NASA/CR–2019-220283

# Safe and Optimal Techniques Enabling Recovery, Integrity, and Assurance

*Kit Y. Siu and Heber Herencia-Zapana*
*General Electric Global Research Center, Niskayuna, New York*

*Panagiotis Manolios*
*Northeastern University, Boston, Massachusetts*

*Michael Noorman and Richard Haadsma*
*General Electric Aviation Systems, Grand Rapids, Michigan*

June 2019

NASA/CR–2019-220283

# Safe and Optimal Techniques Enabling Recovery, Integrity, and Assurance

*Kit Y. Siu and Heber Herencia-Zapana*
*General Electric Global Research Center, Niskayuna, New York*

*Panagiotis Manolios*
*Northeastern University, Boston, Massachusetts*

*Michael Noorman and Richard Haadsma*
*General Electric Aviation Systems, Grand Rapids, Michigan*

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research
Center under Contract NNL15AA02C

June 2019

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

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA  23681-2199
Fax: 757-864-6500

# SOTERIA

Safe and Optimal Techniques Enabling Recovery, Integrity, and Assurance

GE Global Research
Kit Siu Heber Herencia-Zapana
siu@ge.com heber.herencia-zapana@ge.com

Northeastern University
Panagiotis Manolios
pmanolios@gmail.com

GE Aviation Systems
Michael Noorman Richard Haadsma
michael.noorman@ge.com richard.haadsma@ge.com

# Contents

# 1 Introduction

There is a trend in the aviation industry from federated to integrated computing systems (Wolfig & Jakovlijevic, 2008). Combining a number of traditional stand-alone federated systems into an integrated common platform (called Integrated Modular Avionics, IMA) has the benefit of increased power efficiency, reduced support hardware, and reduced cabling. Changing from traditional, federated systems has a significant impact on the system architecture and hence the process of how avionic systems are to be analyzed. Traditional approaches to safety analysis become inefficient when functional boundaries can no longer be assumed for failure independence and fault isolation.

We developed a tool to accelerate the safety engineer's ability to perform safety analysis of IMA systems through modeling, as well as optimize the system engineer's ability to develop a system through architecture synthesis. This work was the result of a three-year research effort called **SOTERIA**[1] (**S**afe and **O**ptimal **T**echniques **E**nabling **R**ecovery, **I**ntegrity, and **A**ssurance). In this program, we developed a compositional modeling language that supports rapid development, modification, and evaluation of architectures. The modeling language is structured such that the end-user defines a library of components with information on component reliability, connectivity, and fault propagation logic. The system model is built by instantiating the components from the library, connecting the components, and identifying the top-level faults of interest. Our tool is compositional in that the end-user only needs to define safety aspects at the component level. The tool takes the model and automatically synthesizes both the qualitative and quantitative safety analyses. We go further by allowing users to describe system information such as components to use in an architecture and their connection compatibility and automatically synthesize an architecture that meets the top-level probability target adhering to end-user specified constraints. This capability allows users to rapidly explore a design space.

We use OCaml as our programming language (www.ocaml.org). OCaml is a high-level, type-safe, functional programming language, that also has imperative language constructs. It is as fast as C and scales very well. There are several online articles that tout the features of OCaml (Meister, 2014) (Waclena, 2006) (Minsky, 2016). OCaml is strongly typed so that one type cannot be accidentally used with another. OCaml does static type checking, where the compiler determines the data types at compile time and will not compile a program with type conflicts. Users can define their own types, but the compiler will also automatically infer types that are not explicitly declared, which is very powerful for preventing inconsistent use of functions and data types.

OCaml is the base language from which Professor Manolios and his collaborators developed CoBaSA, a high-level modeling and specification language coupled with a synthesis tool (Manolios & Papavasileiou, ILP Modulo Theories, 2013) (Hang, Manolios, & Papavasileiou, 2011). CoBaSA, later renamed to Inez (github.com/vasilisp/inez), is based on the Jane Street Core (janestreet.github.io), which is an industrial strength alternative to OCaml's standard library. On this program, we follow Professor Manolios' previous success and use OCaml with the Jane Street Core to develop the modeling language and the safety analysis and synthesis capabilities.

[1] A bit of trivia – in modern Greek, soteria (σωτηρια) means salvation, and in Greek mythology, Soteria was the goddess or spirit of safety, and of deliverance and preservation from harm.

The remainder of the report is organized as follows. Section 2 gives a description of IMA architectures. Section 3 goes over fault-tree analysis. In section 4 we describe how fault-tree analysis is implemented in our tool and how ours compare with existing tools. In section 5 we discuss our fault-tree synthesis design and capability. In section 6 we describe our modeling language including validation checks to assist the end-user. In section 7 we give an example of our analysis capability and describe our ability to handle modeling aspects such as loops. In section 8 we demonstrate our tool's visualization capability which includes the depiction of fault trees and architectures at various levels of abstractions. In section 9 we show an extensive number of examples, including an industrial size problem. In section 10 we describe our architecture synthesis capability. Finally, we end the report with some conclusions in section 11.

The source code is available on GitHub under the following code repositories.

- github.com/ge-high-assurance/safety-analysis
- github.com/ge-high-assurance/SOTERIA

## 2 IMA Architectures

The challenge to assure safety on an IMA architecture is compounded by having multiple vendors supplying different functions that must share resources on the same distributed platform. Safety assumptions made by vendors of the hosted functions may not be valid. The challenge for a platform supplier is to provide platform level services that ensure non-interference between functions and provide the means for safe and coordinated interaction with no undesired emergent properties. The goal of the platform level modeling techniques developed in this program is to enhance the analysis capability of the basic infrastructure features of the platform services to effectively evaluate the safety aspects of the system. The analysis will validate the existing architecture services (such as redundancy and data integrity protection) and uncover areas where architecture changes are needed.

An IMA architecture provides common platform resources such as computing, data transport, and I/O. The architecture can then support hosted functions that implement aircraft functionality using the common IMA resources, as illustrated in Figure 1. There is a published guidance (DO-297, 2005) that discusses platform, hosted function (HF), and IMA system development and safety activities.

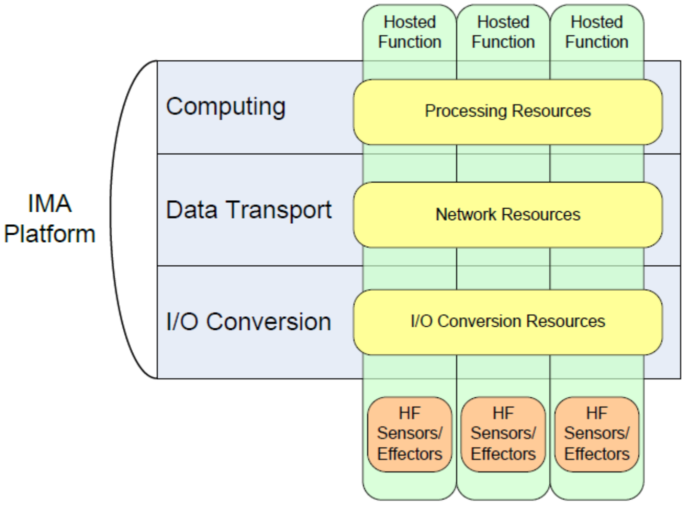


Figure 1 - Hosted functions on an IMA platform, using platform resources.

As an example, Figure 2 shows an IMA architecture that we will model and analyze on this program. Looking at the diagram from the bottom up, the caption “Signal Source/Destination” means that the architecture can support either a sensor (signal source) or an effector (signal destination). For this project, we narrowed our scope to signal sources (i.e., sensors). In this case then there are 3 data sources transporting unidirectional up to a general processing module (GPM). Each data source is connected to a remote interface unit (RIU). There is a network of switches (SW) in between the RIUs and the GPM. With sensor voting, 2 of 3 must match (or be within range of each other) in order for the hosted function to use the data. Note that the models and analysis developed on this program would work the same in the unidirectional down from GPM to effectors.

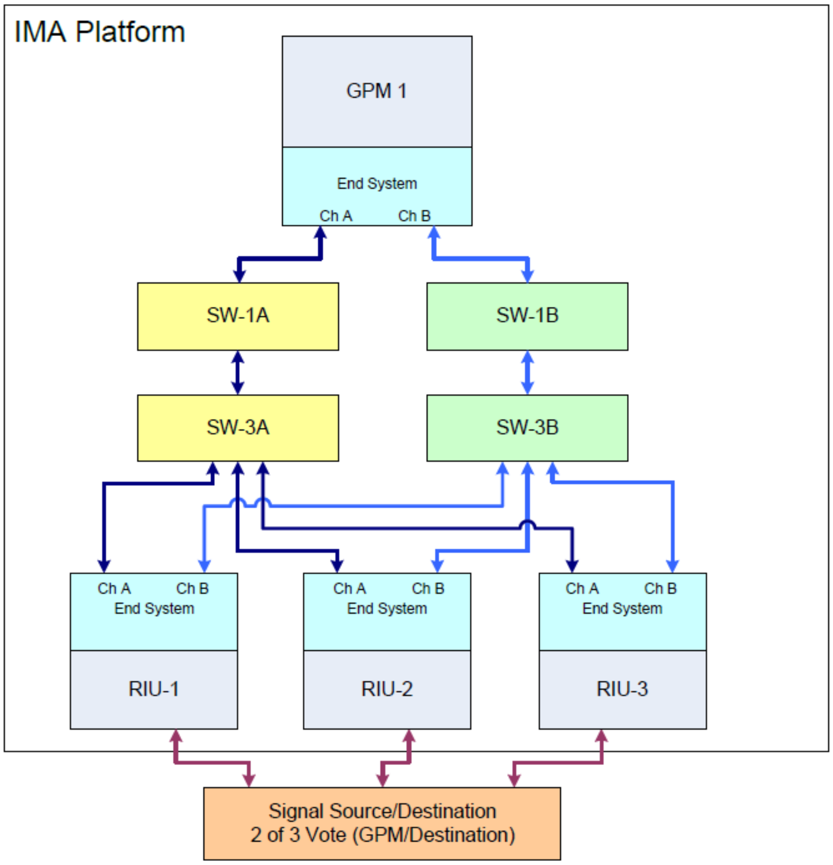


Figure 2 – An example of an IMA Platform Architecture.

To further illustrate, Figure 3 shows how data flows through this architecture (illustrated with the addition of the red arrows). From the bottom up, data is replicated on channel A and channel B of the network. The GPM uses one copy from channel A or channel B (only one is used and there is no voting between the channels). The GPM compares the data from the 3 data sources and does a 2-of-3 vote. The failure conditions of concern here are loss of function and undetected erroneous data. These are the failure condition flows we will model and analyze on this program.

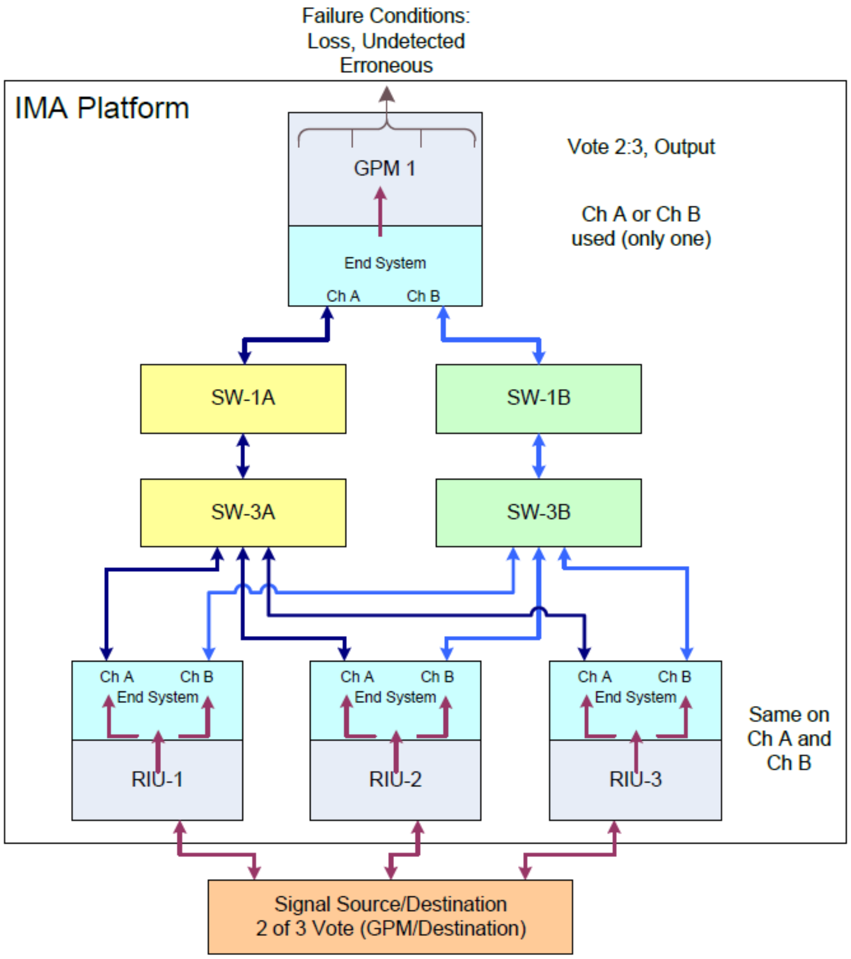


Figure 3 - Data flow through an example IMA platform architecture.

In addition to the architecture in Figure 2, we identified several other generic architectures that are representative of modern day avionics. These architectures include features such as sensor voting and end-to-end network integrity protection. Identifying these different architectures gives us a representative set of layouts such that when we develop the modeling constructs they are able to handle the breath of available possibilities that a safety engineer would have to analyze.

The different architectures are listed and illustrated in the figures below.

- 1 Sensor, 1 RIU with an end system (ES), network switches (SW), 1 GPM with an end system (Figure 4).
- 2 Sensors, 2 RIUs each with an end system, network switches, 1 GPM with an end system (Figure 5).
- 2 Sensors, 2 RIUs each with an end system, network switches, 2 GPMs each with an end system (Figure 6).

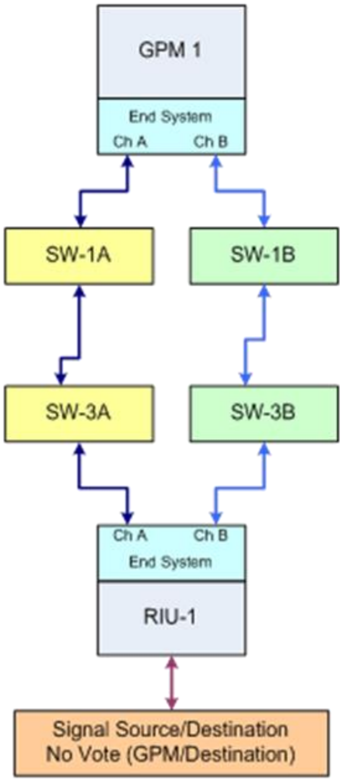


Figure 4 - Generic IMA Platform Architecture 2.

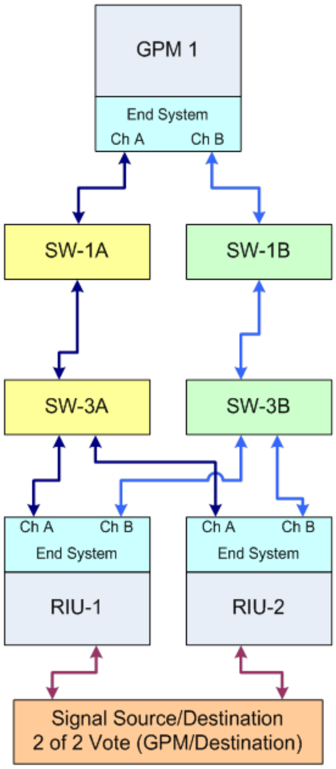


Figure 5 - Generic IMA Platform Architecture 3.

