Using Failure Modes and Effects Simulation as a Means of Reliability Analysis

Daniel L. Palumbo

NASA LaRC
Hampton, VA

Abstract

Failure mode and effects analysis (FMEA) creates a knowledge base of system response to component failure. Reliability analysis draws on the FMEA in combination with component failure rates and system recovery rates to construct a reliability model that can be solved for the system probability of failure.

This paper introduces Failure Modes and Effects Simulation (FMES) as an effective way to do both FMEA and reliability analysis. Using FMES, a system is described as a connected set of components. A component is defined by its interconnects (inputs and outputs), its state (the component and its outputs can occupy different states) and its behavioral description. Behavioral descriptions characterize how changes occur in the component and its output states. The FMES technique has been evaluated in a tool called the Reliability Estimation Testbed (REST). A Reliability Modeling Language (RML) was developed as part of REST to support FMES. REST and RML have been used to analyze a system consisting of a quad computer connected to two reconfiguring mesh networks and several quad redundant devices, a system totaling 100 components.

Introduction

In this paper the analytical method Failure Modes and Effects Simulation (FMES) is introduced and its use in reliability analysis and producing automatically generated failure mode and effects analysis (FMEA) is explained. The first two sections introduce FMEA and demonstrate how it is used in conjunction with reliability analysis tools. The third section recalls the concept of reliability model generation and automated FMEA that was pioneered by the Reliability Model Generator [4][5]. The fourth section defines the FMES technique and how it can be used in conjunction with reliability analysis. The final section presents results that have been obtained using a software tool REST which has been designed specifically for FMES analysis.

Failure Mode and Effects Analysis

Failure Mode and Effect Analysis (FMEA) and its associate Failure Mode, Effects and Criticality Analysis (FMECA) are methods that have been used by the avionics and aerospace community to assess system vulnerability to component failure [1][2][3]. These analyses are typified by a bottom-up approach where a failure mode of a low-level component is assumed and its subsequent effect on system performance deduced.

This process can be generalized if we consider two components, C_1 and C_2 (see figure 1). Under normal circumstances, C_1 receives its input, I , produces output $C_1(I)$ and delivers this output to C_2 . C_2 , receiving a nominal input from C_1 , operates within specification. To perform an FMEA, we would consider a failure mode of C_1 , say C_1' . Component C_1 now produces output $C_1'(I)$. The effect of the failure of C_1 is assessed by reevaluating C_2 with respect to the input C_1' . C_2 may or may not continue to operate within specification. This process of propagating the component failure effect continues until it is determined that either the failure is "covered", i.e., the error effect is contained, or a system malfunction results.

As an example, consider the case where C_1 is a resistor and C_2 a light emitting diode (LED). These components could be used, for example, as a light source for a sensor application. If the resistor fails such that there is no current output (an open circuit), then the LED will cease to function. This could probably be considered a safe failure in that the complete lack of emissions from the sensor signal could be detected. If the resistor fails such that the current exceeds the LED's nominal rating (a short circuit), there would not be an immediate erroneous effect. However, the failure rate of the LED would now be increased with the result that a subsequent failure of the LED would cause a "safe" lack of emissions.

This type of analysis can be done quantitatively or qualitatively, i.e., with or without regard for the probability of the events occurring. In a qualitative analysis, a system is evaluated as being either Fail-Safe, Fail-Operational Fail Safe, or Fail-Operational Fail-Operational Fail-Safe. The Fail-Op Fail-Op Fail-Safe qualification is done for flight critical systems. Given a system of N components with a high degree of integration and, therefore, many state dependencies, a Fail-Op Fail-Op Fail Safe evaluation could take on the order of N factorial analyses.

Using Reliability Analysis with FMEA

The benefit of a quantitative analysis is that, as system unreliability (probability of being unsafe) is computed, many failure event sequences can be terminated because their probability of occurrence becomes insignificant. A quantitative analysis can be done using a Markov model where component failure rates determine the probability of system state transitions. Several Markov analysis software tools are available. Of interest to the discussion here is the SURE tool developed at NASA Langley Research Center [8]. An example of using SURE follows.

