# Verifying the Completeness of Test

Herbert Hecht,
SoHaR Incorporated
5731 W. Slauson Ave.
Culver City, CA 90230
herb@sohar.com

***Abstract*** *– Testing of a software product can be claimed to be complete when these criteria have been satisfied: all elements that can fail must have been tested, and all test outcomes must be in agreement with the expected failure effects. This paper addresses how these criteria can be met with particular reference to critical software intensive systems such as aircraft or plant control systems.*

***Keywords*** *– Software test, software verification, software FMEA, model based development*

## I. INTRODUCTION

Practically all current complex systems are heavily dependent on software. Where these systems perform or are involved with safety or economically critical functions, testing of the software by itself or as part of the system is an essential step in determining that they suitable for the intended application. In many cases this determination is required by a government agency, e. g., the FAA for aircraft systems, the Nuclear Regulatory Commission for nuclear power plants, and the FDA for medical devices as well as for process control of medications. In spite of these safeguards there have been incidents in which inadequate software testing took lives, caused very large economic losses, and exposed people to extremely scary experiences. A few examples:

- THERAC-25: massive overdoses of radiation as a result of not suppressing operator inputs while magnets were being repositioned resulting in deaths and serious injuries [1].
- Ariane 5, Flight 501: The launch vehicle self-destructed after 37 seconds due to faulty exception handling (and disabling language inherent exception handling for efficiency reasons) and failure to deal with simultaneous shut-down of both inertial navigation systems [2]
- Malaysian Airlines Boeing 777-200: On August 1, 2005 the Perth (Australia) to Kuala Lumpur flight experienced sudden pitch-up attitude change, leading to a near stall. Pilot action averted disaster and permitted the aircraft to return to Perth. The incident was traced to inability of the software to deal with an accelerometer failure [3].

In all of these examples the software had undergone extensive testing but it had not been tested under the conditions that caused the in-service failure. In other words, the testing had been incomplete but there had been no objective measure to indicate this. Another common feature of these examples is that the failure occurred under exception conditions: in the Therac incident is was the operator entering data while magnets were being repositioned, in the Ariane it was higher than expected acceleration, and in the Boeing 777 it was failure of an instrument. The centrality of faulty or missing exception handling as a cause of software failures has also been a finding of a review of software testing as part of the Y2K effort [4]: "The main line software code usually does its job. Breakdowns typically occur when the software exception code does not properly handle abnormal input or environmental conditions – or when an interface does not respond in the anticipated or desired manner." In the body of this paper we therefore investigate the reasons for inadequate testing under exception conditions and what can be done to compensate for these difficulties.

## II. FAILURE MODES AND EFFECTS ANALYSIS

A frequently employed strategy for failure prevention in critical systems is to use fault tree analysis to identify the areas in which critical failures can arise, and then to use failure modes and effects analysis (FMEA) to establish the detailed failure modes that can cause undesirable top level events. Testing then concentrates on the failure modes thus identified. This procedure is very explicit in the SAE guidelines for certification and safety assessment of airborne systems [5] but it is also implicit in other widely used safety standards [6, 7]. The starting point for a hardware FMEA is the bill of materials (BOM) that lists all components of a given assembly or higher level item. Each failure mode of each pertinent item is then entered as a row in the FMEA worksheet as shown in Figure 1. The failure modes for each item are typically obtained from a library that is part of the FMEA program. Thus, for a resistor the pertinent failure modes will be entered as open, short, and change in value. The analyst can modify this selection. In an analysis of electrical failures, fasteners and other non-electrical items are usually ignored.