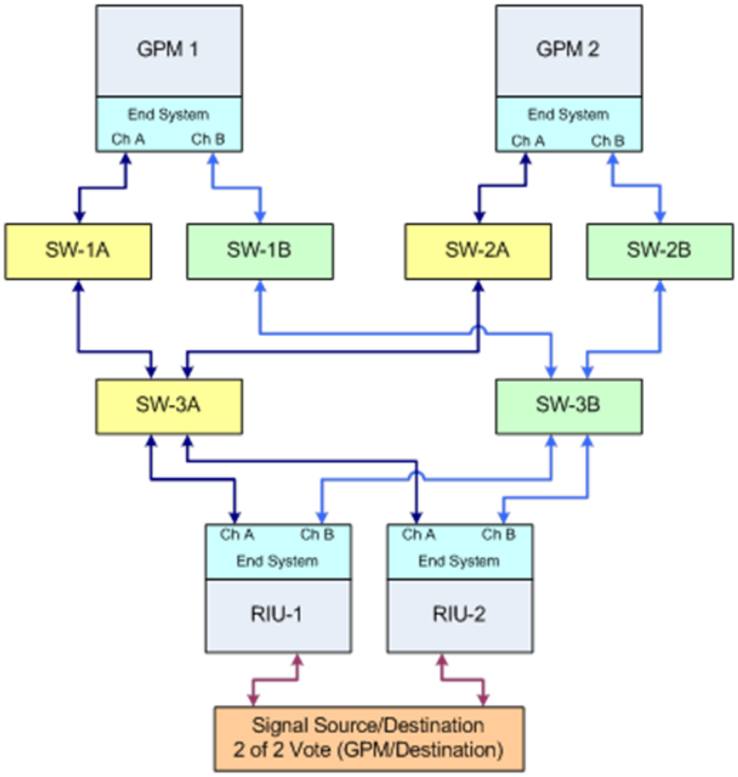


Figure 6 - Generic IMA Platform Architecture 4.

# 3 Fault Tree Analysis (FTA)

FTA is a popular method used in industry that focuses on an undesirable top-level event. The following description is from SAE Standard, ARP4761 and is written here for reference (S18, 1996).

FTA is a “top-down” system evaluation procedure in which a qualitative model for a particular undesired event is formed and then evaluated. FTAs are primarily used by system safety engineers to ensure that design safety aspects are identified and controlled. It is accepted by both the civil and military certification authorities as a method (preferred method) to show compliance with safety and certification regulations, requirements, and objectives. Safety is a measure of average risk – FTA accuracy is good enough to satisfy “on the order of” targets.

FTA usage includes:

a) Facilitation of technical/certification authority assessments and reviews.
b) Assessment of a design modification with regards to its impact on safety.
c) Quantification of the top event probability of occurrence.
d) Allocation of probability budgets to lower-level events.
e) Visibility into the contribution of development errors by providing a format for mixed quantitative and qualitative assessment.
f) Assessment of single and multiple-fault effects.
g) Assessment of exposure intervals, latency, and “at-risk” intervals with regard to their overall impact on the system.
h) Visibility of potential common-cause boundaries.
i) Assessment of common-cause fault sources.
j) Assessment of fail-safe design attributes (fault-tolerant and error-tolerant).

The fault tree graphical representation is hierarchical and takes its name from the branching that it displays. It is this format which makes this analysis a visibility tool for both engineering and the certification authority.

A fault tree minimal cut set is a smallest set of Primary Events which must all occur in order for the undesired top-level event to occur. Cut sets are useful when assessing large fault trees for complex systems. Primary drivers causing the undesired top event, including single point failures, can be easily identified.

## 3.1 IMA Fault Trees

The failure conditions of concern are loss of function and undetected erroneous data.

For the IMA platform depicted in Figure 2, loss of function occurs when there is a loss of the GPM, a loss of the network, or a loss of RIUs. When building the fault tree for loss of the network, we need to account for there being network redundancy. When building the fault tree for loss of RIUs, we need to account for there being a 2-of-3 voting scheme at the GPM. When modeling loss of function, the 2-of-3 vote is modeled as “loss of 2-of-3 sources” because that is the point at which 2 sources are no longer available to compare. The loss of function fault tree for the IMA platform is shown in Figure 7.

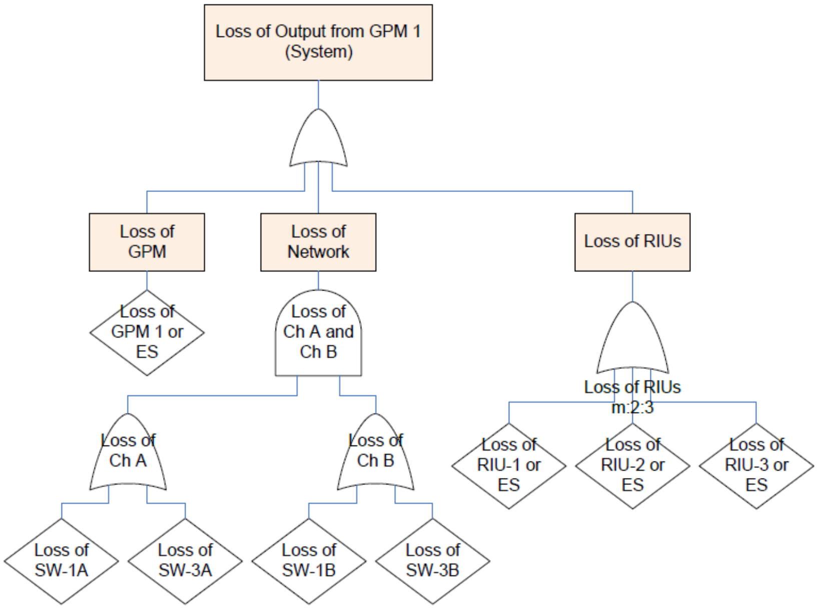


Figure 7 - Loss of function fault tree for IMA platform in Figure 2.

Similarly, undetected erroneous data is outputted by the GPM when the GPM is erroneous, the network is erroneous, or the RIUs are erroneous. When building the fault tree for when the network is erroneous, we need to account for network integrity protection. In determining when the RIUs are erroneous, we need to account for voting of data in the GPM. The fault tree for undetected erroneous data is shown in Figure 8.

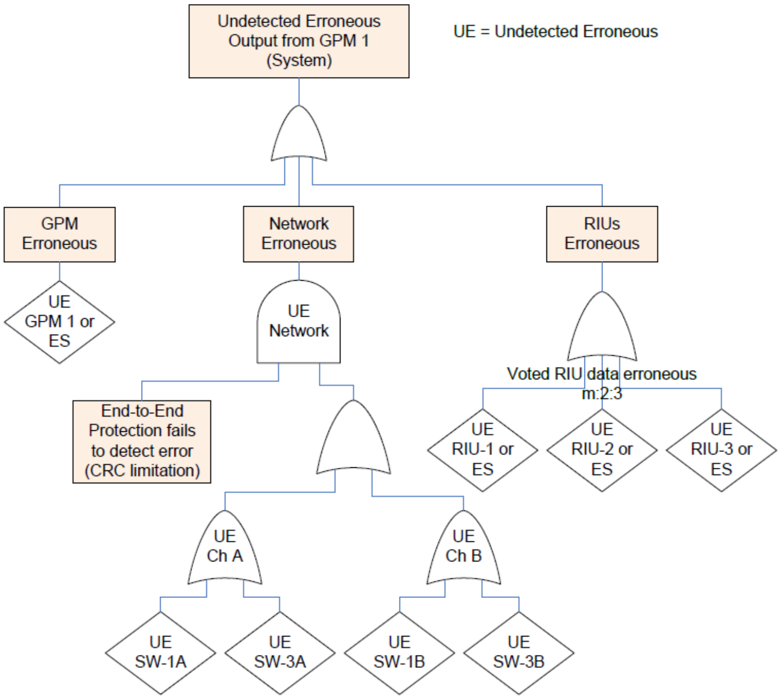

Undetected Erroneous Output from GPM 1 (System)
UE = Undetected Erroneous
GPM Erroneous
Network Erroneous
RIUs Erroneous
UE GPM 1 or ES
UE Network
Voted RIU data erroneous m:2:3
End-to-End Protection fails to detect error (CRC limitation)
UE RIU-1 or ES
UE RIU-2 or ES
UE RIU-3 or ES
UE Ch A
UE Ch B
UE SW-1A
UE SW-3A
UE SW-1B
UE SW-3B

Figure 8 - Undetected erroneous data fault tree for the IMA platform in Figure 2.

## 3.2 Other techniques

Our program is focused on FTA as the method for driving system architecture. There are other techniques, such as Failure Mode Effects Analysis (FMEA). Since FMEAs only look at the effects of single failures, they are limited in value to a system/aircraft level safety assessment. However, FMEA for each component being modeled could be an input to the models developed in this program. Any FMEA work would be considered an input to the failure rate determination for the component models. Every failure mode in the FMEA for an IMA component we model would map to one of the failure conditions of concern, such as loss of function or undetected erroneous data, or could be deemed as having no effect. If there was a "loss of redundancy" type failure effect (e.g. something that alone isn't loss of function or undetected erroneous data, but also cannot be deemed as having no effect), a fault tree would need to be developed separately to feed the probabilities in the component model.

# 4 FTA Implementation in SOTERIA

We developed functions to model fault trees and do fault tree analysis. Although developing algorithms to do this type of analysis is not an innovation in itself, since there are COTS tools that can do fault tree probability calculations, determine fault tree minimal cut sets, etc., we still needed to develop these functionalities so that we can apply them to our modeling construct and architecture synthesis. This section includes some material in our paper (Manolios, Siu, Noorman, & Liao, 2017) submitted for publication.

We defined a new OCaml variant data type called `ftree`. Similar to the construct of a search tree, `ftree` is either a node or a leaf. A node is of an operator, either a sum or a product, and a list of `ftree`s and a leaf contains the failure rate, Lambda, and the exposure time, Tau, modeled as floats. This variant data type can handle fault trees of any size.

We added code for going from fault trees to propositional formulas. The code symbolically manipulates the propositional formulas. This includes code to do simplification using Boolean logic rules shown in SAE ARP4761 (S18, 1996) (A+A=A, A*A=A, A+AK=A, AAK=AK) to aid in the construction of minimal cut sets. In addition, there is code for distributing conjunction (disjunction) over disjunction (conjunction), constant propagation, flattening formulas, removing redundant variables, simplifying conjunctions and disjunctions containing one variable and determining if a formula is in a sum of products (with no further structure). We also defined functions to compute approximate (Lambda * Tau) and exact (1 - e^(-Lambda * Tau)) probability of failure calculations, as per SAE ARP4761, directly on the fault trees.

Finally, we added code for generating cut sets from a fault tree formula. The code uses the symbolic manipulation code mentioned above. Getting the minimal cut set is important because if primary events occur more than once in a fault tree the probability calculated for the undesired top-level event will be incorrect. (It will either be greater than or less than the actual probability because the primary event will be doubly accounted for.)

We also compute the probabilities for each cut set, generate an importance metric for each cut set, and sort the cut sets by importance. The importance metric is defined as the ratio of the failure probability for the cut set divided by the top-level failure probability.

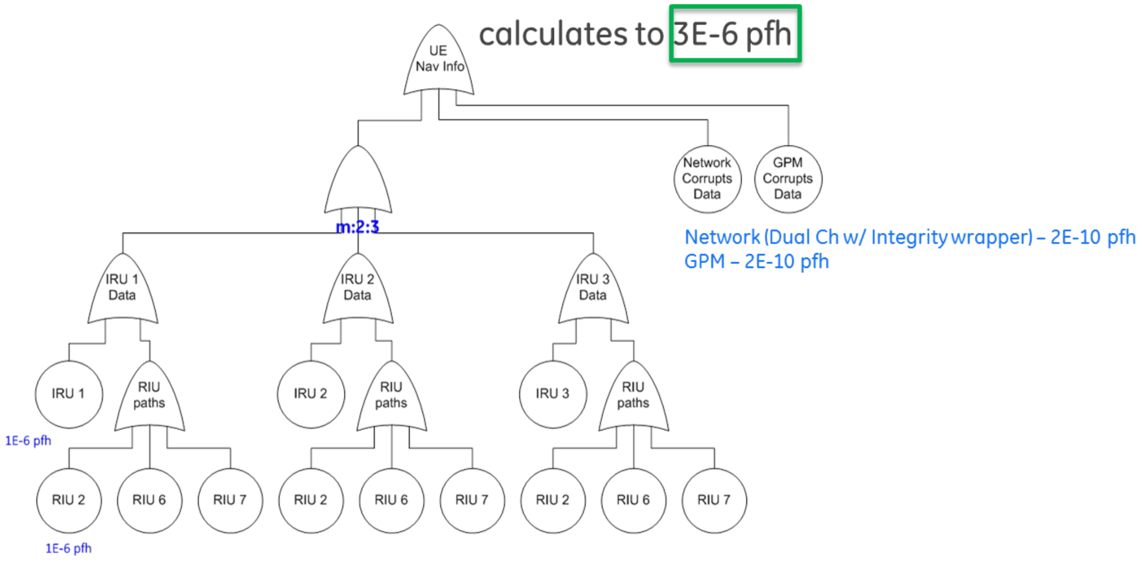
## 4.1 Verified against existing tools

We verified our SOTERIA tool against two existing tools: 1) OSATE which is an open-source AADL modeling tool with its fault tree generation output interpreted and analyzed using OpenFTA, and 2) Windchill FTA (formerly Relex Fault Tree), which is a commercial fault tree modeling tool used in industry. We identified 5 simple, yet interesting examples, in that they all have multiple layers and have repeated events.

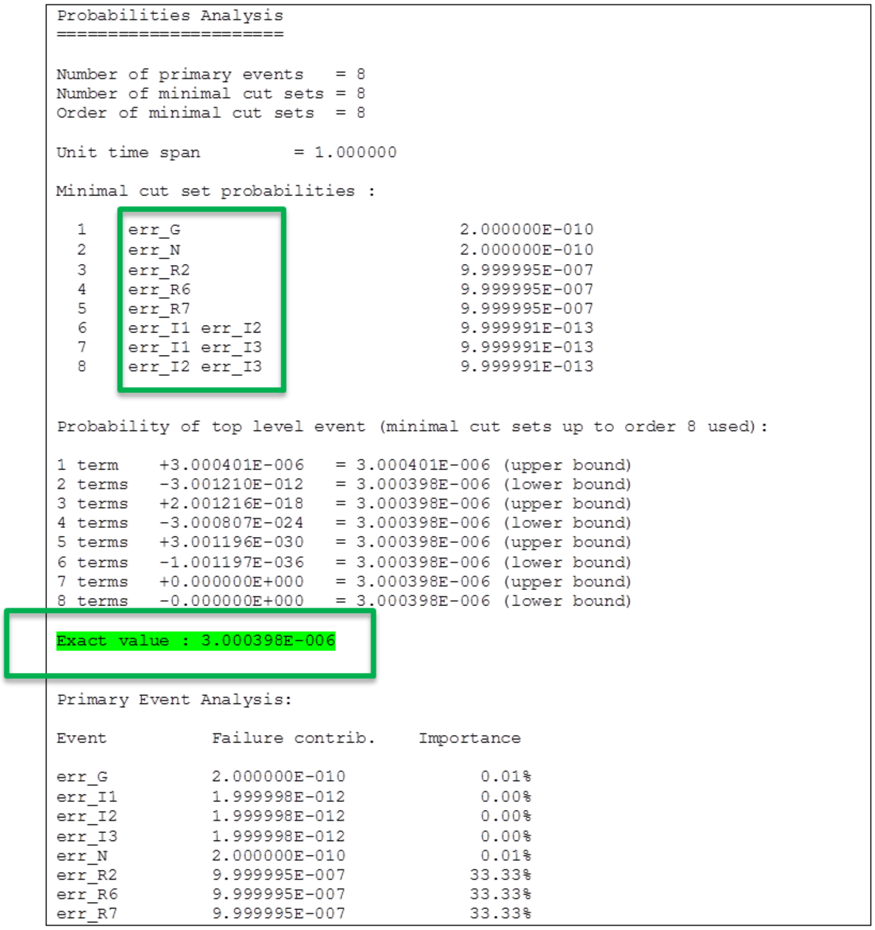
| | |
|---|---|
| Example 1) | a sample IMA architecture |
| Example 2) | Fault Tree Handbook (U.S. Nuclear Regulatory Commission, 1981), Page VIII-13 |
| Example 3) | Fault Tree Handbook (U.S. Nuclear Regulatory Commission, 1981), Page XI-3 |
| Example 4) | Online course material (Fault Tree Analysis: A tutorial, pdf., 1999), slide 88 |
| Example 5) | Reference material for a graduate course at RPI – Statistical Methods for Reliability Engineering (Tutorial - Fault Tree Analysis, 1998), slides 36-37 |

We built the fault trees in OSATE, Windchill FTA, and OCaml and compared their analysis results. We verified that our functions computed cut sets and the top-level failure probabilities that were in agreement with the outputs from OSATE/OpenFTA and Windchill FTA. Below are the results of the comparisons.