Returning to the resistor/LED sensor, consider the SURE models in figure 1a. This model shows the 3 failure events that are possible from the fully operational state, state 1. Having completed a first level FMEA, we would know that end states 2 and 3 are death states, i.e., the system no longer functions in these states, and that state 4 is an operational state. Before continuing with the analysis of state 4, the model is evaluated. Suppose the component failure rates are as follows

ResistorFailOpen = $1.0E-6$; ResistorFailShort = $1.0E-6$;
LEDFailNominal = $1.0E-6$; LEDFailOverDriven = $1.0E-2$;

The end state probabilities become

End State	Probability
2	$1.0E-5$
3	$1.0E-5$
4	$1.0E-5$

It appears in this case that further expansion is necessary (see figure 1b). The LED failure rate will increase to LEDFailOverDriven after the resistor fails shorted.

Figure 1. Markov Model of Resistor/LED.

The end state probabilities are now

End State	Probability
2	1.0E-5
3	1.0E-5
5	0.5E-6

As expected the secondary failure of the LED becomes less significant.

If a higher reliability is desired (lower probability of failure), redundant sensors can be interfaced to a voter (presumably in a processor). Using the above methodology and assuming, for the moment, that the voter and sensors are perfect, the initial system state will have 9 failure transitions (3 for each resistor/LED). The system remains operational due to the error masking function of the voter. The analysis then continues to the second failure level where over 50 component failures must be considered.

It can be seen that as system size and complexity grows, working with a Markov modeling tool such as SURE becomes cumbersome. The model can be raised to a higher level of abstraction with the realization that, for some system states, the system's response to a component's failure is generally the same. Given this, a set of state transition rules can be derived which efficiently describe a system's failure modes and effects. (In the worst case, one rule can be written for every model transition, but system behavior is rarely this degenerate.) The ASSIST modeling language was designed to provide such a capability [9].

Figure 2 is a listing of an ASSIST model for the redundant sensor system. The model starts with constant definitions of the component failure rates and state values. Using the rule based modeling paradigm, the analyst defines a state space over which the components of the system can range. In the *Space* statement, 3 resistors (R[1], R[2] and R[3]) and 3 LEDs (LED[1], LED[2] and LED[3]) are defined over their possible state values. These state variables are then initialized to start in the Nominal state.

The failure modes and effects are expressed in *Deathif* and *Tranto* statements. The *Deathif* statements describe conditions, in terms of the system's state variables, under which the system is considered failed. The ASSIST model in figure 2 has two sets of *Deathif* statements. The first *Deathif* covers the case in which there are functioning (or Nominal) LEDs. The second group of *Deathifs* describe a voter coverage requirement in which two functioning LEDs are needed to mask the errors from an overdriven LED.

```
ResFailOpen = 1.0E-6; ResFailShort = 1.0E-6;
LEDFailNominal = 1.0E-5; LEDFailOverDriven = 1.0E-2;
Nominal = 0;
ResShorted = 1; ResOpened = 2;
LEDOvDrv = 1; LEDFailed = 2;
```

```
Space = (R:array[1..3] of Nominal..ResOpened,
        LED: array[1..3] of Nominal..LEDFailed);
Start = (3 of Nominal, 3 of Nominal);
```

```
Deathif (LED[1] <> Nominal) and
        (LED[2] <> Nominal) and (LED[3] <> Nominal);
```

```
Deathif (LED[1] = LEDOvDrv) and (LED[2] <> Nominal) and
        (LED[3] <> Nominal);
```

```
Deathif (LED[2] = LEDOvDrv) and (LED[3] <> Nominal) and
        (LED[1] <> Nominal);
```

```
Deathif (LED[3] = LEDOvDrv) and (LED[1] <> Nominal) and
        (LED[2] <> Nominal);
```

```
For I=1,3;
```

```
If (LED[I] = Nominal) Tranto LED[I] = LEDFailed
By LEDFailNominal;
```

```
If (LED[I] = LEDOvDrv) Tranto LED[I] = LEDFailed
By LEDFailOverDriven;
```

```
If (R[I] = Nominal)
Tranto R[I] = ResOpened, LED[I] = LEDFailed
By ResFailOpen;
```

```
If (R[I] = Nominal) and (LED[I] = Nominal)
Tranto R[I] = ResShorted, LED[I] = LEDOvDrv
By ResFailShort;
```

```
EndFor;
```