FAILURE MODE AND EFFECTS ANALYSIS

SYSTEM_____ DATE_____
INDENTURE LEVEL _____ SHEET_____ OF_____
REFERENCE DRAWING_____ COMPILED BY_____
MISSION_____ APPROVED BY_____

| IDENTIFICATION NUMBER | ITEM/FUNCTIONAL IDENTIFICATION (NOMENCLATURE) | FUNCTION | FAILURE MODES AND CAUSES | MISSION PHASE/ OPERATIONAL MODE | FAILURE EFFECTS | | | SEVERITY CLASS | COMPENSATING PROVISIONS | FAILURE DETECTION METHOD | REMARKS |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | LOCAL EFFECTS | NEXT HIGHER LEVEL | END EFFECTS | | | | |
| | | | | | | | | | | | |

**Figure 2 Typical FMEA Worksheet**

Once the failure modes have been entered the analyst determines failure effects at the local, next higher and system level (end effects). The local effects are usually assessed at the output of a circuit card or equivalent assembly, and the next higher level is typically a line replaceable unit (LRU). At the system level the worst effects should be consistent with those used in the fault tree analysis. The severity classification is conventionally designated by Roman numerals, with I representing a catastrophic outcome, II total loss of mission, III partial loss of mission, etc.

The failure detection method represents a very important entry – failures that are not detected (or not always detected) can be an indirect cause of catastrophic outcomes even if the immediate effect is much more moderate. Sometimes several means of detection may be provided, e. g., an over-temperature condition may be detected by a local sensor and by a higher unit level sensor. The column labeled Compensating Provisions is used to list existing means of overcoming failure effects, such as redundancy, work-arounds, or safe modes. The severity is assessed based on availability of the compensating provisions but the remarks column should state consequences if these should fail.

When all items and failure modes have been entered and analyzed the FMEA is complete. A test program can then be developed to verify for each failure mode (i) the detection method, (ii) the failure effect, and (iii) the adequacy of the compensation or mitigation provisions. The production testing for failure effects can be considered complete when it has been shown that none of the failure effects listed in the FMEA worksheets are present. In developmental testing failure modes may have to be simulated to verify that they cause the listed failure effects and no others.

## III. SOFTWARE FMEA

Because the FMEA has such a central role in the acceptance of hardware systems for critical applications, there have been several suggestions for adapting the hardware oriented FMEA format for software. One obvious difficulty encountered in this quest was the lack of an equivalent to the BOM in software. What should be the criterion for row entries in a software FMEA worksheet? The initial approach was to focus on software "functions" [8, 9, 10]. Also, an alternative was proposed to consider specific effects due to faulty processing of one input variable at a time [11].

The main disadvantage of the functional approach is that function boundaries are not formally defined. In previous work we have shown that exception handling is a very frequent cause of failures [12], and not every analyst will consider an exception handling routine as a function. Thus the visibility over exception handling tends to get lost in functional partitioning. And because the need for exception handling frequently arises from conditions on more than one variable, the alternative approach has obvious limitations as well.

The advent of model-based software development has opened new vistas, not only for a more systematic and better documented way of creating programs, but also because it provides building blocks for software that are analogous to those for hardware in a BOM. In UML these building blocks are referred to as *methods* [13] while in Simulink and other contemporary modeling tool they are called *blocks* [14]. This feature permits an approach to software FMEA that is compatible with that used for hardware and facilitates the generation of a system FMEA and a test program based on that. In the current context we concentrate on the software portion and show how substantial parts of it can be automated.

## IV. COMPUTER AIDED SOFTWARE FMEA

The MOCET tool developed by SoHaR under AFRL sponsorship permits largely automated generation of a software FMEA. The essential steps of this process are

a. Parsing the model file to obtain a formatted list of the model blocks

b. Selection of applicable failure modes and associated local effects from a context sensitive menu

c. Selection of higher levels for evaluation of the propagation of local effects and assignment of severity classification at the system level (with menu prompts)

d. Identification of detection and mitigation provisions

In the following, the implementation of each of these steps will be discussed for an example of an aircraft longitudinal control system. The purpose of this system is to let the aircraft follow an externally generated vertical acceleration command. The Simulink block model for this system is shown in figure 2. The input command is designated Nz_cmd in the figure and the actual aircraft acceleration as well as other pertinent parameters are fed into the block as Pitch FB (input 2). An excerpt from the text file that implements this model is shown in figure 3 and the FMEA structure generated by MOCET is shown in figure 4. The format shown in the latter figure is that of a BOM (step a, above) and it is the crucial step in arriving at a software FMEA that can be called equivalent to that for hardware.