### 4.1.1 Example 1 – Sample IMA Architecture

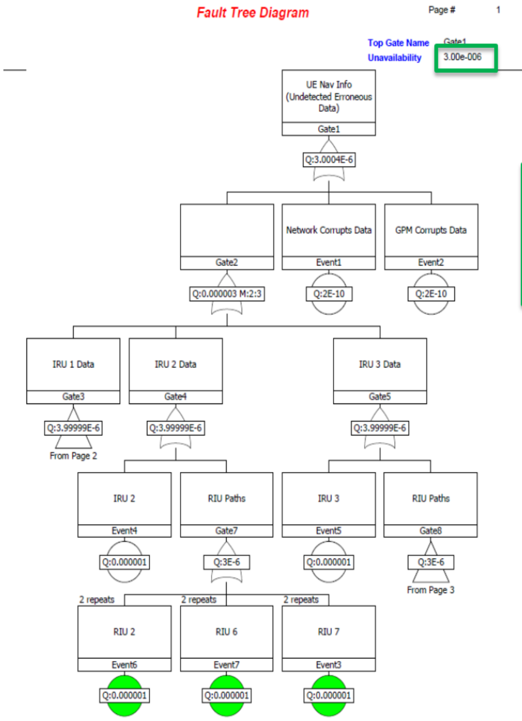


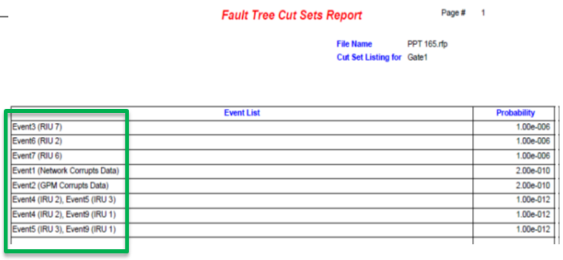
Here are the results from OSATE with OpenFTA.



```
Probabilities Analysis
======================

Number of primary events   = 8
Number of minimal cut sets = 8
Order of minimal cut sets  = 8

Unit time span         = 1.000000

Minimal cut set probabilities :

  1     err_G                               2.000000E-010
  2     err_N                               2.000000E-010
  3     err_R2                              9.999995E-007
  4     err_R6                              9.999995E-007
  5     err_R7                              9.999995E-007
  6     err_I1 err_I2                       9.999991E-013
  7     err_I1 err_I3                       9.999991E-013
  8     err_I2 err_I3                       9.999991E-013


Probability of top level event (minimal cut sets up to order 8 used):

1 term    +3.000401E-006   = 3.000401E-006 (upper bound)
2 terms   -3.001210E-012   = 3.000398E-006 (lower bound)
3 terms   +2.001216E-018   = 3.000398E-006 (upper bound)
4 terms   -3.000807E-024   = 3.000398E-006 (lower bound)
5 terms   +3.001196E-030   = 3.000398E-006 (upper bound)
6 terms   -1.001197E-036   = 3.000398E-006 (lower bound)
7 terms   +0.000000E+000   = 3.000398E-006 (upper bound)
8 terms   -0.000000E+000   = 3.000398E-006 (lower bound)

Exact value : 3.000398E-006


Primary Event Analysis:

Event          Failure contrib.    Importance

err_G          2.000000E-010             0.01%
err_I1         1.999998E-012             0.00%
err_I2         1.999998E-012             0.00%
err_I3         1.999998E-012             0.00%
err_N          2.000000E-010             0.01%
err_R2         9.999995E-007            33.33%
err_R6         9.999995E-007            33.33%
err_R7         9.999995E-007            33.33%
```

Here are the results using WindChill FTA.



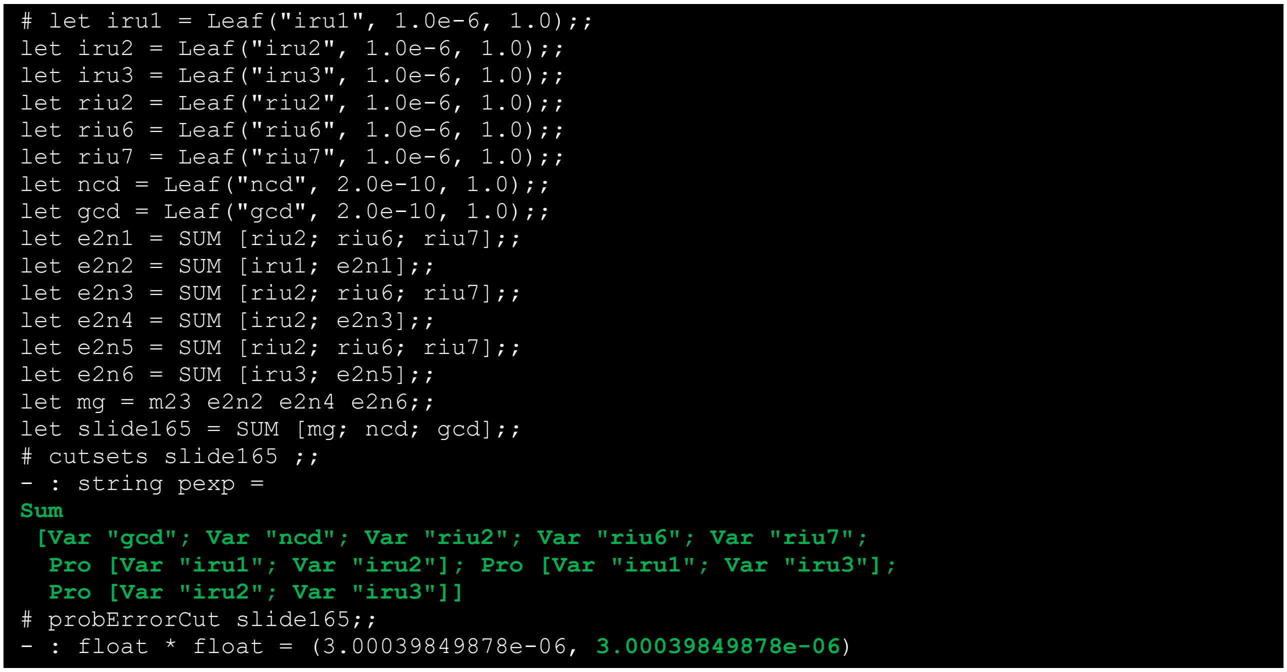


Fault Tree Cut Sets Report Page # 1

File Name PPT 165.rfp
Cut Set Listing for Gate1

| Event List | Probability |
|---|---|
| Event3 (RIU 7) | 1.00e-006 |
| Event6 (RIU 2) | 1.00e-006 |
| Event7 (RIU 6) | 1.00e-006 |
| Event1 (Network Corrupts Data) | 2.00e-010 |
| Event2 (GPM Corrupts Data) | 2.00e-010 |
| Event4 (IRU 2), Event5 (IRU 3) | 1.00e-012 |
| Event4 (IRU 2), Event9 (IRU 1) | 1.00e-012 |
| Event5 (IRU 3), Event9 (IRU 1) | 1.00e-012 |

Here are the results from our function. We confirm that we calculate the same results.

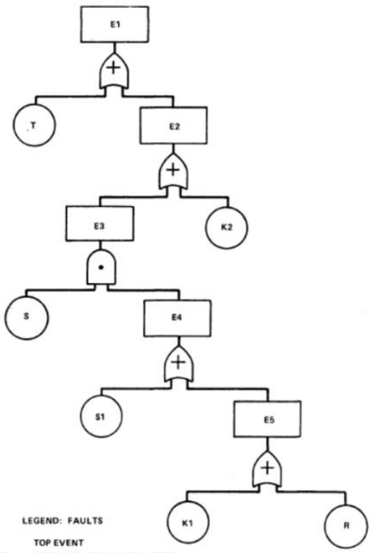


```
# let iru1 = Leaf("iru1", 1.0e-6, 1.0);;
let iru2 = Leaf("iru2", 1.0e-6, 1.0);;
let iru3 = Leaf("iru3", 1.0e-6, 1.0);;
let riu2 = Leaf("riu2", 1.0e-6, 1.0);;
let riu6 = Leaf("riu6", 1.0e-6, 1.0);;
let riu7 = Leaf("riu7", 1.0e-6, 1.0);;
let ncd = Leaf("ncd", 2.0e-10, 1.0);;
let gcd = Leaf("gcd", 2.0e-10, 1.0);;
let e2n1 = SUM [riu2; riu6; riu7];;
let e2n2 = SUM [iru1; e2n1];;
let e2n3 = SUM [riu2; riu6; riu7];;
let e2n4 = SUM [iru2; e2n3];;
let e2n5 = SUM [riu2; riu6; riu7];;
let e2n6 = SUM [iru3; e2n5];;
let mg = m23 e2n2 e2n4 e2n6;;
let slide165 = SUM [mg; ncd; gcd];;
# cutsets slide165 ;;
- : string pexp =
Sum
 [Var "gcd"; Var "ncd"; Var "riu2"; Var "riu6"; Var "riu7";
  Pro [Var "iru1"; Var "iru2"]; Pro [Var "iru1"; Var "iru3"];
  Pro [Var "iru2"; Var "iru3"]]
# probErrorCut slide165;;
- : float * float = (3.00039849878e-06, 3.00039849878e-06)
```

### 4.1.2 Example 2 – Fault Tree Handbook (NUREG-0492) Pressure Tank Example

PRESSURE TANK EXAMPLE

VIII-13



LEGEND: FAULTS

| | |
|---|---|
| E1 | TOP EVENT |
| E2, E3, E4, E5 | INTERMEDIATE FAULT EVENTS |
| R | PRIMARY FAILURE OF TIMER RELAY |
| S | PRIMARY FAILURE OF PRESSURE SWITCH |
| S1 | PRIMARY FAILURE OF SWITCH S1 |
| K1 | PRIMARY FAILURE OF RELAY K1 |
| K2 | PRIMARY FAILURE OF RELAY K2 |
| T | PRIMARY FAILURE OF PRESSURE TANK |

Figure VIII-14. Basic (Reduced) Fault Tree for Pressure Tank Example

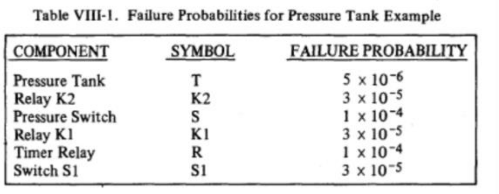
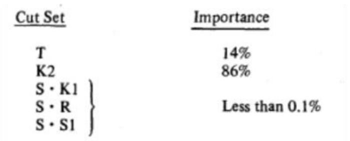


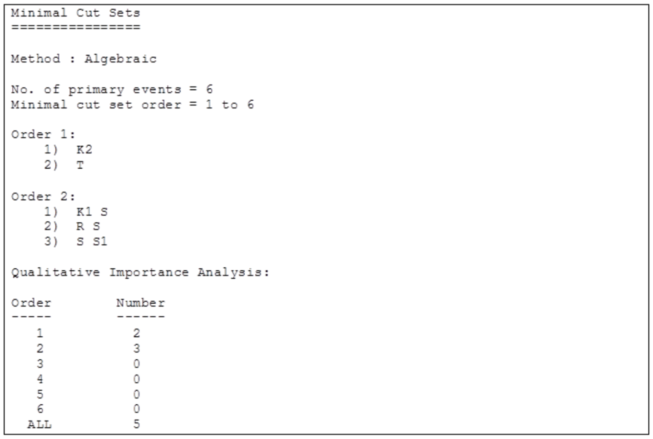
Table VIII-1. Failure Probabilities for Pressure Tank Example

| COMPONENT | SYMBOL | FAILURE PROBABILITY |
|---|---|---|
| Pressure Tank | T | $5 \times 10^{-6}$ |
| Relay K2 | K2 | $3 \times 10^{-5}$ |
| Pressure Switch | S | $1 \times 10^{-4}$ |
| Relay K1 | K1 | $3 \times 10^{-5}$ |
| Timer Relay | R | $1 \times 10^{-4}$ |
| Switch S1 | S1 | $3 \times 10^{-5}$ |

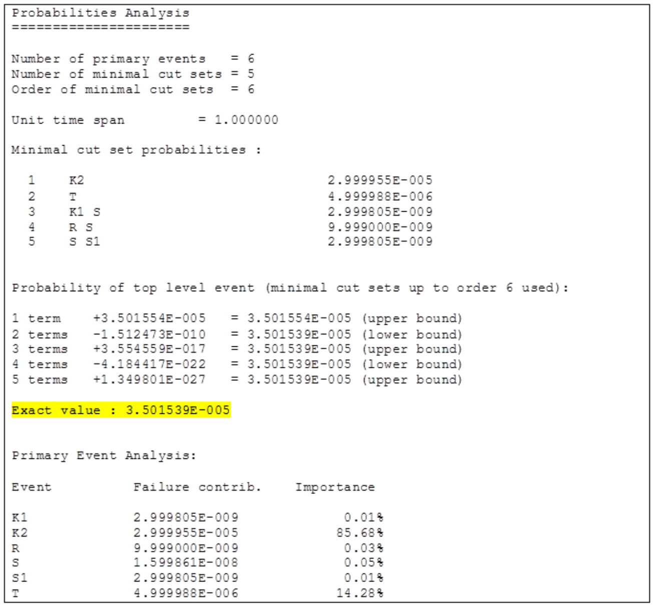


| Cut Set | Importance |
|---|---|
| T | 14% |
| K2 | 86% |
| S • K1 | Less than 0.1% |
| S • R | |
| S • S1 | |

Here’s the output from OSATE with OpenFTA.