Figure 2. Hand Coded Assist Model

The *Tranto* statements contain three parts: a condition, an effect and a rate. The LED *Tranto* statements are straightforward and describe the fact that the LED will fail by its nominal rate unless it is being overdriven. The resistor *Trantos* are more interesting in that their effect sections also include the response of the LED to the resistor failure. The resistor FMEA rules thus become specialized by the components to which the resistor is attached. Although it may seem trivial in this example, this inter-dependency of the rule base makes larger and more complex models harder to create, maintain and understand. For example, if the characteristics of the LED change, then the analyst must search the model for all places where another component might have an effect on an LED of this type. Note that a component does not have to be directly connected to another to have a failure effect on that component. As we shall see, this weakness is addressed in the automated FMEA techniques used in the FMES methodology.

Automated FMEA in RMG

One of the tasks of the Integrated Airframe/Propulsion Control System Architecture program (IAPSA) [10] was to do the reliability analysis of the proposed system design. The ASSIST/SURE tools were chosen for this analysis. During the course of the analysis two things were observed. First, the ASSIST language was foreign to system designers and thus impeded their use of the tool. Second, the order and regularity of the ASSIST transition rules made it appear that the ASSIST reliability model could be automatically generated. More specifically, it was speculated that the ASSIST reliability model could be automatically generated from a block diagram description of the system. This is preferred because system designers would find a

reliability analysis tool easier to use if the model could be expressed in terms of a block diagram.

A system block diagram is an interconnected set of discrete components. If a reliability model is to be generated from a system block diagram, it becomes immediately apparent that the system FMEA must first be generated from that same description. It was decided to study the design of an expert system for just such a purpose.

Such an algorithm was defined and embodied in the Reliability Model Generator (RMG) [4][5] prototype. RMG was used to successfully produce accurate ASSIST models from block diagram descriptions (models that were in some cases more accurate than hand coded models) [6]. The algorithm is based on a characterization of a component (or subsystem) as follows:

1. Components of the same type have similar behavior and therefore can be described by a single module definition.
2. A component has associated with it one or more mode variables. For example, a component might have one mode variable to describe its physical condition and one to describe its logical state.
3. The mode variables can take on one of several values. Examples of mode values are Nominal, Failed, Failed Open or Failed Shorted.
4. A component has inputs and outputs. A component's inputs and outputs are the means by which a component may affect or be affected by other components. These interconnections are often associated with explicit physical connections (such as a wire) but can be related to implicit dependencies (such as a heat conduction path).
5. A component's mode variable changes value due to local and remote effects. These mode value transitions are described as rules which are written in terms of a component's mode variables and inputs. Local transition rules, i.e., those without external stimulus, must be defined with a rate of occurrence.
6. A component's outputs change state due to input and mode variable transitions. Examples of output states are Nominal, Error and None.
7. A component's inputs retain a value equivalent to last received effect message.

Not getting bogged down in the exact syntactic details of RMG, the resistor and LED components can be characterized as shown in figure 4 (Ignore the bolded entries as they are part of the RML syntax).

The resistor definition (figure 4a) is straightforward and very similar to the ASSIST definition except, of course, for the output specification. The ON condition signifies that the output transition rule is enable only once, that is when the condition becomes true for the first time. Notice also the lack of any reference to the LED.

The LED (figure 4b) is more interesting. As specified by the ON statements, both mode and output transitions can be triggered by input transitions, i.e., resistor current effects. Note that the LED's output is None if either the LED fails or the input Current is None. The LED's dependency on the resistor current is completely specified in the LED module.

RMG was written in an expert shell that provided a graphical interface in which the user is able to define module types such as the resistor and LED above. To create the system description, the module types are instantiated as actual components and interconnected, i.e., the resistor output is connected to the LED input. To complete the system description, a system component is added and connected to the LED output. The system component creates specific requirements. For the case of a single LED, the system is considered to be failed if the LED is not emitting light.