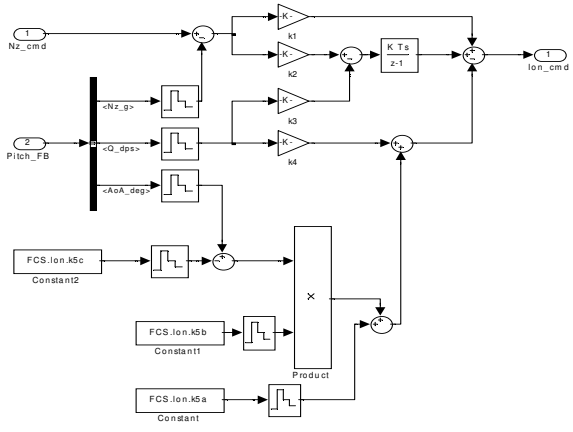


**Figure 2 Longitudinal Flight Control System**

In figure 2 it can be seen that there are several repetitively used symbols, extending the equivalence with a hardware schematic that typically contains several resistor, diode capacitor, etc. symbols. A competent hardware analyst knows that there are specific failure modes associated with each symbol (component type), and similarly the Simulink symbol (block) determines the failure modes that need to be investigated. This knowledge has been incorporated in the MOCET library that can display pertinent failure modes for many frequently used blocks. An example for continuously varying signals is shown in figure 5. For discrete signals a typical selection includes spurious switching, failure to change when commanded, and absence of signal.

```
Block {
  BlockType              Constant
  Name                   "Constant1"
  Position               [155, 496, 240, 524]
  Value                  "FCS.lon.k5b"
}
Block {
  BlockType              Constant
  Name                   "Constant2"
  Position               [35, 411, 120, 439]
  Value                  "FCS.lon.k5c"
}
Block {
  BlockType              DiscreteIntegrator
  Name                   "Discrete-Time\nIntegrato
  Ports                  [1, 1]
  Position               [395, 160, 430, 200]
  ShowName               off
  IntegratorMethod       "Forward Euler"
  ExternalReset          "none"
  InitialConditionSource "internal"
```

**Figure 3 Excerpt of Text File**

- 1 longitudinal_claw
- 1.1 Lon
- 1.1.1 Lon
- 1.1.1.1 Nz_cmd
- 1.1.1.2 Pitch_FB
- 1.1.1.3 Bus\nSelector
- 1.1.1.3.1 · Nz_g ·
- 1.1.1.3.2 · Q_dps ·
- 1.1.1.3.3 · AoA_deg ·
- 1.1.1.4 Constant
- 1.1.1.5 Constant1
- 1.1.1.6 Constant2
- 1.1.1.7 Discrete-Time\nIntegrator
- 1.1.1.8 Product
- 1.1.1.9 Sum
- 1.1.1.10 Sum1
- 1.1.1.11 Sum2
- 1.1.1.12 Sum3
- 1.1.1.13 Sum4
- 1.1.1.14 Sum5
- 1.1.1.15 Zero-Order\nHold1