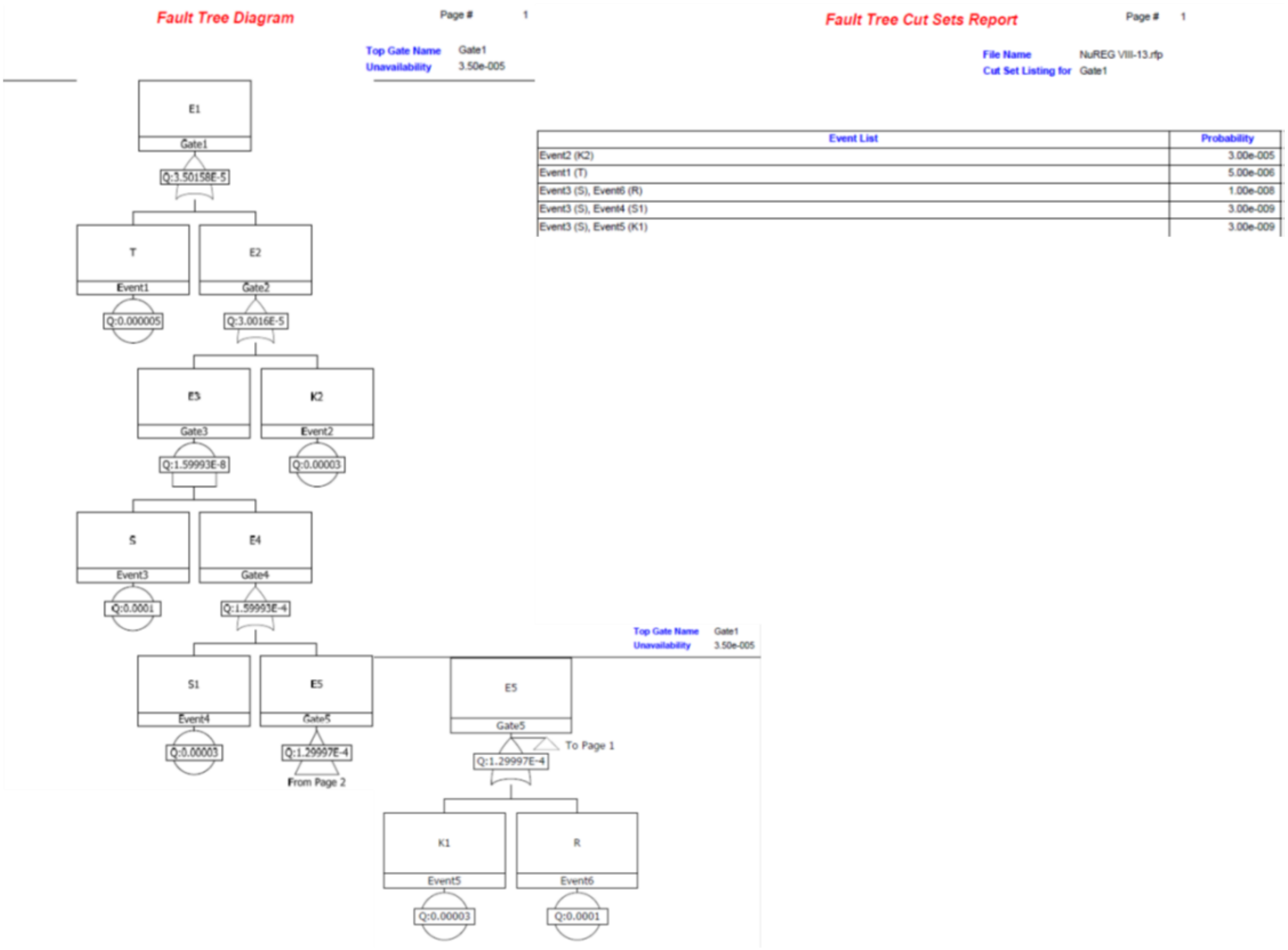


```
Minimal Cut Sets
================

Method : Algebraic

No. of primary events = 6
Minimal cut set order = 1 to 6

Order 1:
    1)  K2
    2)  T

Order 2:
    1)  K1 S
    2)  R S
    3)  S S1

Qualitative Importance Analysis:

Order        Number
-----        ------
   1           2
   2           3
   3           0
   4           0
   5           0
   6           0
  ALL          5
```



```
Probabilities Analysis
======================

Number of primary events   = 6
Number of minimal cut sets = 5
Order of minimal cut sets  = 6

Unit time span         = 1.000000

Minimal cut set probabilities :

  1    K2                              2.999955E-005
  2    T                               4.999988E-006
  3    K1 S                            2.999805E-009
  4    R S                             9.999000E-009
  5    S S1                            2.999805E-009


Probability of top level event (minimal cut sets up to order 6 used):

1 term    +3.501554E-005   = 3.501554E-005 (upper bound)
2 terms   -1.512473E-010   = 3.501539E-005 (lower bound)
3 terms   +3.554559E-017   = 3.501539E-005 (upper bound)
4 terms   -4.184417E-022   = 3.501539E-005 (lower bound)
5 terms   +1.349801E-027   = 3.501539E-005 (upper bound)

Exact value : 3.501539E-005


Primary Event Analysis:

Event          Failure contrib.    Importance

K1             2.999805E-009           0.01%
K2             2.999955E-005          85.68%
R              9.999000E-009           0.03%
S              1.599861E-008           0.05%
S1             2.999805E-009           0.01%
T              4.999988E-006          14.28%
```

Here’s the output from Windchill FTA.



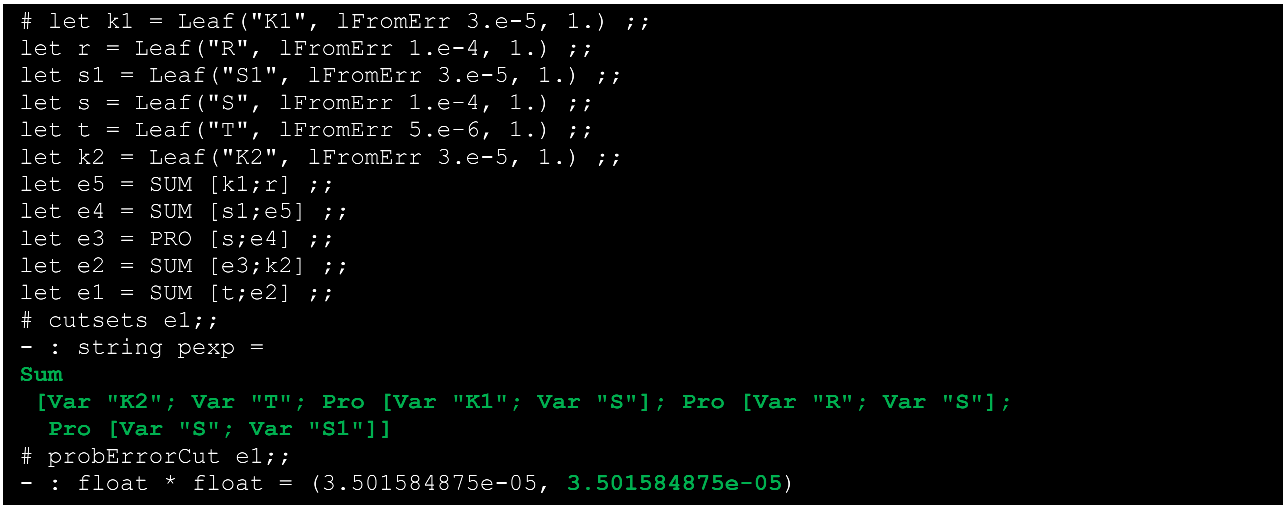
Fault Tree Cut Sets Report

Page # 1

File Name NuREG VIII-13.rfp
Cut Set Listing for Gate1

| Event List | Probability |
|---|---|
| Event2 (K2) | 3.00e-005 |
| Event1 (T) | 5.00e-006 |
| Event3 (S), Event6 (R) | 1.00e-008 |
| Event3 (S), Event4 (S1) | 3.00e-009 |
| Event3 (S), Event5 (K1) | 3.00e-009 |

Here is the output from our functions. We confirm that we match the output from the handbook.

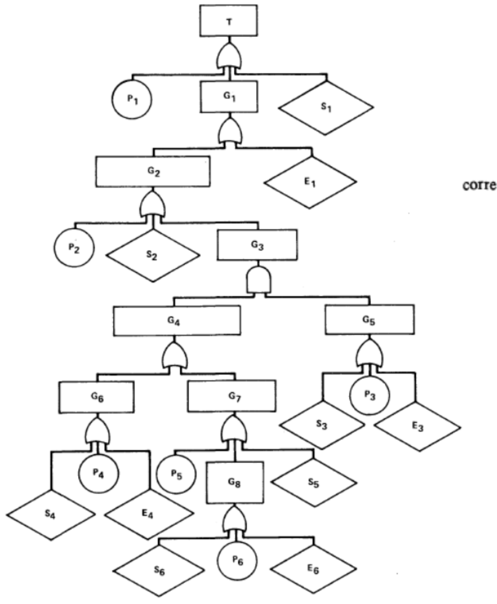


```
# let k1 = Leaf("K1", lFromErr 3.e-5, 1.) ;;
let r = Leaf("R", lFromErr 1.e-4, 1.) ;;
let s1 = Leaf("S1", lFromErr 3.e-5, 1.) ;;
let s = Leaf("S", lFromErr 1.e-4, 1.) ;;
let t = Leaf("T", lFromErr 5.e-6, 1.) ;;
let k2 = Leaf("K2", lFromErr 3.e-5, 1.) ;;
let e5 = SUM [k1;r] ;;
let e4 = SUM [s1;e5] ;;
let e3 = PRO [s;e4] ;;
let e2 = SUM [e3;k2] ;;
let e1 = SUM [t;e2] ;;
# cutsets e1;;
- : string pexp =
Sum
 [Var "K2"; Var "T"; Pro [Var "K1"; Var "S"]; Pro [Var "R"; Var "S"];
  Pro [Var "S"; Var "S1"]]
# probErrorCut e1;;
- : float * float = (3.501584875e-05, 3.501584875e-05)
```

### 4.1.3 Example 3 – Fault Tree Handbook (NUREG-0492) An evaluation example

This example did not include probabilities, so we used arbitrary numbers for testing.

FAULT TREE EVALUATION TECHNIQUES XI-3

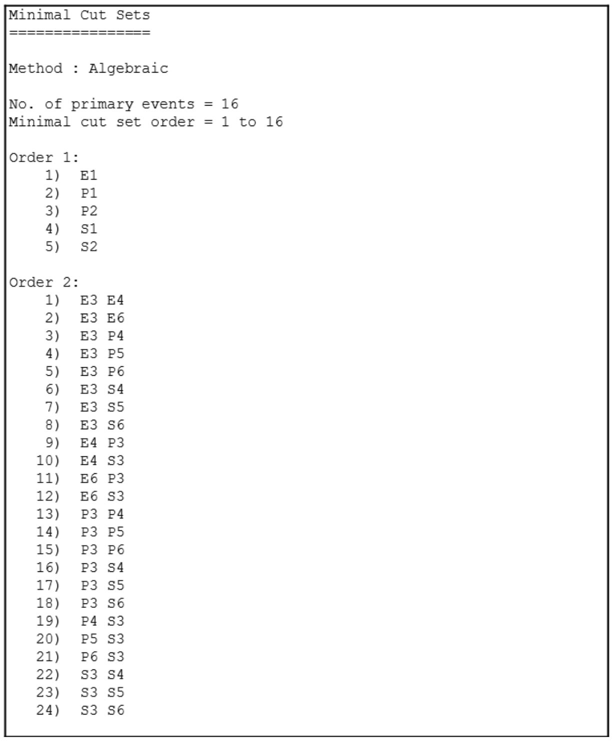


correct minimal cut set form.

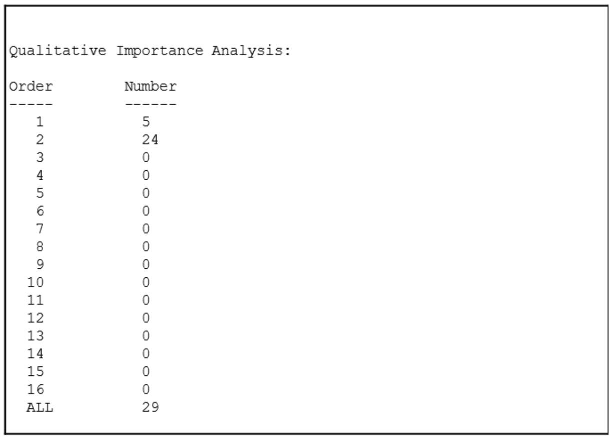
$$\begin{aligned} T = {} & P_1 + S_1 + E_1 + P_2 + S_2 \\ & + (P_4 \cdot P_3) + (P_4 \cdot S_3) + (P_4 : E_3) \\ & + (S_4 \cdot P_3) + (S_4 \cdot S_3) + (S_4 \cdot E_3) \\ & + (E_4 \cdot P_3) + (E_4 \cdot S_3) + (E_4 \cdot E_3) \\ & + (P_5 \cdot P_3) + (P_5 \cdot S_3) + (P_5 \cdot E_3) \\ & + (S_5 \cdot P_3) + (S_5 \cdot S_3) + (S_5 \cdot E_3) \\ & + (P_6 \cdot P_3) + (P_6 \cdot S_3) + (P_6 \cdot E_3) \\ & + (S_6 \cdot P_3) + (S_6 \cdot S_3) + (S_6 \cdot E_3) \\ & + (E_6 \cdot P_3) + (E_6 \cdot S_3) + (E_6 \cdot E_3). \end{aligned}$$

Figure IX-1. Pressure Tank Fault Tree

Here are the results using OSATE with OpenFTA.

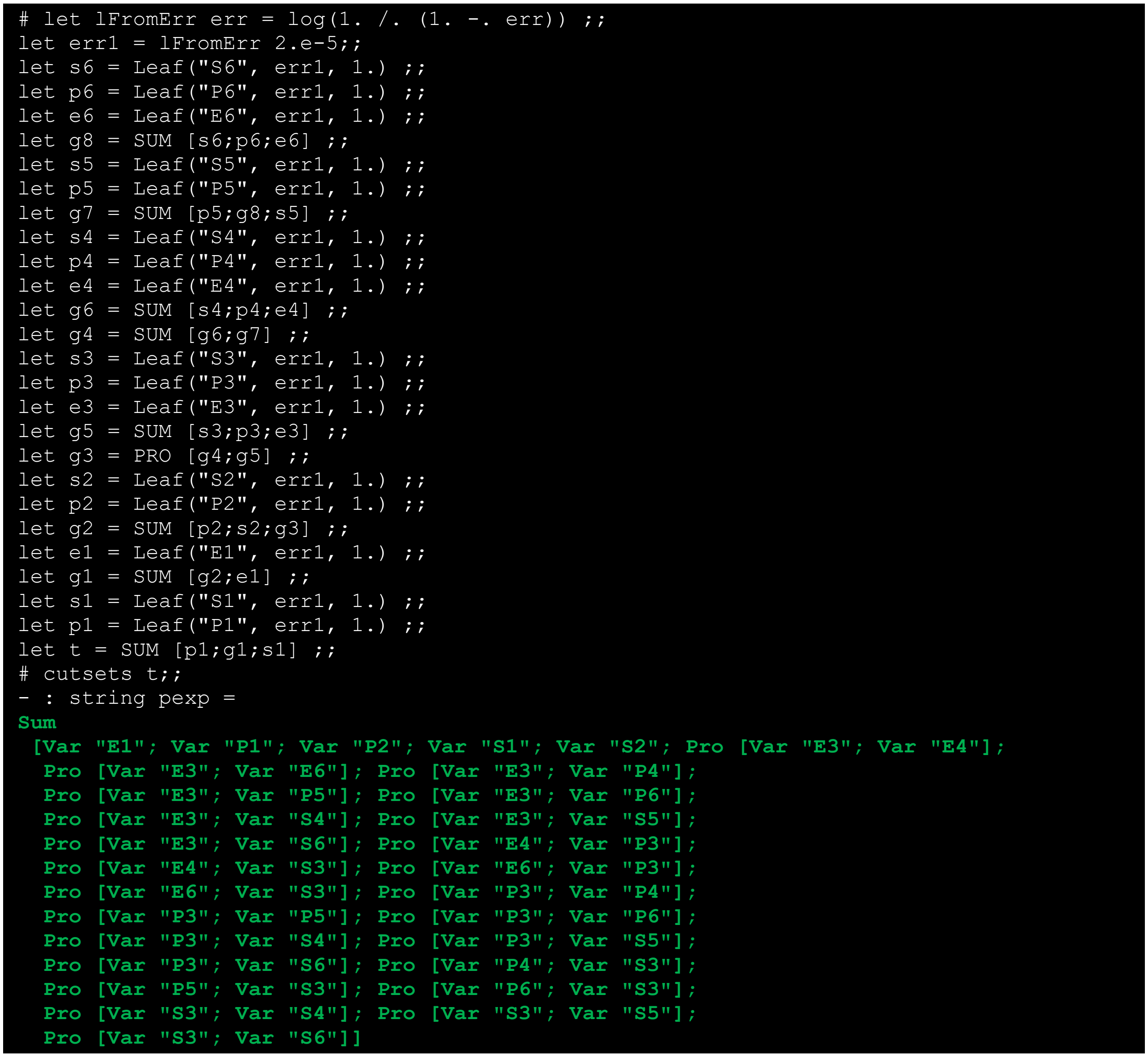


```
Minimal Cut Sets
================

Method : Algebraic

No. of primary events = 16
Minimal cut set order = 1 to 16

Order 1:
    1)  E1
    2)  P1
    3)  P2
    4)  S1
    5)  S2

Order 2:
    1)  E3 E4
    2)  E3 E6
    3)  E3 P4
    4)  E3 P5
    5)  E3 P6
    6)  E3 S4
    7)  E3 S5
    8)  E3 S6
    9)  E4 P3
   10)  E4 S3
   11)  E6 P3
   12)  E6 S3
   13)  P3 P4
   14)  P3 P5
   15)  P3 P6
   16)  P3 S4
   17)  P3 S5
   18)  P3 S6
   19)  P4 S3
   20)  P5 S3
   21)  P6 S3
   22)  S3 S4
   23)  S3 S5
   24)  S3 S6
```