Module: System;
Mode: State = (Nominal, Fail);
Input: Light = (Nominal, None, Bright);

Output: System = (Nominal, Fail);
ModeTransition:
ON (Light = None) TO (State = Fail);
OutputTransition:
ON (State = Fail) TO (System = Fail);

The reliability model is generated by defining a state space, reducing Deathif conditions and assembling Tranto statements. The state space is simply the set of component mode variables. The Deathif statements are constructed by taking those conditions that cause the system output to fail and then backward chaining through the system until the conditional expression is in terms of mode variables. In our resistor/LED example we would start with the System OutputTransition which specifies that the System fails when the System Mode variable System.State is equal to Fail. From here we trace back as follows.

1. As per the System ModeTransition rule, the System.State mode variable transitions to Fail when the System.Light input transitions to None, i.e. the failure condition is (System.Light=None).
2. The System.Light input is connected to the LED.Light output. The LED.Light output transitions to None on two conditions: (LED.PhysicalState = Fail), or (LED.Current = None). The first condition is in now in terms of a mode variable.
3. Tracing the second term, we have the LED.Current input connected to the Resistor.Current output. This output transitions to None when the Resistor.PhysicalState is FailOpen.
4. The trace is completed with two Deathif statements generated. Deathif (LED.PhysicalState = Fail); Deathif (Resistor.PhysicalState = FailOpen);

In most cases the Tranto statements are easily written from the ModeTransition statements. For example the LED ModeTransition

```
IF (LED.PhysicalState=Nominal)
TO (LED.PhysicalState=Fail)
BY OffRate;

becomes

IF (LED.PhysicalState=Nominal)
TRANTO (LED.PhysicalState=Fail)
BY OffRate;
```

However, consider this LED ModeTransition.

```
ON (LED.Current=OverSpec) AND
IF (LED.PhysicalState=Nominal)
TO (LED.PhysicalState=OverDriven);
```

This transition is triggered by the LED.Current input transition to OverSpec. This must be resolved in terms a mode variable. Tracing back to the resistor definition, we find the Resistor.Current is OverSpec when the Resistor.PhysicalState is FailShort. The LED ModeTransition thus becomes

```
ON (Resistor.PhysicalState=FailShort) AND
IF (LED.PhysicalState=Nominal)
TO (LED.PhysicalState=OverDriven);
```

ASSIST, however, cannot process ON transitions, i.e., instantaneous transitions without rates, so this ModeTransition must be further reduced using a resistor mode transition rule.

```
IF (Resistor.PhysicalState=Nominal) AND
IF (LED.PhysicalState=Nominal)
TO (LED.PhysicalState=OverDriven)
TO (Resistor.PhysicalState=FailShort)
BY (ShortRate);
```

This may now be written as the ASSIST statement

```
IF (ResistorPhysicalState=Nominal) and
(LEDPhysicalState=Nominal)
TRANTO (ResistorPhysicalState=FailShort)
```

```

(LEDPHYSICALSTATE=OverDriven)
BY ShortRate;

ResOpenRate = 1.0E-6; ResShortRate = 1.0E-6;
ResNominal=0; ResFailOpen=1; ResFailShort=2;

LEDOFFRate=1.0E-6; LEDOverRate=1.0E-2;
LEDNominal=0; LEDFail=1; LEDOverDriven=2;

Space = (ResPhyState:ResNominal..ResFailShort,
          LEDPhyState:LEDNominal..LEDOverDriven);
Start = (ResNominal, LEDNominal);

Deathif (LEDPhyState = LEDFail);
Deathif (ResPhyState = ResFailOpen);

IF (LEDPhyState=LEDNominal)
  TRANTO LEDPhyState=LEDFail
  BY LEDOffRate;

IF (LEDPhyState=LEDOverDriven)
  TRANTO LEDPhyState=LEDFail
  BY LEDOverRate;

IF (ResPhyState=ResNominal)
  TRANTO ResPhyState=ResFailOpen
  BY ResOpenRate;

IF (ResPhyState=ResNominal) and (LEDPhyState=LEDNominal)
  TRANTO ResPhyState=ResFailShort,
  LEDPhyState=LEDOverDriven
  BY ResShortRate;

```

Figure 3. RMG Generated ASSIST Model.

The completed ASSIST model is shown in figure 3. One of the difficulties in writing the model in ASSIST is the construction of unique names that fit into Assist's 12 character name requirement. The model in figure 3 is an idealization of what RMG would do. The RMG program generated much more cryptic names.