**Figure 4 MOCET-generated FMEA Structure**

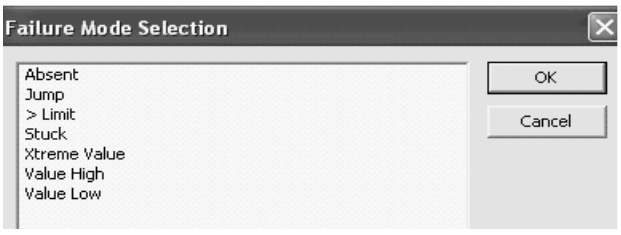


**Figure 5 Example Failure Mode Selection**

When the analyst highlights a model element on a screen similar to that shown in figure 4, the library failure modes for that element will appear on a drop-down menu in a format shown in figure 5. Not all failure modes may be pertinent in a given case; reasons for not using the entire library list include that a failure mode will not cause unsafe conditions or that the effects of a failure mode are identical with those of another one that is already being analyzed. The MOCET user can delete irrelevant failure modes and can add new ones that are pertinent in a given environment. All editing can be to the project library (so that changes will appear on the future menu for that component) or temporary (they will not appear on future component menus).

Local effects are usually evaluated at the output of the computing function on which the software is being processed and in simple systems they are highly correlated with the failure mode, e. g., "absent" failure mode results in loss of one or more data elements in the computer output, "stuck" failure mode results in stale output. These local effects are presented by MOCET as defaults after a failure mode is entered. They can be overridden by the analyst or permanently modified for a given project.

System level effects are normally provided at the outset of an FMEA by project safety personnel and/or system engineering, and they are derived from or made consistent with the end effects evaluated in the system fault tree analysis. In a simple case, like the longitudinal control system shown in figure 2, the analyst may be able to reason directly what end level effect will be caused by a given local effect. In general the FMEA should be a collaborative effort between software, system and safety engineering and the determination of end effects is preferably accomplished by such collaboration. MOCET provides for entering end effects by selection from a menu as shown in figure 6.

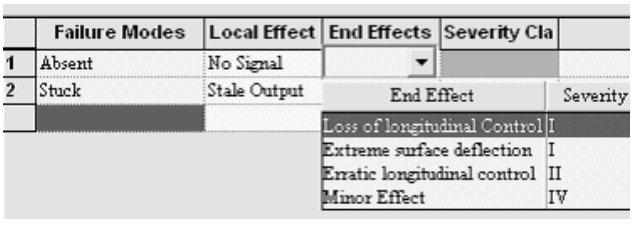


**Figure 6 End Effect and Severity Selection**

The severity classification is directly associated with the end effect and is assigned when the list of end effects is being compiled with participation by system and safety engineering. To complete this part of the FMEA the analyst first selects a failure mode and local effect from the main FMEA screen and then highlights an end effect and severity in the drop down menu. In more complex systems one or more intermediate effects levels may be desirable. MOCET provides for this by a drop-down screen as shown in figure 7.

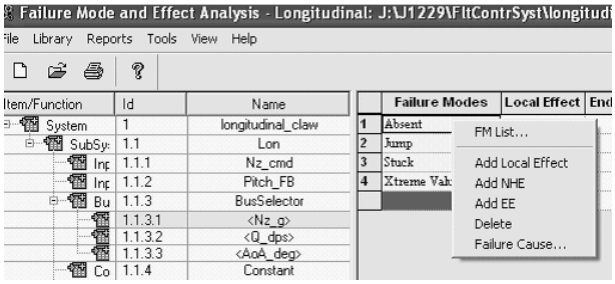


**Figure 7 Adding Next Higher Effect (NHE)**

When the analyst highlights a row (block) at a lower indenture level than previously analyzed a drop-down menu appears that permits adding a next higher effect (NHE). The effect is by default assigned to the next higher indenture level. In this example the ID of the selected row is 1.1.3.1 and the NHE will be assigned to the 1.1.3 level. The type of effect can be selected from library entries appropriate for that block and can be edited as previously described for the local effect. There are provisions for skipping an indenture level and assigning the NHE to a higher one.

## V. ADDING FAILURE DETECTION

The longitudinal FCS shown in figure 2 does not contain any explicit failure detection provisions. This does not necessarily mean that failures will go undetected; there could be additional longitudinal channels, permitting failure detection by comparison or voting, or there could be independent monitoring of attitude, attitude rate and angle of attack to detect system failures. For this discussion we assume that there are no such external failure detection means and that the highest priority is to protect the aircraft against excessive longitudinal acceleration. The possible sources for such excessive acceleration are (a) the input command, Nz_cmd, and (b) hardware or software failures within the FCS.