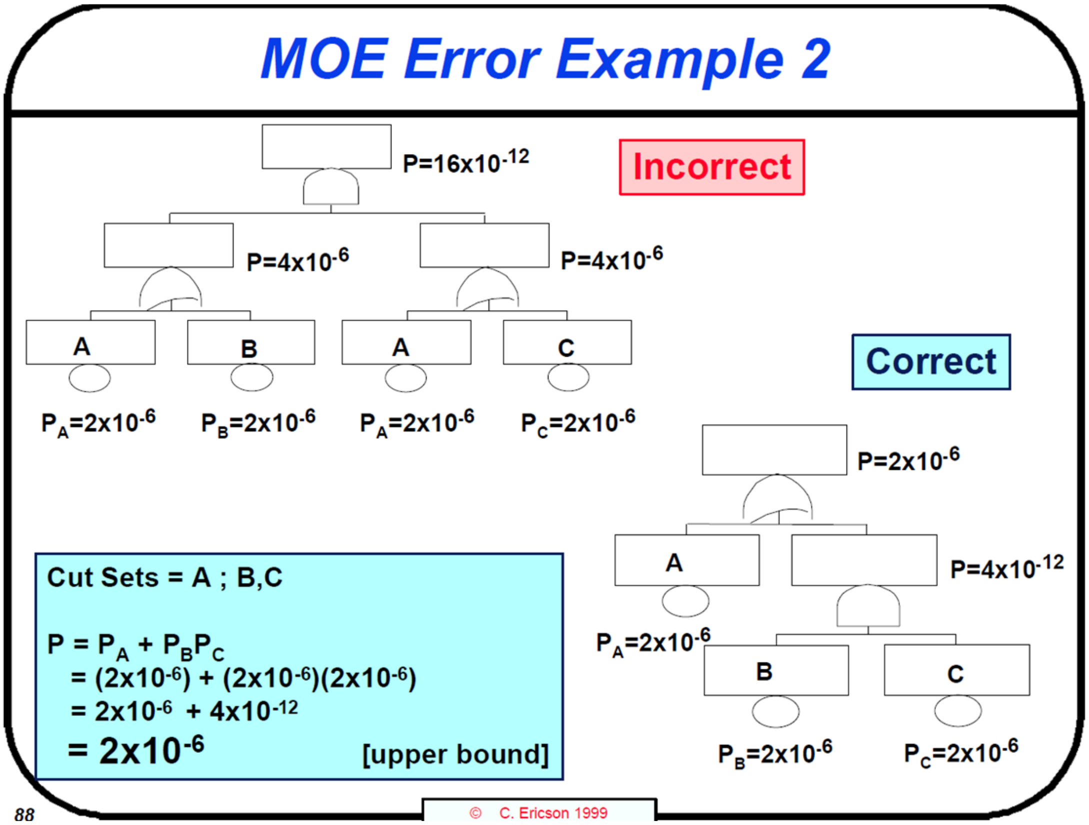
```
Qualitative Importance Analysis:

Order        Number
-----        ------
  1            5
  2            24
  3            0
  4            0
  5            0
  6            0
  7            0
  8            0
  9            0
 10            0
 11            0
 12            0
 13            0
 14            0
 15            0
 16            0
 ALL           29
```

Here are the results from our FTA implementation.



```
# let lFromErr err = log(1. /. (1. -. err)) ;;
let err1 = lFromErr 2.e-5;;
let s6 = Leaf("S6", err1, 1.) ;;
let p6 = Leaf("P6", err1, 1.) ;;
let e6 = Leaf("E6", err1, 1.) ;;
let g8 = SUM [s6;p6;e6] ;;
let s5 = Leaf("S5", err1, 1.) ;;
let p5 = Leaf("P5", err1, 1.) ;;
let g7 = SUM [p5;g8;s5] ;;
let s4 = Leaf("S4", err1, 1.) ;;
let p4 = Leaf("P4", err1, 1.) ;;
let e4 = Leaf("E4", err1, 1.) ;;
let g6 = SUM [s4;p4;e4] ;;
let g4 = SUM [g6;g7] ;;
let s3 = Leaf("S3", err1, 1.) ;;
let p3 = Leaf("P3", err1, 1.) ;;
let e3 = Leaf("E3", err1, 1.) ;;
let g5 = SUM [s3;p3;e3] ;;
let g3 = PRO [g4;g5] ;;
let s2 = Leaf("S2", err1, 1.) ;;
let p2 = Leaf("P2", err1, 1.) ;;
let g2 = SUM [p2;s2;g3] ;;
let e1 = Leaf("E1", err1, 1.) ;;
let g1 = SUM [g2;e1] ;;
let s1 = Leaf("S1", err1, 1.) ;;
let p1 = Leaf("P1", err1, 1.) ;;
let t = SUM [p1;g1;s1] ;;
# cutsets t;;
- : string pexp =
Sum
 [Var "E1"; Var "P1"; Var "P2"; Var "S1"; Var "S2"; Pro [Var "E3"; Var "E4"];
  Pro [Var "E3"; Var "E6"]; Pro [Var "E3"; Var "P4"];
  Pro [Var "E3"; Var "P5"]; Pro [Var "E3"; Var "P6"];
  Pro [Var "E3"; Var "S4"]; Pro [Var "E3"; Var "S5"];
  Pro [Var "E3"; Var "S6"]; Pro [Var "E4"; Var "P3"];
  Pro [Var "E4"; Var "S3"]; Pro [Var "E6"; Var "P3"];
  Pro [Var "E6"; Var "S3"]; Pro [Var "P3"; Var "P4"];
  Pro [Var "P3"; Var "P5"]; Pro [Var "P3"; Var "P6"];
  Pro [Var "P3"; Var "S4"]; Pro [Var "P3"; Var "S5"];
  Pro [Var "P3"; Var "S6"]; Pro [Var "P4"; Var "S3"];
  Pro [Var "P5"; Var "S3"]; Pro [Var "P6"; Var "S3"];
  Pro [Var "S3"; Var "S4"]; Pro [Var "S3"; Var "S5"];
  Pro [Var "S3"; Var "S6"]]
```

### 4.1.4 Example 4 – from www.thecourse-pm.com



MOE Error Example 2

Cut Sets = A ; B,C

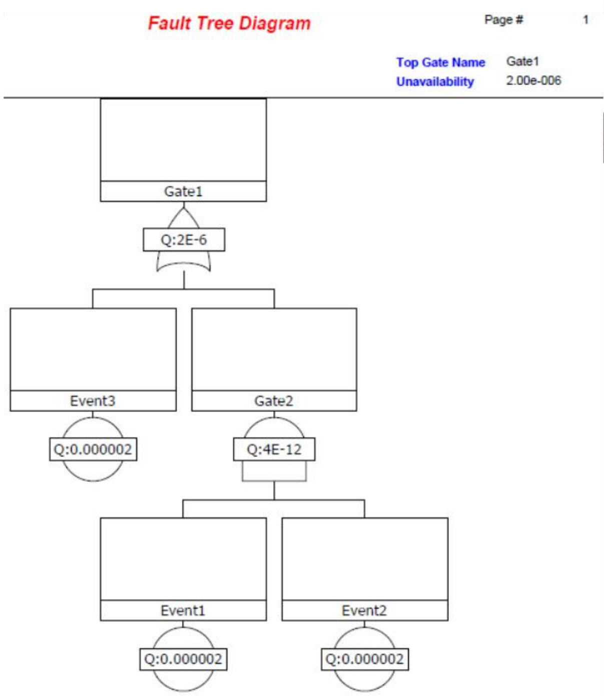
$$
\begin{aligned}
P &= P_A + P_B P_C \\
&= (2\times10^{-6}) + (2\times10^{-6})(2\times10^{-6}) \\
&= 2\times10^{-6} + 4\times10^{-12} \\
&= 2\times10^{-6}
\end{aligned}
$$

[upper bound]

88

© C. Ericson 1999

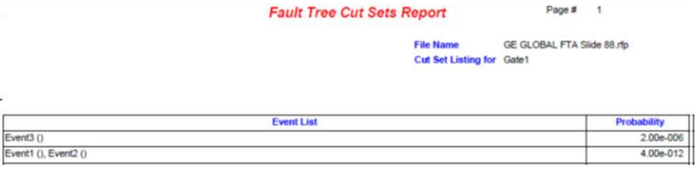
Here are the results from Windchill FTA.



Fault Tree Diagram

Page # 1

Top Gate Name Gate1
Unavailability 2.00e-006



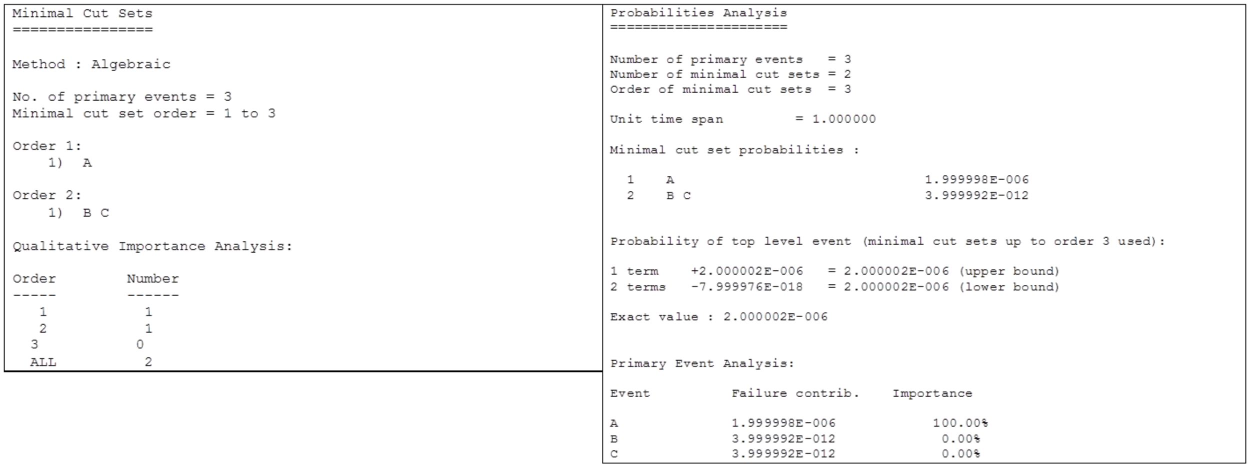
Fault Tree Cut Sets Report

Page # 1

File Name GE GLOBAL FTA Slide 88.rfp
Cut Set Listing for Gate1

| Event List | Probability |
|---|---|
| Event3 () | 2.00e-006 |
| Event1 (), Event2 () | 4.00e-012 |

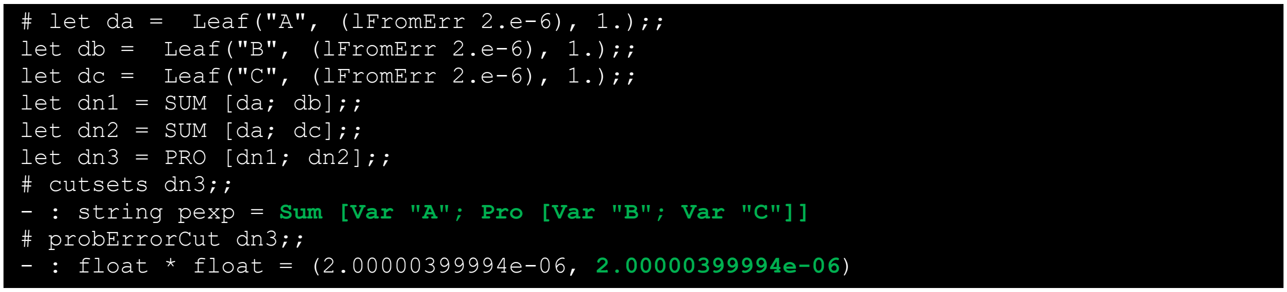
Here are the results from OSATE with OpenFTA.



```
Minimal Cut Sets
================

Method : Algebraic

No. of primary events = 3
Minimal cut set order = 1 to 3

Order 1:
    1)  A

Order 2:
    1)  B C

Qualitative Importance Analysis:

Order        Number
-----        ------
   1            1
   2            1
  3            0
  ALL           2
```

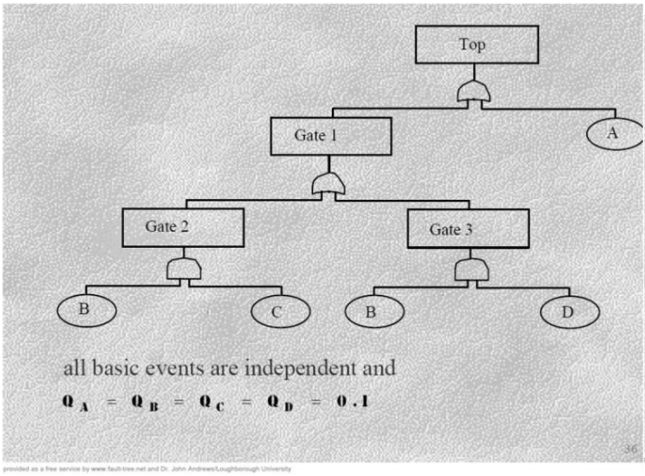
```
Probabilities Analysis
======================

Number of primary events   = 3
Number of minimal cut sets = 2
Order of minimal cut sets  = 3

Unit time span         = 1.000000

Minimal cut set probabilities :

  1    A                                 1.999998E-006
  2    B C                               3.999992E-012


Probability of top level event (minimal cut sets up to order 3 used):

1 term    +2.000002E-006   = 2.000002E-006 (upper bound)
2 terms   -7.999976E-018   = 2.000002E-006 (lower bound)

Exact value : 2.000002E-006


Primary Event Analysis:

Event          Failure contrib.    Importance

A              1.999998E-006           100.00%
B              3.999992E-012             0.00%
C              3.999992E-012             0.00%
```

Here are the results from SOTERIA.



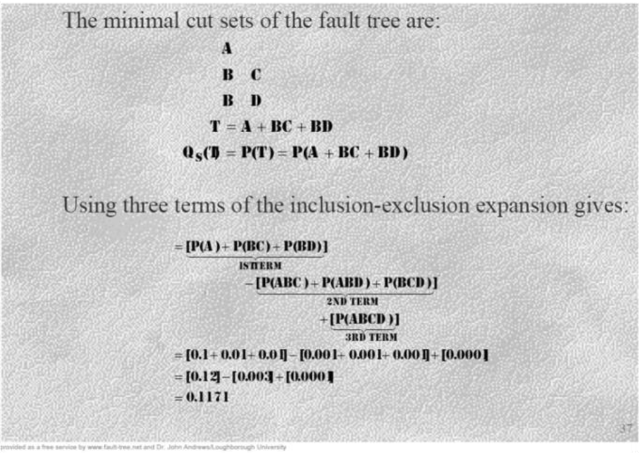
```
# let da =  Leaf("A", (lFromErr 2.e-6), 1.);;
let db =  Leaf("B", (lFromErr 2.e-6), 1.);;
let dc =  Leaf("C", (lFromErr 2.e-6), 1.);;
let dn1 = SUM [da; db];;
let dn2 = SUM [da; dc];;
let dn3 = PRO [dn1; dn2];;
# cutsets dn3;;
- : string pexp = Sum [Var "A"; Pro [Var "B"; Var "C"]]
# probErrorCut dn3;;
- : float * float = (2.00000399994e-06, 2.00000399994e-06)
```

### 4.1.5 Example 5 – from lecture notes


Top
Gate 1
A
Gate 2
Gate 3
B
C
B
D
36
provided as a free service by www.fault-tree.net and Dr. John Andrews/Loughborough University

all basic events are independent and

$$Q_A = Q_B = Q_C = Q_D = 0.1$$



The minimal cut sets of the fault tree are:

**A**

**B C**

**B D**

$$T = A + BC + BD$$

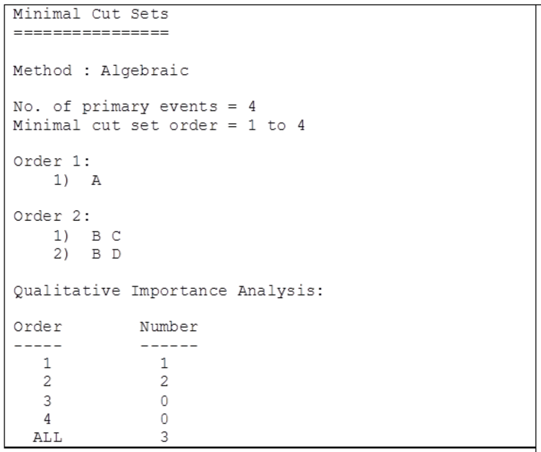
$$Q_S(T) = P(T) = P(A + BC + BD)$$

Using three terms of the inclusion-exclusion expansion gives:

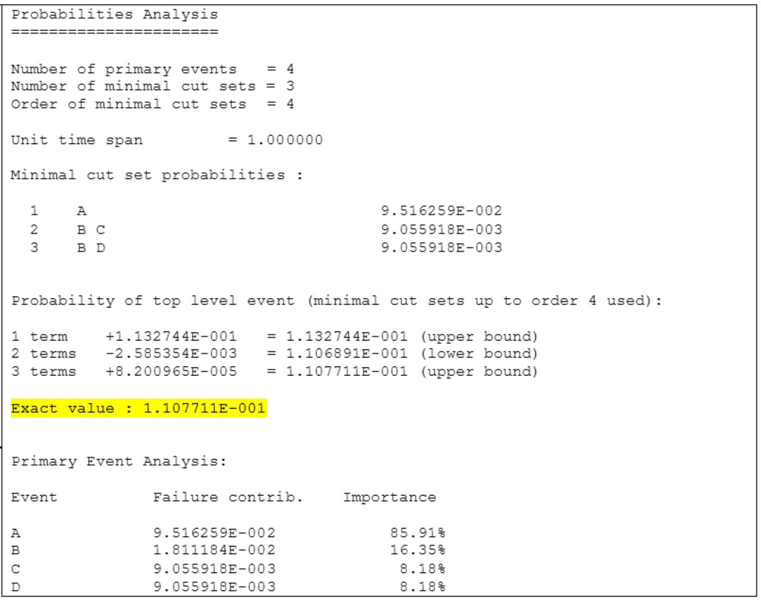
$$= \underbrace{[P(A) + P(BC) + P(BD)]}_{\text{1STTERM}}$$
$$- \underbrace{[P(ABC) + P(ABD) + P(BCD)]}_{\text{2ND TERM}}$$
$$+ \underbrace{[P(ABCD)]}_{\text{3RD TERM}}$$
$$= [0.1 + 0.01 + 0.01] - [0.001 + 0.001 + 0.001] + [0.0001]$$
$$= [0.12] - [0.003] + [0.0001]$$
$$= 0.1171$$

37
provided as a free service by www.fault-tree.net and Dr. John Andrews/Loughborough University

Here are results from OSATE and OpenFTA.