In practice RMG generated highly accurate models [6]. One feature of the models is hinted at in the example model in figure 3, i.e., the system failure conditions are written in low-level terms. As has been observed in practice, when an analyst codes directly in ASSIST, the tendency is to write the Deathif conditions in high-level terms and encode the effects in the Tranto statements as in figure 2. This method of modeling holds more information. In the RMG model, information is lost. We know that a resistor failing open is system failure, but do not know why. In the process of generating the reliability model, the FMEA was not retained thus impeding verification of the analysis.

Several other aspects of RMG limited its usefulness. RMG was written in an expert shell which, although it had a graphical interface, too often presented the shell's formidable internals to the user. This made RMG difficult to use. The RMG ASSIST models were much larger than manually coded models (due partially to the Deathif expansions) and often required long generation times. The larger ASSIST models caused longer ASSIST processing times and, in one case, could not be processed at all due to memory restrictions. These factors led to the extension of the RMG algorithm into Failure Modes and Effects Simulation.

Failure Modes and Effects Simulation

The Failure Modes and Effects Simulation (FMES) uses module definitions similar to that of RMG. However, as the name implies, the FMES simulates the process of failure occurrence and effect propagation. Thus, the FMES more closely emulates the FMEA that an analyst performs. The FMES is coded in the Reliability Modeling Language (RML) [7] which is then translated to the C language and executed to perform the analysis. Although the syntactic details of RML are different from the example

```

Module: Resistor;
Mode: PhysicalState = (Nominal, FailOpen, FailShort);
Rate: OpenRate = 1.0E-6;
Rate: ShortRate = 1.0E-6;
Output: Current = (Nominal, None, OverSpec);
ModeTransition: Open
  IF (PhysicalState=Nominal)
  TO (PhysicalState = FailOpen)
  BY OpenRate;
ModeTransition: Short
  IF (PhysicalState=Nominal)
  TO (PhysicalState = FailShort)
  BY ShortRate;
OutputTransition: NoCurrent
  ON (PhysicalState=FailOpen)
  TO (Current=None);
OutputTransition: OverCurrent
  ON (PhysicalState=FailShort)
  TO (Current=OverSpec);

```

Figure 4a. Resistor Module

```

Module: LED;
Mode: PhysicalState = (Nominal, Fail, OverDriven);
Rate: OffRate = 1.0E-6;
Rate: OverRate = 1.0E-2;
Input: Current = (Nominal, None, OverSpec);
Output: Light = (Nominal, None, Bright);
ModeTransition: FailOff
  IF (PhysicalState=Nominal)
  TO (PhysicalState=Fail)
  BY OffRate;
ModeTransition: FailOver
  IF (PhysicalState=OverDriven)
  TO (PhysicalState=Fail)
  BY OverRate;
OutputTransition: NoLight
  ON (Current=None) OR
  ON (PhysicalState=Fail)
  TO (Light=None);
ModeTransition: OverDriven
  ON (Current=OverSpec) AND
  IF (PhysicalState=Nominal)
  TO (PhysicalState=OverDriven);
OutputTransition: Bright
  ON (PhysicalState=OverDriven)
  TO (Light=Bright);

```

Figure 4b. LED Module

```

Module: System;
Mode: State = (Nominal, Fail);
Input: Light = (Nominal, None, Bright);
Deathif: SysFail (State = Fail);
ModeTransition: NoLight
  ON (Light = None)
  TO (State = Fail);

```

Figure 4c. System Module

Figure 4. Module Definitions for Resistor and LED

descriptions of the resistor and LED given in the previous section, the semantic content is similar except for the following extensions.

1. In RML Deathifs are expressed explicitly in either the component modules or a system module.
2. Transition and Deathif statements can be given labels. The labels aid in quickly identifying sequences in the FMES output.
3. Output transitions generate messages to signal the output's change of state.

4. Transition and Deathif statements can contain references to C language functions. This extension gives the analyst great latitude in expressing component behavior.

