay in sending service person to equipment</td>
<td>4. Critical</td>
<td>3-Visible with any off nominal testing</td>
<td>3- Likely</td>
<td>36</td>
<td>Add specification for what light is on, if any, during initialization</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>The software shall display a red light if at least one system failure has been detected.</td>
<td>Dead state when transitioning from red to green- (there are no specifications for transitioning from failed state to clear state)</td>
<td>A service person is sent immediately to this equipment when not needed</td>
<td>3. Major</td>
<td>4 – Visible only with fault injection testing</td>
<td>3- Likely</td>
<td>36</td>
<td>Change specifications to add transition from red to green, add transition to test plan</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>The software shall display a yellow light if there are no system failures and at least one system warning has been detected.</td>
<td>No transition from yellow to green- (there are no specifications for transitioning from warning state to operational state)</td>
<td>A service person is sent to this equipment when not needed</td>
<td>2. Minor</td>
<td>4 – Visible only with fault injection testing</td>
<td>3- Likely</td>
<td>24</td>
<td>Change specifications to add transition from yellow to green, add transition to test plan</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>The software shall display a red light if at least one system failure has been detected.</td>
<td>Dead state when transitioning from red to yellow- (there are no specifications for transitioning from failed state to warning state)</td>
<td>A service person is sent immediately to this equipment when not needed</td>
<td>3. Major</td>
<td>4 – Visible only with fault injection testing</td>
<td>3- Likely</td>
<td>36</td>
<td>Change specifications to add transition from red to yellow, add transition to test plan</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

6.0 Generate a Critical Items List (CIL)
6.0 Generate a Critical Items List (CIL)

The before and after risk matrix is presented to management

<table>
<thead>
<tr>
<th>Severity</th>
<th>Likelihood</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 – Very likely</td>
</tr>
<tr>
<td>5 – Catastrophic</td>
<td></td>
</tr>
<tr>
<td>4 – Critical</td>
<td></td>
</tr>
<tr>
<td>3 – High</td>
<td>#2</td>
</tr>
<tr>
<td>2 – Moderate</td>
<td></td>
</tr>
<tr>
<td>1 – Negligible</td>
<td></td>
</tr>
</tbody>
</table>

In the below table, the high RPN items are mitigated

<table>
<thead>
<tr>
<th>Severity</th>
<th>Likelihood</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 – Very likely</td>
</tr>
<tr>
<td>5 – Catastrophic</td>
<td></td>
</tr>
<tr>
<td>4 – Critical</td>
<td></td>
</tr>
<tr>
<td>3 – High</td>
<td></td>
</tr>
<tr>
<td>2 – Moderate</td>
<td></td>
</tr>
<tr>
<td>1 – Negligible</td>
<td></td>
</tr>
</tbody>
</table>
What you learned

• How to prepare for the software FMEA to minimize cost and maximize effectiveness
• How to get in the right mindset
• How to analyze the failure modes that apply to the entire software system
• How to analyze the failure modes that apply to a feature
• How to analyze the failure modes that apply to a specific software specification
• How to assess the consequences of each failure mode
• How to assess the mitigations of each failure mode
• How to track each failure mode to closure
## More information

| Software failure modes effects analysis class | Online self guided training  
|                                           | Online instructor led training  
|                                           | On site training  
|                                           | Open session training  
|                                           | [http://missionreadysoftware.com/training](http://missionreadysoftware.com/training)  

**Effective Application of Software Failure Modes Effects Analysis**
References


• Mil-Std 1629A Procedures for Performing a Failure Mode, Effects and Criticality Analysis, November 24, 1980.


References for famous software related failures from history

  - Overview of the DART Mishap Investigation Results For Public Release
  - [http://www.nasa.gov/pdf/148072main_DART_mishap_overview.pdf](http://www.nasa.gov/pdf/148072main_DART_mishap_overview.pdf)


- [DRUM] Dan Stockman, A Single Zero Turns Training to Tragedy-City built software linked to deadly ‘02 Army friendly fire, May 18, 2008


  - Intel’s discussion of the Pentium bug can be found here: [http://www.intel.com/support/processors/pentium/sb/CS-013007.htm](http://www.intel.com/support/processors/pentium/sb/CS-013007.htm)
  - The web site for Dr. Nicely (the person who originally detected the defect):
    - [http://www.trnicely.net/pentbug/pentbug.html](http://www.trnicely.net/pentbug/pentbug.html)
    - [http://www.apnewsarchive.com/1995/Intel-Takes-$475-Million-Charge-To-Replace-Flawed-Pentium-Net-Declines/id-853864f8f1a74457adf39ab272181e08](http://www.apnewsarchive.com/1995/Intel-Takes-$475-Million-Charge-To-Replace-Flawed-Pentium-Net-Declines/id-853864f8f1a74457adf39ab272181e08)
References for famous software related failures from history

  • The NTSB report can be found here: www.airdisaster.com/reports/ntsb/AARoo-01.pdf
  • http://nssdc.gsfc.nasa.gov/nmc/spacecraftDisplay.do?id=MARIN1
• [MARS] Douglass Isbell, Mary Hardin, Joan Underwood, “Mars Climate Orbiter Team Finds Likely Cause of Loss, Sept. 30, 1999”
  • http://sunnyday.mit.edu/accidents/MCO_report.pdf
  • http://www2.gwu.edu/~nsarchiv/nukevault/ebb371/
  • http://www-pub.iaea.org/MTCD/publications/PDF/Pub1114_scr.pdf
  • http://www.fda.gov/Radiation-EmittingProducts/RadiationSafety/AlertsandNotices/ucm116533.htm
  • http://www.russianspaceweb.com/phobos.html
  • http://nssdc.gsfc.nasa.gov/planetary/phobos.html
References for famous software related failures from history


  • [http://www.washingtonpost.com/wp-dyn/content/article/2007/04/12/AR2007041202061.html](http://www.washingtonpost.com/wp-dyn/content/article/2007/04/12/AR2007041202061.html)


References for famous software related failures from history

- **[SPIRIT]** Ron Wilson, “The trouble with Rover is revealed”, EE Times, 2/20/2004.
## SFMEA guidance

<table>
<thead>
<tr>
<th>Guidance</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mil-Std 1629A Procedures for Performing a Failure Mode, Effects and Criticality Analysis, November 24, 1980. Cancelled on 8/1998.</td>
<td>Defines how FMEAs are performed but it doesn’t discuss software components</td>
</tr>
<tr>
<td>MIL-HDBK-338B, Military Handbook: Electronic Reliability Design Handbook, October 1, 1998.</td>
<td>Adapted in 1988 to apply to software. However, the guidance provides only a few failure modes and a limited example. There is no discussion of the software related viewpoints.</td>
</tr>
<tr>
<td>“SAE ARP 5580 Recommended Failure Modes and Effects Analysis (FMEA) Practices for Non-Automobile Applications”, July, 2001, Society of Automotive Engineers.</td>
<td>Introduced the concepts of the various software viewpoints. Introduced a few failure modes but examples and guidance is limited.</td>
</tr>
</tbody>
</table>