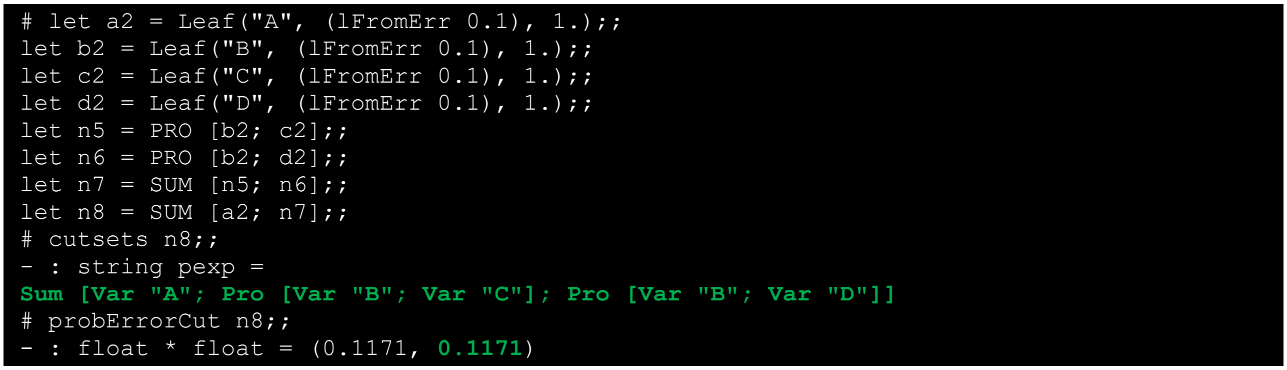


```
Minimal Cut Sets
================

Method : Algebraic

No. of primary events = 4
Minimal cut set order = 1 to 4

Order 1:
    1)  A

Order 2:
    1)  B C
    2)  B D

Qualitative Importance Analysis:

Order        Number
-----        ------
   1            1
   2            2
   3            0
   4            0
  ALL           3
```



```
Probabilities Analysis
======================

Number of primary events   = 4
Number of minimal cut sets = 3
Order of minimal cut sets  = 4

Unit time span         = 1.000000

Minimal cut set probabilities :

  1    A                                    9.516259E-002
  2    B C                                  9.055918E-003
  3    B D                                  9.055918E-003


Probability of top level event (minimal cut sets up to order 4 used):

1 term    +1.132744E-001   = 1.132744E-001 (upper bound)
2 terms   -2.585354E-003   = 1.106891E-001 (lower bound)
3 terms   +8.200965E-005   = 1.107711E-001 (upper bound)

Exact value : 1.107711E-001


Primary Event Analysis:

Event          Failure contrib.    Importance

A              9.516259E-002          85.91%
B              1.811184E-002          16.35%
C              9.055918E-003           8.18%
D              9.055918E-003           8.18%
```

Here are results from SOTERIA.



```
# let a2 = Leaf("A", (lFromErr 0.1), 1.);;
let b2 = Leaf("B", (lFromErr 0.1), 1.);;
let c2 = Leaf("C", (lFromErr 0.1), 1.);;
let d2 = Leaf("D", (lFromErr 0.1), 1.);;
let n5 = PRO [b2; c2];;
let n6 = PRO [b2; d2];;
let n7 = SUM [n5; n6];;
let n8 = SUM [a2; n7];;
# cutsets n8;;
- : string pexp =
Sum [Var "A"; Pro [Var "B"; Var "C"]; Pro [Var "B"; Var "D"]]
# probErrorCut n8;;
- : float * float = (0.1171, 0.1171)
```

## 4.2 Limitations of OSATE and OpenFTA

OSATE is an open-source AADL modeling tool with its fault tree output interpreted and analyzed using OpenFTA. In constructing the models in OSATE and evaluating the fault tree analysis output in OpenFTA, we uncovered some limitations.

- OSATE cannot handle repeated events. When modeling the composite error behavior, even when repeated events are modeled as such, the tool will always assign different labels to each event. We hacked the OSATE fault tree generation output file by manually replacing labels that were meant to be repeated. Only then was OpenFTA able to generate the correct cut sets.
- OSATE does not generate a very compact fault tree because its logic gates are limited to two inputs. As a result, the fault tree can be very deep and cannot be flattened.
- OpenFTA has some bugs in probability analysis, e.g., for Example 5 above, OpenFTA generated an incorrect result of top-level event probability.
- OpenFTA has some scalability issues, e.g., for Example 3 above, OpenFTA took about 40 minutes to perform the probability analysis.

- OSATE does not seem to be able to generate the fault tree analysis output without the composite error behavior model, which essentially requires the user to manually provide the fault tree logic.

Uncovering these limitations tells us that there are gaps in existing tools. Our program to come up with a technique that automatically transforms an IMA architecture model into a fault tree fills a void in the capabilities of current tools.

# 5 Fault Tree Synthesis

Fault tree synthesis advances the safety engineer's capability to perform safety analysis. As Banach & Bozaano wrote in their paper, "The manual construction of fault trees relies on the ability of the safety engineer to understand and to foresee the system behaviour. As a consequence, it is a time consuming and error-prone activity; moreover, managing the generated fault trees is challenging in case of large complex systems" (Banach & Bozzano, 2006). Also, Wang quoted in her thesis, "Although general guidelines are available for fault tree synthesis, learning how to create fault trees is akin to learning to ski: some people never quite make it" (Wang, 2004).

We conducted a literature search of papers on automatically generating fault trees from models. We found 10 well cited papers that covered a wide range of modeling techniques – di-graphs, cause-and-effect models, AADL, UML, SysML, and Matlab/Simulink. The range of modeling techniques were a result of the different domains, such as chemical processing systems, mechanical systems, P&ID (piping and instrumentation diagrams), and computer systems. Within each domain, we found different emphasis on the modeling techniques, such as the ability to handle feedback and feedforward loops for chemical processing systems. We also found that the papers spanned a wide range of dates, from 1970's to 2010's. While there was a spread in modeling techniques and domains, when it comes to synthesizing fault trees there was one similarity amongst all of these which is the need to have two separate types of models: 1) a system model that gives the physical components and their connections, and 2) fault model(s) that describe behaviors, usually on the unit component level.

We studied the literature and narrowed down to a modeling technique and fault tree construction procedure that was based on decision tables. The technique from decision tables to fault trees was not overwhelmed with features for handling domain-specific system-level attributes. Therefore, the use of decision tables was sufficient and divorced us from having to deal with any added system level complexity.

## 5.1 Decision Tables

The concept of using decision tables to construct fault trees was first introduced in a 264 page report from UCLA prepared for the Electric Power Research Institute at Palo Alto, CA in 1976 (Salem, Apostolakis, & Okent, 1976). This report introduced a means of representing component behavior via decision tables by relating each component's input(s) and internal failure(s) to its output(s). It also presented techniques for making the tables compact in order to simplify their use in constructing fault trees.  Then, the report described how to take these component models to systematically construct the fault tree, including how to pick out the top-level event and identify boundary conditions. Finally, the technical report showed, via examples, how the methodology was implemented and verified in a computer program called CAT (Computer Aided Trees).

An extension to the original technical report was published in a 1984 paper written by Contini and Squellati for the European Commission's "Nuclear Science and Technology" series of publications (Contini & Squellati, 1984). This paper addressed two limitations of the CAT method. First, it extended the decision tables to include bidirectional relationships between components. The example illustrated that without bidirectional relationships the resulting fault tree could be incomplete and miss a cut set. Another extension presented in this paper was a new algorithm for simplifying the decision tables to speed up the CAT algorithm.

A more recent paper that illustrated the use of decision tables was from 2009 by Majdara and Wakabayashi, published in the Reliability Engineering and System Safety journal (Majdara & Wakabayashi, 2009). In this paper, the authors combined ideas from various different modeling approaches. Their component based models included the use of decision tables, augmented with state transition tables and special junction and extension components. In spirit, the use of decision tables was the same as the original technical report from UCLA. The authors also published several subsequent papers using their combined modeling approach, illustrated their approach with a different example in each paper.

We will show the construction of a decision table using one of our IMA components as an illustrative example. A GPM has 1 input, an internal failure mode, and 1 output. The complete GPM decision table, listing all combinations of inputs and internal states, is shown below. The table describes undetected erroneous data failure, where 0 = no failure, 1 = undetected erroneous.

**GPM**

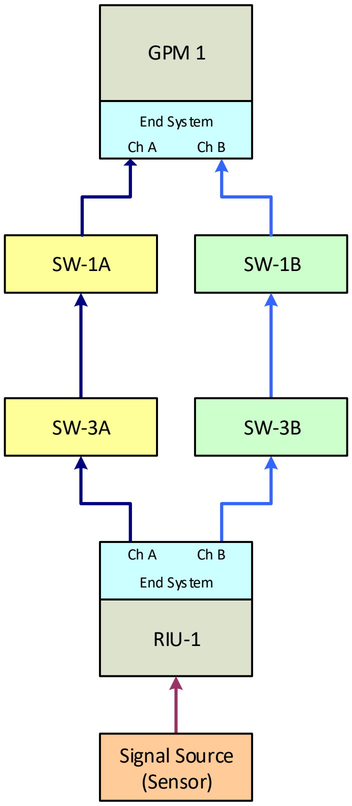
| Input = GESo | internal failure = Sgpm | Output = Go |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

This decision table could be reduced by noting that when an internal integrity failure occurs, the output is 1 regardless of the input. Similarly, when there is undetected erroneous data at the input, the output is 1 regardless of the internal failure. The reduced decision table (where 0 = no failure, 1 = undetected erroneous failure, -- = don't care) would be:

**GPM**

| Input = GESo | internal failure = Sgpm | Output = Go |
|---|---|---|
| 0 | 0 | 0 |
| -- | 1 | 1 |
| 1 | -- | 1 |

The reduced decision tables for other IMA components are shown below. The input and output names are also listed. For reference, a generic IMA architecture is illustrated below, which shows the relationship between inputs and outputs from one component to the next.



- Go : GPM output (top-level event)
- GESo : GPM End system output
- SW-1Ao : SW-1A output
- SW-1Bo : SW-1B output
- SW-3Ao : SW-3A output
- SW-3Bo : SW-3B output
- RESo : RIU End system output
- RIUo : RIU output
- Seno : Sensor output

**GPMES**

| SW-1Ao | SW-1Bo | Sges | GESo |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | -- | -- | 1 |
| -- | 1 | -- | 1 |
| -- | -- | 1 | 1 |

**SW1A**

| SW-3Ao | Ssw1a | SW-1Ao |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

**SW1B**

| SW-3Bo | Ssw1b | SW-1Bo |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

**SW3A**

| RESo | Ssw3a | SW-3Ao |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

**SW3B**

| RESo | Ssw3b | SW-3Bo |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

**RIUES**

| RIUo | Sres | RESoa | RESob |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| -- | 1 | 1 | 1 |
| 1 | -- | 1 | 1 |

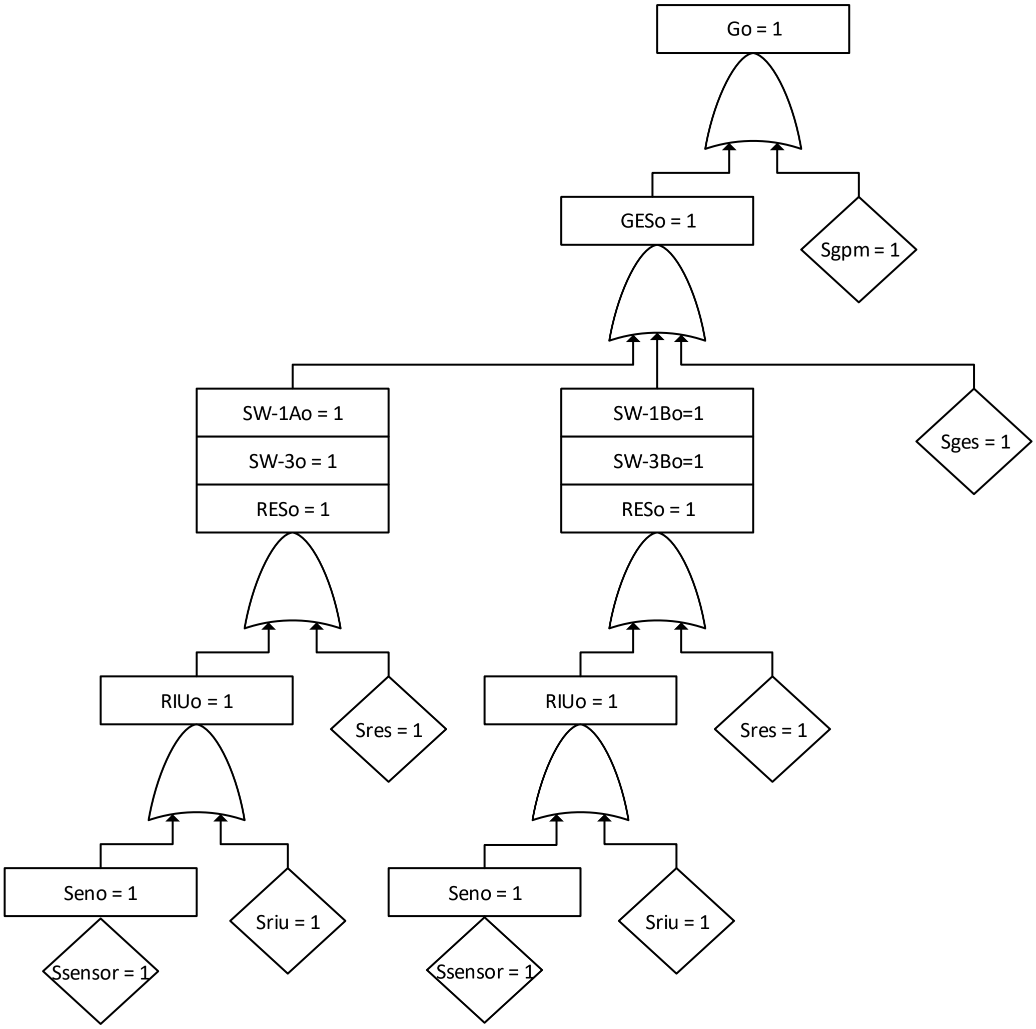
**RIU**

| Seno | Sriu | RIUo |
|---|---|---|
| 0 | 0 | 0 |
| -- | 1 | 1 |
| 1 | -- | 1 |

**Sensor**

| Ssensor | Seno |
|---|---|
| 0 | 0 |
| 1 | 1 |

Next we describe the systematic procedure for generating a fault tree using the component decision tables. Selecting the top-level event as GPM with an output that has an undetected erroneous data failure (Go=1), we start with the GPM decision table and find when the output is 1. There are two rows with Go = 1, therefore we use an OR gate with two inputs. The first input is from the GPM internal failure, Sgpm, which can be considered a basic-event, i.e., an event internal to the system and requires no further development. The second input is from the GPMES output, therefore we reference the GPMES decision table for when its output is 1. We continue this procedure until we reach the end of the components. The resulting fault tree is illustrated below.