Returning to the resistor/LED example, consider the module descriptions of figure 4. The module descriptions are identical to the RMG descriptions except for the bolded entries, i.e., the labels and Deathif statement. The following instantiations and connections are made to complete the model. (The details of component initialization are straightforward and not presented here. See [7] for a more thorough description.)

```
Resistor: R1;
LED: LED1;
System: System;

Connect(R1.Current, LED1.Current);
Connect(LED1.Light, System.Light);
```

Assuming that resistor and LED modules have been defined, instantiated and connected in RML, the process begins by constructing a simulation executable. The system state space, a set of mode transition rules and a set of Deathif rules are assembled from the system components as was done in RMG. Because RML retains the module input/output information, the mode transition and system failure rules can be copied verbatim making this process straight forward.

At system start, the state space is as follows.

```
R1.PhysicalState = Nominal;
LED1.PhysicalState = Nominal;
System.PhysicalState = Nominal;
```

The simulation then begins by first evaluating the Deathif statements and then expanding the mode transitions. For the purpose of illustration, we will trace the R1.Short sequence.

- I. Evaluate state (Nominal, Nominal, Nominal).
 1. Deathif System.SysFail is False.
 2. Mode rule R1.Short is enabled.
 - R1.PhysicalState set to FailShort.
 - Evaluate R1 Outputs.
 - a. Output rule R1.OverCurrent is enabled.
 - R1.Current set to OverSpec.
 - b. Message "OverSpec" sent to input LED1.Current.
 - c. Mode rule LED1.OverDriven is enabled.
 - LED1.PhysicalState set to OverDriven.
 - Evaluate LED1 outputs.
 - a. Output rule LED1.Bright is enabled.
 - LED1.Light is set to Bright.
 - b. Message "Bright" sent to input System.Light.
 - No effect.
 - c. New state (FailShort, OverDriven, Nominal) is saved.
 3. Mode rule R1.Open is enabled.
 - ...
 - f. New state (FailOpen, Nominal, Fail) is saved.
 4. Mode rule LED1.FailOff is enabled.
 - ...
 - d. New state (Nominal, Fail, Fail) is saved.
 - II. Evaluate state (FailShort, OverDriven, Nominal)
 1. Deathif System.SysFail is False.
 2. Mode rule LED1.FailOver is Enabled.
 - LED1.PhysicalState set to Fail.
 - Evaluate LED1 outputs.
 - a. Output rule LED1.NoLight is enabled.
 - LED1.Light is set to None.
 - b. Message "None" sent to input System.Light.
 - c. Mode rule System.NoLight is enabled.
 - System.State is set to Fail.
 - Evaluate system outputs.
 - None.
 - d. New state (FailShort, Fail, Fail) is saved.
 - III. Evaluate state (FailOpen, Nominal, Fail)
 1. Deathif System.SysFail is True.

- Analysis of this sequence terminates.
- IV. Evaluate state (Nominal, Fail, Fail)
 1. Deathif System.SysFail is True.
 - Analysis of this sequence terminates.
- V. Evaluate state (FailShort, Fail, Fail)
 1. Deathif System.SysFail is True.
 - Analysis of this sequence terminates.
- VI. Analysis terminates.

As can be seen the FMES mimics the FMEA process. An additional aspect of the FMES is that reliability can be computed as the FMES is processed. With reliability analysis integrated with the FMEA analysis, the FMES output would appear as follows.

```
(R1.Open)(System.Fail)
[9.99E-06, 1.00E-5] cum 9.99E-06
(LED1.FailOff)(System.Fail)
[9.99E-06, 1.00E-5] cum 1.99E-05
(R1.Short)(LED1.FailOver)(System.Fail)
[4.83E-07, 5.00E-7] cum 2.00E-05
```

Where the first column of numbers are the bounds on probability of failure for that particular sequence and the second column is a cumulative total. To view the complete FMEA a trace output can be obtained.

```
(R1.Open->R1.NoCurrent->LED1.NoLight->System.NoLight)
(System.Fail) [9.99E-06, 1.00E-5] cum 9.99E-06
(LED1.FailOff->LED1.NoLight->System.NoLight)
(System.Fail) [9.99E-06, 1.00E-5] cum 1.99E-05
(R1.Short->R1.OverCurrent->LED1.OverDriven->LED1.Bright)
(LED1.FailOver->LED1.NoLight->System.NoLight)
(System.Fail) [4.83E-07, 5.00E-7] cum 2.00E-05
```

Having the probability of occurrence of a failure event sequence available in the FMES output aids in the inspection of large complex systems by reducing the number of sequences that must be considered. An additional benefit of the FMES technique is that demand for computer memory resources during the analysis is greatly reduced, making the analysis of large system possible. The results of the analysis of a significant part of the IAPSA architecture follows.