With respect to the input command the designers decided that it is more effective to limit it rather than to detect excessive values. There are a number of internal failure mechanisms that can cause excessive acceleration (e. g., those with end effects loss of longitudinal and extreme surface deflection) and rather than providing failure detection for each individual cause it was decided to base detection on the normal acceleration feedback signal (Nz-g), item 1.1.1.3.1 in figure 4. The failure detection level for the feedback signal must be higher than the limit value of the input command to allow for response of the system to aerodynamic disturbances but it must be set well below the design acceleration level of the aircraft structure. The resulting Simulink model of the longitudinal FCS is shown in figure 8. The command limiter appears in line with the command input at the top of the figure and the protection against internal failures is provided

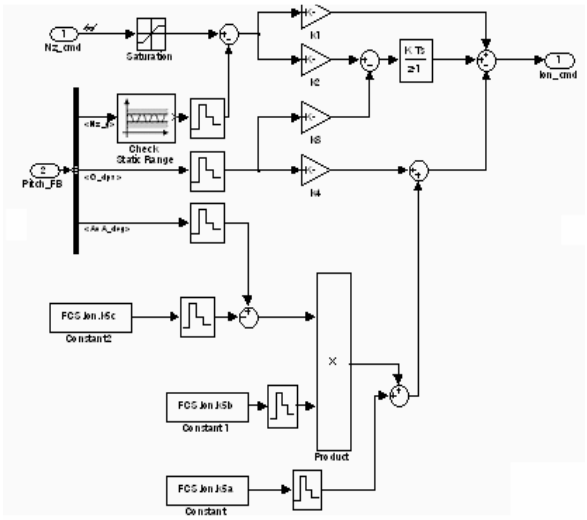


**Figure 8 FCS with Protection against Excessive Acceleration**

by the Check Static Range block in line with the Nz_g feedback. Details of the latter block are shown in figure 9.

When the normal acceleration feedback signal goes above the upper limit or below the lower limit the output is stopped and an assertion is raised. The latter event can be used to transfer longitudinal control to another channel, to switch to a standby flight control system, or to discontinue automatic control altogether.

Because the occurrence of larger than expected (or safe) values of a variable are a common tool for failure detection, the Check Static Range module has been added to the MOCET library for use at the analyst's discretion. A similar module, Check Dynamic Range, is provided for detection of excessive rates of change of a variable. Another type of generic failure detection implementation is the N-Wait module shown in figure 10. A typical use is the detection of absent or stale output due to hardware or software failures in a processing element. The N-part of the name indicates the number of wait cycles before a diagnosis of stuck output is made (zero output is a special case of stuck output). Figure 10