Notice that in this fault tree there are repeated events. As discussed earlier, repeated events could be removed via Boolean logic, the algorithm which we implemented in our SOTERIA tool. Applying the reduction would result in the fault tree with the correct cut set.

Notice also that in this example to demonstrate fault tree synthesis from decision tables we left out the network integrity protection for ease of illustration. The following is a more comprehensive example where we illustrate how CRC integrity protection is added into the model using decision tables. The same IMA architecture is illustrated in Figure 9 along with its expected fault tree for undetected erroneous data. The expected fault tree is one that would be created by a safety engineer and that would have the expected minimal cut set.

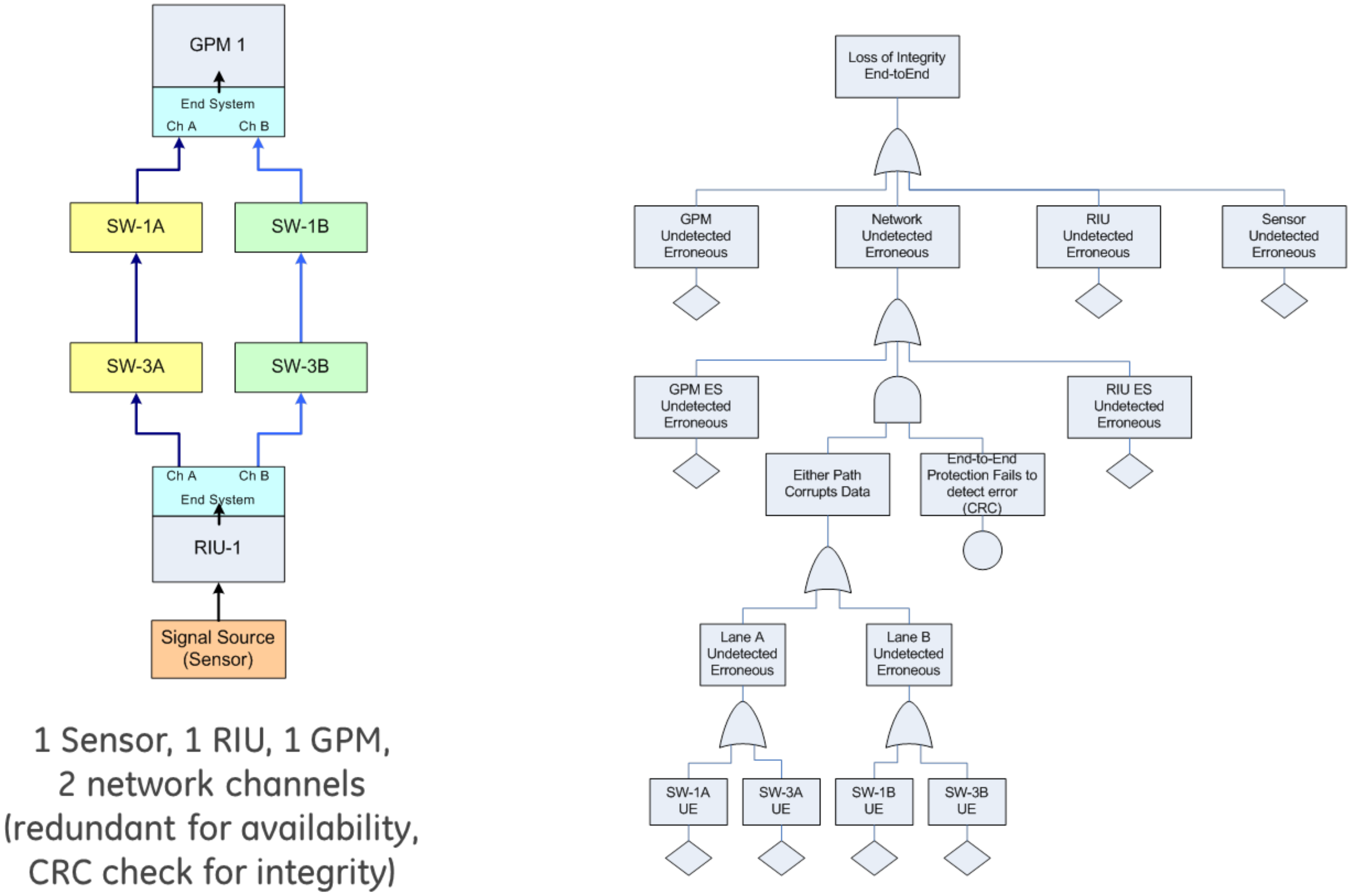

GPM 1
End System
Ch A
Ch B
SW-1A
SW-1B
SW-3A
SW-3B
Ch A
Ch B
End System
RIU-1
Signal Source (Sensor)
1 Sensor, 1 RIU, 1 GPM, 2 network channels (redundant for availability, CRC check for integrity)
Loss of Integrity End-toEnd
GPM Undetected Erroneous
Network Undetected Erroneous
RIU Undetected Erroneous
Sensor Undetected Erroneous
GPM ES Undetected Erroneous
RIU ES Undetected Erroneous
Either Path Corrupts Data
End-to-End Protection Fails to detect error (CRC)
Lane A Undetected Erroneous
Lane B Undetected Erroneous
SW-1A UE
SW-3A UE
SW-1B UE
SW-3B UE

Figure 9 - IMA architecture with its expected fault tree for undetected erroneous data.

This architecture includes a CRC check for integrity that is physically applied by the transmitting end system (RIU ES) and checked in the receiving end system (GPM ES). In this architecture, the CRC is only on the message traffic between the end systems. It does not protect against failures outside of equipment between the two communicating end systems. There could be, for example, an Application layer CRC, where the sensor encodes data with a CRC and the Application on the GPM checks the CRC. In that case, the fault tree would have that CRC protection mechanism modeled higher in the tree.

What we discovered in terms of modeling is that in order to correctly represent the CRC integrity wrapper we needed to include the feature in the model of the protected components (i.e., the switches) and not in the components where the feature is applied. Doing it in this way resulted in the fault tree we expected for undetected erroneous data. Below are the component models that make up this architecture. The column highlighted in yellow shows where CRC integrity wrapper is modeled, where 1 represents when a CRC protection fails to detect an error. Note that in this architecture we show CRC protection to all the switches. There could be, for example, a mixed network, where there are different fault protection approaches on different sub-networks. We could still accommodate for that in the models (or any other component protected by the CRC).

**GPM**

| In = GESo | internal failure = Sgpm | Output = Go |
|---|---|---|
| 0 | 0 | 0 |
| -- | 1 | 1 |
| 1 | -- | 1 |

**GPM ES**

| In1 = SW-1Ao | In2 = SW-1Bo | Sges | GESo |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | -- | -- | 1 |
| -- | 1 | -- | 1 |
| -- | -- | 1 | 1 |

**SW1A**

| In = SW-3Ao | Ssw1a | EScrc | SW-1Ao |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | -- | -- | 1 |
| -- | 1 | 1 | 1 |
| -- | 1 | 0 | 0 |

**SW3A**

| In = RESoa | Ssw3a | EScrc | SW-3Ao |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | -- | -- | 1 |
| -- | 1 | 1 | 1 |
| -- | 1 | 0 | 0 |

**SW1B**

| In = SW-3Bo | Ssw1b | EScrc | SW-1Bo |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | -- | -- | 1 |
| -- | 1 | 1 | 1 |
| -- | 1 | 0 | 0 |

**SW3B**

| In = RESob | Ssw3b | EScrc | SW-3Bo |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | -- | -- | 1 |
| -- | 1 | 1 | 1 |
| -- | 1 | 0 | 0 |

**RIU ES**

| In = RIUo | Sres | RESoa | RESob |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| -- | 1 | 1 | 1 |
| 1 | -- | 1 | 1 |

**RIU**

| In = Seno | Sriu | RIUo |
|---|---|---|
| 0 | 0 | 0 |
| -- | 1 | 1 |
| 1 | -- | 1 |

**Sensor**

| Ssensor | Seno |
|---|---|
| 0 | 0 |
| 1 | 1 |

Another architectural feature is sensor voting. An example of a multi-sensor IMA architecture is illustrated in Figure 10.

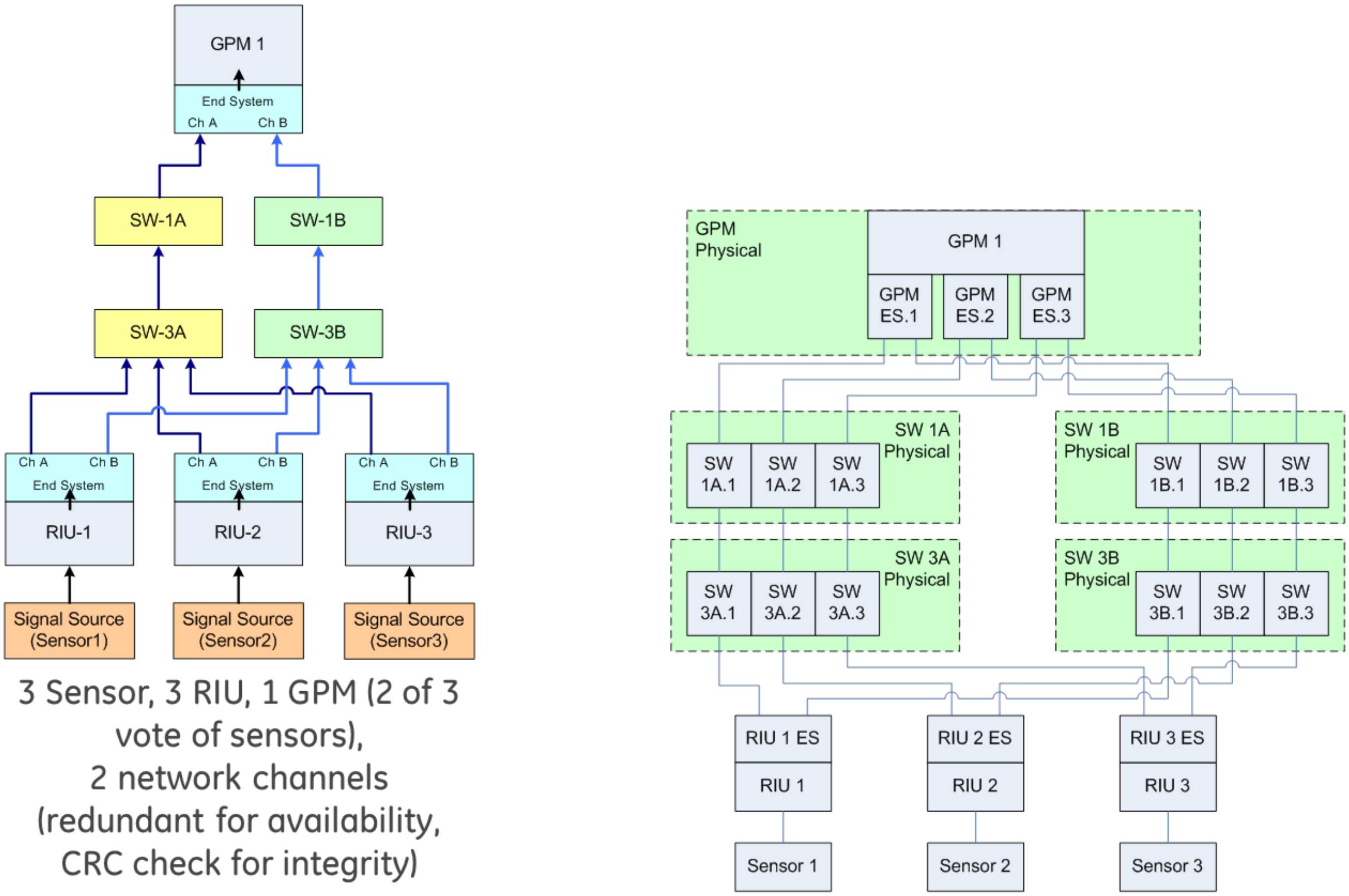


Figure 10 - IMA architecture with sensor voting.

Note that in this architecture there are two network switch paths, A and B. The modeling construct allows as many network channels as needed. The key here would be to make sure the component models for the equipment receiving the data off of the network are modeled to handle the data (i.e., support the correct number of input paths). We validated the modeling constructs using known architectures, but the methods are still applicable for other types of physical architectures.

In this architecture there are 3 sensors, 3 RIUs, and 1 GPM, where a 2-of-3 vote of sensors is applied. The voting feature is represented by extending a GPM component model to include more inputs and by altering the output. The GPM component model with 2-of-3 voting feature is illustrated in the decision table below.

**GPM (2-of-3 vote)**

| In1 = GESo1 | In2 = GESo2 | In3 = GESo3 | internal failure = Sgpm | Output = Go |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| -- | -- | -- | 1 | 1 |
| 1 | 1 | -- | -- | 1 |
| -- | 1 | 1 | -- | 1 |
| 1 | -- | 1 | -- | 1 |

In the process of extending our model to handle more features and more complex architectures what we discovered is that we needed to instantiate virtual models of the system based on data flows. For example, even though physically we have only 1 receiving end system (i.e., GPM ES), we need to instantiate the GPM ES component model 3 times in order to represent the dataflow. This is illustrated in Figure 10. The physical encapsulation is illustrated by the green dotted box while the virtual component models are in gray boxes.

## 5.2 Monotone Formulas

An interesting insight we discovered is that our component models are decision tables with limited expressiveness. For the cases when we have a faulted output (i.e., when there is a 1 in the output column), the inputs are either faulted (1) or don't care (--). This is a good thing because from a safety perspective things are modeled and analyzed from the view of when events and faults occur, not when they don't occur. Furthermore, FTA typically does not allow the use of NOT gates[2]. Not having to deal with negations increases analyzability and makes defining tables easier. What we further discovered is that we can support monotone formulas, a generalization of such decision tables. A Boolean formula is called monotone if it uses AND and OR gates and does not contain any negation gates. This is a more

[2] This is addressed in the technical report, "Automated Fault Tree Construction," by S. Contini, G. Squellati, section 3.2 Component Modeling: Method of Decision Tables, page 69, "Although certain codes, such as BAM, allow the use of NOT gates, the general fault tree analysis techniques do not."

convenient way for the end-user to model a component and express its behavior – he simple writes the formula expressing its faulted behavior in terms of ANDs and ORs, rather than listing all the combinations of inputs, internal states, and output behaviors. The fault tree synthesis algorithms complexity remains unchanged.

Below is an example of a monotone formula generalization of a decision table, described using a Switch that is protected with a CRC integrity wrapper. Undetected erroneous data propagates if input 1 propagates a fault *OR* an internal failure occur *AND* the end-to-end protection fail to detect an error (CRC fail). The decision table can be described using the following monotone formula:

**Switch w/CRC**

| In | Internal failure | EScrc | Out |
|---|---|---|---|
| 1 | -- | -- | 1 |
| -- | 1 | 1 | 1 |

$$\Rightarrow \quad Out = In \lor [Internal\ failure\ \land EScrc]$$

Consider another example which highlights the benefits of using monotone formulas. In this case, the monotone formula is a simpler expression. This behavior is not expressible directly using a decision table.

| In1 | In2 | In3 | In4 | Out |
|---|---|---|---|---|
| 1 | -- | -- | -- | 1 |
| -- | 1 | -- | 1 | 1 |
| -- | -- | 1 | 1 | 1 |

$$\Rightarrow \quad Out = In1 \lor (In2 \land In4) \lor (In3 \land In4)$$

$$Out = In1 \lor [In4 \land (In2 \lor In3)]$$

Given a monotone formula, we can automatically generate a fault tree using the procedure described earlier in this section. Note that the examples in this report have been with respect to undetected erroneous data, but the method can be used to model various types of fault propagation such as loss of function.