Results

The Reliability Estimation System Testbed (REST) [7] was developed to probe the use of RML and FMES. A target system architecture was taken from the IAPSA study. The architecture consists of a quad fault tolerant computer connect to 2 reconfiguring mesh networks. Four quad redundant I/O devices are attached to the network.

Modules can represent both physical and logical components. Logical components contribute functionality to the design definition but are not associated with a failure rate. An example of a logical component is the Voter which has unique functionality yet is considered to be an integral part of the Processor, i.e., when the Voter fails the Processor fails. A second logical module type is the redundancy manager algorithm. Ten module types were defined for this model.

1. Processor: This is the CPU processing unit. It has one failure mode transition labeled Fail.
2. Voter: This inter channel majority voting device retains error data and allows channels to be disabled. This is a logical component.
3. FTPmodule: This is the redundancy management algorithm that manages that Voters. It is a logical component with one recovery mode transition labeled Recover.
4. Interface: This device connects the CPU to the mesh network. Only one Interface is enabled to a network at a time. The Interface has on failure mode transition labeled Fail.
5. Node: This device connects Links together into a virtual bus. The interconnections are enabled and disabled by the

- Network Manager. The Node has one failure transition labeled Fail.
6. Link: These are simple devices used to interconnect nodes, Interfaces and Devices. The Link has one failure mode transition labeled Fail.
 7. NetMan: The Network Manager is the redundancy manager for the networks. It responds to network errors by reconfiguring the links around failures. This is a logical component with one recovery mode transition labeled Recover.
 8. Device: This is a generic I/O device. The Device has one failure mode transition labeled Fail.
 9. DevMan: This is a redundancy management algorithm for a quad set of I/O Devices. The DevMan is a logical component with one recovery mode transition labeled Recover.
 10. System: The System module contains the 6 Deathif statements corresponding to the six fault tolerant subsystems (FTP, Net, DevA, DevB, DevC and DevD).

A total of 105 physical and logical components were instantiated from these 10 module types. The resulting RML model contained 1100 lines of code. The model took 3 hours to process on a Sparc 2 workstation. The analysis searched over 1.3 million states at a probability depth of 1.0E-12. A total of 265,000 death states were found.

The FMES output can be viewed at 3 levels of detail. The following excerpts from the FMES reports exemplify the kind of information and insight the reports contain. The top level is "Summary By Type" in which the transition events are grouped according to module type.

```
(Processor,Fail)(FTPmodule,Recover)(NetMan,Recover)
(Processor,Fail)(FTPmodule,Recover)(NetMan,Recover)
(Processor,Fail)(System,FTP)
[6.52E-10,6.67E-10] cum. 6.67E-10
```

This is the first and most probable sequence in the SummaryByType report. Two processor failures are each followed by FTP and network recoveries. The third processor failure brings system failure. This is a "lack of spares" failure sequence, i.e., the system ran out of spare processors. In the event of two processor failures and no recoveries, we would expect a "lack of coverage" failure due to voter error. In fact we find this event sequence

```
(Processor,Fail)(Processor,Fail)(System,Net)
[1.10E-10,1.11E-10] cum 4.94e-09
```

which is the fourteenth failure sequence. Note that the network Deathif condition signaled system failure. These failed processors passed errors to their associated networks with the result that the System network Deathif condition triggered (i.e., before the FTP deathif condition). If two processors are connected to the same network, their failures will effect only one network and thus not enable the System.Net Deathif. The FTP Deathif then catches these less probable events.

```
(Processor,Fail)(Processor,Fail)(System,FTP)
[5.47E-11,5.56E-11] cum. 5.54e-09
```