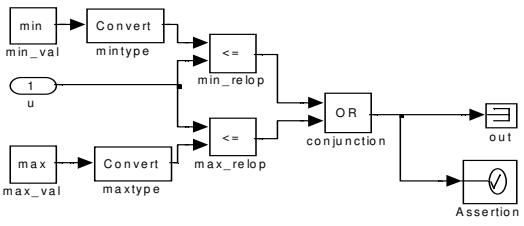


**Figure 9 Details of Check Static Range**

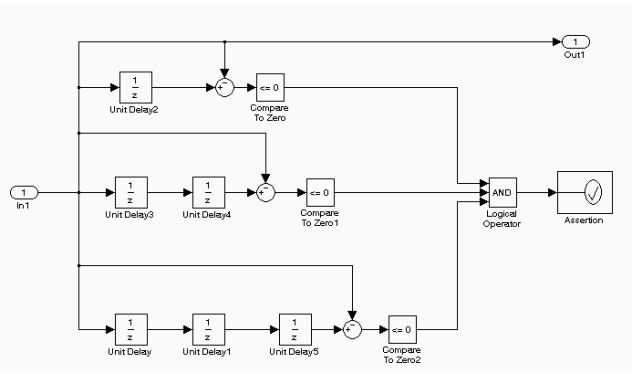


**Figure 10 N-Wait for N = 3**

shows the implementation for three processing steps. The current value of a digital variable is compared to its value during the preceding one (top), two (middle) and three (bottom) computing cycles. When the difference is zero, a logical "1" is sent to the triple AND gate and when all three inputs are "1s" (signifying that the signal has not changed over the last three cycles), the assertion is raised. The output is not interrupted because its value does not endanger the system. The assertion is used to transfer to another mode of control as described for the Check Static Range module.

## VI. CONCLUSION

It has been shown that model-based development environments, and particularly MATLAB/Simulink, provide a representation of programming elements that is equivalent to the use of parts in an electrical schematic. This permits generation of a software FMEA that is consistent with and can be combined with a hardware FMEA of the same component (e. g., a circuit board that contains the processing hardware and the program). This provides the following major benefits compared to previous implementations of software FMEA based on functional decomposition:

1. The program is composed of a finite collection of primitives each of which has defined failure modes
2. A complete listing of these primitives can be obtained from the design file.
3. The FMEA is constructed from the list of the primitives and considering for each primitive its pre-defined failure modes (that can be modified if necessary)
4. The FMEA thus constructed and augmented with analyst entries of failure effects and means of failure detection is as complete as a hardware FMEA compiled from a BOM and can be used for generation of a production test program.

The MOCET tool automates steps 1 – 3 above, and provides assistance for step 4. It is not only a labor saving device but assures consistent completion of all steps. A

software test program based on these steps can make the same claim of completeness as a hardware test program derived from an FMEA that is based on a BOM.

## REFERENCES


[1]Nancy G. Leveson and Clark S. Turner, "An Investigation of the Therac-25 Accidents", *IEEE Computer,* vol 26 no. 7, July 1993

[2] Wikipedia, Ariane 5/Launch History

[3] Michael A. Dornheim, "A Wild Ride". *Aviation Week & Space Technology,* September 26, 2005

[4] C. K. Hansen, "The Status of Reliability Engineering Technology 2001", *Newsletter of the IEEE Reliability Society,* January 2001

[5] SAE ARP 4754, *Certification Considerations for Highly-Integrated or Complex Aircraft Systems* and SAE 47761, *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.*

[6] MIL-STD 882, *System Safety Program Requirements,* Department of Defense

[7] ISA-84, *Process Safety Standards and User Resources,* ISA, Research Triangle Park, NC

[8] Fragola, J. R. and Spahn, J. F., "The Software Error Effects Analysis - A Qualitative Design Tool, 1973 IEEE Symposium on Computer Software Reliability, New York, NY

[9] Reifer, Donald J., "Software Failure Mode and Effects Analysis", *Transactions on Reliability*, vol.28, no.3, August 79

[10] Bowles, J. B. and Chi Wan, " Software Failure Modes and Effects Analysis for a Small Embedded Control System**"**, *Proc. of the 2001 Reliability and Maintainability Symposium,* Philadelphia PA, January 2001, pp. 1 – 6.

[11] Goddard, P. L. "Software FMEA Techniques", *Proc. of the 2000 Reliability and Maintainability Symposium,* Los Angeles CA, January 2000, pp. 118 – 122.

[12] Hecht, Herbert and Patrick Crane, "Rare Conditions and their Effect on Software Failures", *Proceedings of the 1994 Reliability and Maintainability Symposium,* pp. 334 - 337, January 1994

[13] Boggs, Wendy and Michael, *Mastering UML with Rational Rose*, Sybex, 2002

[14] http://users.isr.ist.utl.pt/~alex/micd0506/simulink.pdf