This effect is better understood if the actual processor instantiations are known. The next level of detail is "Summary By Array" where transition events are grouped according to arrays of the same type. (Note that the totals have been left off the following reports).

```
(P4,Fail)(P1,Fail)(Sys,Net) [1.37E-11,1.39E-11]
(P1,Fail)(P4,Fail)(Sys,Net) [1.37E-11,1.39E-11]
(P4,Fail)(P2,Fail)(Sys,Net) [1.37E-11,1.39E-11]
(P1,Fail)(P3,Fail)(Sys,Net) [1.37E-11,1.39E-11]
(P1,Fail)(P2,Fail)(Sys,FTP) [1.37E-11,1.39E-11]
(P2,Fail)(P1,Fail)(Sys,FTP) [1.37E-11,1.39E-11]
(P4,Fail)(P3,Fail)(Sys,FTP) [1.37E-11,1.39E-11]
(P3,Fail)(P1,Fail)(Sys,Net) [1.37E-11,1.39E-11]
```

```
(P2,Fail)(P4,Fail)(Sys,Net) [1.37E-11,1.39E-11]
(P3,Fail)(P4,Fail)(Sys,FTP) [1.37E-11,1.39E-11]
(P3,Fail)(P2,Fail)(Sys,Net) [1.37E-11,1.39E-11]
(P2,Fail)(P3,Fail)(Sys,Net) [1.37E-11,1.39E-11]
```

The processors have been instantiated as four, one element arrays P1, P2, P3 and P4. Viewing this output makes it clear that the only sequences which do not trigger the network death condition are those sequences involving P1,P2 and P3,P4, i.e., processors connected to the same network.

Increased detail does not always yield more information. Take for example the Summary By Array sequences which describe link and node failures.

```
(ANodes,Fail)(BLinks,Fail)(Sys,Net) [5.18E-12,5.28E-12]
(BNodes,Fail)(ALinks,Fail)(Sys,Net) [5.18E-12,5.28E-12]
(ALinks,Fail)(BNodes,Fail)(Sys,Net) [5.18E-12,5.28E-12]
(BLinks,Fail)(ANodes,Fail)(Sys,Net) [5.18E-12,5.28E-12]
```

These sequence describe the events where a node or link fails on one network and then a node or link fails on the other network, thus disabling both networks. These sequences could be viewed in the lowest level of detail, "Detailed Events", where events are recorded without grouping. The four sequences would expand into nearly 2,000 event sequences in the Detailed Events report and yield virtually no more information.

Concluding Remarks

The FMES method merges FMEA and reliability analysis providing an alternative approach to system safety analysis. FMES has been shown to be computationally efficient in calculating reliability without sacrificing accuracy. The output of an FMES analysis yields indepth information about the system failure effects as well as the probability that these effects will occur. This output aids in understanding system and model behavior. On more than one occasion the FMES approach has produced analyses which uncovered errors in hand-generated models. The modularity of FMES models makes the FMES expandable and reusable. These combined factors lead to the conclusion that FMES can be used with confidence in practical applications.

References

1. ARP 1834, "Fault/Failure Analysis for Digital Systems and Equipment", Society of Automotive Engineers, August 7, 1986.
2. MIL-STD-1629A "Procedures for Performing a Failure Mode, Effects and Criticality Analysis", Notice 2 Version, November, 1985.
3. D. Elwell, "Avionics Data Bus Integration Technology", DOT/FAA/CT-91/19, December 1991
4. G. C. Cohen, "Reliability Model Generator," NASA Contractor Report 182005, 1990.
5. C. M. McCann and D. L. Palumbo, "Reliability Model Generator for Fault Tolerant Systems," in AIAA Design, Systems and Operations Conference, Sept. 1988.
6. D. L. Palumbo, "Advanced Techniques in Reliability Model Representation and Solution", NASA TP 3242, August 1992.
7. D. Nicol, "User's Guide to the Reliability Estimation System Testbed (REST)", NASA TM 107596, August 1992.
8. R. W. Butler, "SURE Reliability Analysis Program and Mathematics", NASA TP 2764, March 1988.
9. R. W. Butler, "An Abstract Language for Specifying Markov Reliability Models," IEEE Trans. Rel., vol. R-35, no. 5, pp. 595-601, Dec. 1986.
10. G. C. Cohen, "Final Report: Design of an Integrated Airframe/Propulsion Control System Architecture," NASA Contractor Report 182007, 1990.