## 6 Modeling Language

We developed a modeling language for IMA architectures. The modeling language defines a library of components and describes the architecture of the system, which includes the components, their connections, and the type of flow that the user is interested in analyzing (for example, undetected erroneous data (UED) or loss of function/availability (LOA)). In coming up with the modeling constructs for the library and for the model, we took care to separate as much as possible the responsibility of the end-user that defines the library components and the end-user that constructs the system. We thereby decomposed the modeling construct into 2 local problems: 1) a library of components, and 2) connection definition. We also made the modeling language such that the users do not have to be an OCaml programmer in order to use it. In addition to the modeling constructs, we also came up with a list of validations needed to ensure that all connections described have been made and if that they are physically valid. Finally, we implemented the fault tree synthesis algorithms to automatically generate fault trees from the system descriptions. This section includes some material in our paper (Manolios, Siu, Noorman, & Liao, 2017) submitted for publication.

## 6.1 Library of Components

A library is a list of components. A component consists of a name, a list (possibly empty) of faults, a list (possibly empty) of input flow names, a list (possibly empty) of basic events, a list of basic event information consisting of pairs of floats corresponding to the failure rate (Lambda) and exposure time (Tau) of the basic events (so this list has the same length as the list of basic events), a list of output flow names, and a list of monotone Boolean formulas over the input flow names and the basic events. An example of a component with no input flows is a sensor that can only fail due to an internal fault. All components must have an output flow, otherwise there is nothing to analyze.

A library of components is a list of components, such that no two components have the same name.

Below is how the end-user would describe the library sufficient for modeling an IMA architecture such as that in Figure 2.

```
let library =
[  {name         = "Sensor";
    faults       = ["ued"; "loa"];
    input_flows  = [];
    basic_events = ["sen_flt_ued";"sen_flt_loa"];
    event_info   = [(1.0e-6, 1.0); (1.0e-5, 1.0)];
    output_flows = ["out"];
    formulas     = [(["out";"ued"], F["sen_flt_ued"]);
                    (["out";"loa"], F["sen_flt_loa"])]};

   {name         = "RIU";
    faults       = ["ued"; "loa"];
    input_flows  = ["in"];
    basic_events = ["riu_flt_ued";"riu_flt_loa"];
    event_info   = [(2.0e-7, 1.0); (5.0e-6, 1.0)];
    output_flows = ["out"];
    formulas     = [(["out";"ued"], Or[F["in";"ued"]; F["riu_flt_ued"]]);
                    (["out";"loa"], Or[F["in";"loa"]; F["riu_flt_loa"]])]};

   {name         = "Switch";
    faults       = ["ued"; "loa"];
    input_flows  = ["in1";"in2";"in3"];
    basic_events = ["sw_flt_ued"; "sw_flt_loa"; "crc32_flt"];
    event_info   = [(1.0e-6, 1.0); (1.0e-6, 1.0); (2.**(-32.),1.0)];
    output_flows = ["out1";"out2";"out3"];
    formulas     = [(["out1";"ued"],
                       Or[F["in1";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                    (["out2";"ued"],
                       Or[F["in2";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                    (["out3";"ued"],
                       Or[F["in3";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                    (["out1";"loa"], Or[F["in1";"loa"];F["sw_flt_loa"]]);
                    (["out2";"loa"], Or[F["in2";"loa"];F["sw_flt_loa"]]);
                    (["out3";"loa"], Or[F["in3";"loa"];F["sw_flt_loa"]])]};

    {name         = "GPM";
     faults       = ["ued"; "loa"];
     input_flows  = ["in1a1";"in1a2";"in1a3";"in1b1";"in1b2";"in1b3";];
     basic_events = ["gpm_flt_ued"; "gpm_flt_loa"];
     event_info   = [(2.0e-10, 1.0); (3.0e-5, 1.0)];
     output_flows = ["out"];
     formulas     = [(["out"; "ued"],
                        Or[F["gpm_flt_ued"];
                           N_of(2, [Or[F["in1a1";"ued"];F["in1b1";"ued"]];
                                    Or[F["in1a2";"ued"];F["in1b2";"ued"]];
                                    Or[F["in1a3";"ued"];F["in1b3";"ued"]]])]);
                     (["out"; "loa"],
                        Or[F["gpm_flt_loa"];
                           N_of(2, [And[F["in1a1";"loa"];F["in1b1";"loa"]];
                                    And[F["in1a2";"loa"];F["in1b2";"loa"]];
                                    And[F["in1a3";"loa"];F["in1b3";"loa"]]])])]};
];;
```

## 6.2 Model

A model consists of a list of instances, a list of connections, and the out flow of interest. An instance consists of the instance name, the name of the component it is instantiating from the library, and the exposure time. The connection information is a list of pairs consisting of an input flow and an output flow. Both flows have to exist in the list of instances of the model. The pair ((x, i), (y, o)) indicates that instance x's input flow i is fed by instance y's output flow o. Input flows have exactly 1 output flow connected to them. One output flow can feed multiple input flows.

Instances can optionally override the failure rate Lambda and/or the exposure times Tau originally defined in the library. Tau has to be editable for each basic event since this can change based on how often the component is checked, e.g. PBIT, CBIT, IBIT, or in some cases, physical inspection or test (more likely for analog and mechanical components). The failure rate, Lambda, for the most part remains constant, but can also change. Lambda is sourced from either the reliability prediction or FMEA. These analyses take into account several factors that can impact the basic failure rate of a component, e.g. installed environment, quality, etc. The intent is to build the component library for each project's architecture based on the failure rates determined by the reliability process and for the most part the rates should remain static for that project. There are times, however, when Lambda would change. For instance, obsolescence often forces a component change (same function, but physically different) and revision to the project reliability data and subsequent update to the quantitative safety analyses. A change in supplier or a change in device (e.g. higher performance) may also force a change in a component failure rate. Therefore, we provide the user the capability to change these values when instancing the model.

Below is how the end-user would describe a model. The instance of Sensor demonstrates the option to override the exposure time Tau originally defined in the library component.

```
(* ----- UED MODEL ----- *)
let figure2_ued =
  { instances =
      [makeInstance ~i:"sensor" ~c:"Sensor" ~t:[("sen_flt_ued", 5.0)] ();
       makeInstance "riu1" "RIU" ();
       makeInstance "riu2" "RIU" ();
       makeInstance "riu3" "RIU" ();
       makeInstance "sw1a" "Switch" ();
       makeInstance "sw1b" "Switch" ();
       makeInstance "sw3a" "Switch" ();
       makeInstance "sw3b" "Switch" ();
       makeInstance "gpm1" "GPM" ();
      ];
    connections =
      [ (("riu1",  "in"), ("sensor", "out"));
        (("riu2",  "in"), ("sensor", "out"));
        (("riu3",  "in"), ("sensor", "out"));
        (("sw3a", "in1"), ("riu1", "out"));
        (("sw3a", "in2"), ("riu2", "out"));
        (("sw3a", "in3"), ("riu3", "out"));
        (("sw3b", "in1"), ("riu1", "out"));
        (("sw3b", "in2"), ("riu2", "out"));
        (("sw3b", "in3"), ("riu3", "out"));
        (("sw1a", "in1"), ("sw3a", "out1"));
        (("sw1a", "in2"), ("sw3a", "out2"));
        (("sw1a", "in3"), ("sw3a", "out3"));
        (("sw1b", "in1"), ("sw3b", "out1"));
        (("sw1b", "in2"), ("sw3b", "out2"));
        (("sw1b", "in3"), ("sw3b", "out3"));
        (("gpm1", "in1a1"), ("sw1a", "out1"));
        (("gpm1", "in1a2"), ("sw1a", "out2"));
        (("gpm1", "in1a3"), ("sw1a", "out3"));
        (("gpm1", "in1b1"), ("sw1b", "out1"));
        (("gpm1", "in1b2"), ("sw1b", "out2"));
        (("gpm1", "in1b3"), ("sw1b", "out3"));
      ];
    top_fault = ("gpm1", F["out";"ued"])
  } ;;

(* ----- LOA MODEL ----- *)
let figure2_loa =
  { instances = figure2_ued.instances;
    connections = figure2_ued.connections;
    top_fault = ("gpm1", F["out";"loa"])
  };;
```

## 6.3 Validation Checks

We came up with a list of validation checks to assist the end-user during component library and model construction. Some checks pertain to just the component library, some to just the system model, and other to the model with respect to the component library. The checks are ordered so that the flow makes procedural sense to the end-user, as shown in Table 1.

Table 1. Validation checks presented in order

| Order | Validation Check | Description |
|---|---|---|
| 1 | `checkLibrary_componentUnique` | No two components have the same name. |
| 2 | `checkLibrary_nonEmptyFaults` | For any component, faults is not empty. |
| 3 | `checkLibrary_disjointInputFlowsandBasicEvents` | For any component, {input_flows} ∩ {basic_events}=empty set. |
| 5 | `checkLibrary_allOutputFaultsHaveFormulas` | For any component, all outputs faults have a formula defined. |
| 6 | `checkLibrary_formulasMakeSense` | For any formula (a, f), a is in faults, f uses references variables that are well defined within the component. |

| Order | Validation Check | Description |
|---|---|---|
| 8 | `checkModel_instanceNameUnique` | All instance names are disjoint. |
| 9 | `checkModel_cnameInstanceIsDefinedInLibrary` | When instancing components in a model, all component names are defined in the library. |
| 10 | `checkModel_exposureOfBasicIsDefinedInLibrary` | For any exposure (a, x), a is a well defined basic event in that component of the library. |
| 11 | `checkModel_validConnections` | All connections are valid, including input flows and instance. |
| 12 | `checkModel_inputFlowUnique` | In the connections of the model, each input flow has exactly 1 instance connected to it. |

The following sections contain examples to demonstrate the checks. Each example contains an error injected in the component library (and/or system model), as well as the corresponding output of the validation check. The validation checks are organized into three categories: 1) checks on component library only, 2) checks on system model only, and 3) checks on model with respect to the component library.

### 6.3.1 Checks on the Component Library

All error injections described within this sub-section are in the component library. Errors are highlighted in yellow.

#### *6.3.1.1 No two components have the same name*

A library of components is a list of components, such that no two components have the same name.
Error Description: two components have the same name, “Sensor.”

```
let library =
  [ {name         = "Sensor";
     faults       = ["ued"; "loa"];
     input_flows  = [];
     basic_events = ["sen_flt_ued";"sen_flt_loa"];
     event_info   = [(1.0e-6, 1.0); (1.0e-5, 1.0)];
     output_flows = ["out"];
     formulas     = [(["out";"ued"], F["sen_flt_ued"]);
                     (["out";"loa"], F["sen_flt_loa"])]};

    {name         = "Sensor";
     faults       = ["ued"; "loa"];
     input_flows  = ["rin"];
     basic_events = ["sen_flt_ued";"sen_flt_loa"];
     event_info   = [(2.0e-7, 1.0); (5.0e-6, 1.0)];
     output_flows = ["out"];
     formulas     = [(["out";"ued"], Or[F["rin";"ued"]; F["sen_flt_ued"]]);
                     (["out";"loa"], Or[F["rin";"loa"]; F["sen_flt_loa"]])]};
     …
   ];;
```

The check detects the error and gives the output below.

```
# checkLibrary_componentUnique library;;
Duplicate component: Sensor
- : (string, string) Core.Std._result =
Core.Std.Error "Library component not unique - see above"
```

### 6.3.1.2 All components must support at least one fault

For any component in the component library, the “faults” section must not be empty.

Error Description:  the RIU component in the library does not support any faults.

```
let library =
  [ …
    {name          = "RIU";
     faults        = [];
     input_flows   = ["riu_flt_ued";"riu_flt_loa"];
     basic_events  = ["riu_flt_ued";"riu_flt_loa"];
     event_info    = [(1.0e-6, 1.0);(1.0e-5, 1.0)];
     output_flows  = ["out"];
     formulas      = [(["out";"ued"], Or[F["rin";"ued"]; F["riu_flt_ued"]]);
                       (["out";"loa"], Or[F["rin";"loa"]; F["riu_flt_loa"]])]};
     …
   ];;
```

The check detects the error and gives the output below.

```
# checkLibrary_nonEmptyFaults library;;
- : (string, string) Core.Std._result =
Core.Std.Error "Faults not defined for library component RIU"
```

### 6.3.1.3 Input_flows and basic_events names are disjoint

For each component in the component library, check that {input_flows} ∩ {basic_events} = empty set.

Error Description: "riu_flt_ued" and "riu_flt_loa" are used in both input_flows and basic_events.

```
let library =
  [
   …
    {name          = "RIU";
     faults        = [];
     input_flows   = ["riu_flt_ued";"riu_flt_loa"];
     basic_events  = ["riu_flt_ued";"riu_flt_loa"];
     event_info    = [(1.0e-6, 1.0);(1.0e-5, 1.0)];
     output_flows  = ["out"];
     formulas      = [(["out";"ued"], Or[F["rin";"ued"]; F["riu_flt_ued"]]);
                       (["out";"loa"], Or[F["rin";"loa"]; F["riu_flt_loa"]])]};

   …
  ];;
```

The check detects the error and gives the output below.

```
# checkLibrary_disjointInputFlowsandBasicEvents library;;
- : (string, string) Core.Std._result =
Core.Std.Error
"Names used for input_flows and basic_events are not disjoint in component RIU"
```

#### 6.3.1.4 Consistency between length of basic_events and event_info; Consistency between fault and formulas

One way to check consistency is to check the following list lengths in the component library.

- Length (basic_events list) = Length (event_info list)
- Length (faults list) * Length (output_flows list) = Length (formulas list)

Error Description: the fault “ued_or_loa” does not have an associated formula defined.

```
let library =
  [
    {name         = "Sensor";
     faults       = ["ued"; "loa"; "ued_or_loa"];
     input_flows  = [];
     basic_events = ["sen_flt_ued";"sen_flt_loa"];
     event_info   = [(1.0e-6, 1.0); (1.0e-5, 1.0);];
     output_flows = ["out"];
     formulas     = [(["out";"ued"], F["sen_flt_ued"]);
                      (["out";"loa"], F["sen_flt_loa"])]};


     …
  ];;
```

The check detects the error and gives the output below.

```
# checkLibrary_listsAreConsistentLengths library;;
- : (string, string) Core.Std._result =
Core.Std.Error
"Faults and formulas lists are of inconsistent lengths in component Sensor"
```

Here, a different error is injected. Error Description: the length of the list basic_events is 2, whereas the length of the list event_info is 1.

```
let library =
  [
    {name         = "Sensor";
     faults       = ["ued"; "loa"];
     input_flows  = [];
     basic_events = ["sen_flt"; "sen_flt2"];
     event_info   = [(1.0e-6, 1.0)];
     output_flows = ["out"];
     formulas     = [(["out";"ued"], F["sen_flt"]);
                      (["out";"loa_1"], F["out";"ued"])]};
     …
   ];;
```

The check detects the error and gives the output below.

```
# checkLibrary_listsAreConsistentLengths library;;
- : (string, string) Core.Std._result =
Core.Std.Error
"Basic events and event info are of inconsistent lengths in component Sensor".
```

#### 6.3.1.5 All faults have a formula defined

Check that all faults in a component have a formula associated with it.

```
let library =
  [
    {name         = "Sensor";
     faults       = ["ued"; "loa"; "ued_or_loa"];
     input_flows  = [];
     basic_events = ["sen_flt_ued";"sen_flt_loa"];
     event_info   = [(1.0e-6, 1.0); (1.0e-5, 1.0); (1.0e-5, 1.0);];
     output_flows = ["out"];
     formulas     = [(["out";"ued"], F["sen_flt_ued"]);
                      (["out";"loa"], F["sen_flt_loa"])]};
     …
  ];;
```

Error Description: the flow “ued_or_loa” does not have an associated formula defined.

The check detects the error and gives the output below.

```
# checkLibrary_allOutputFaultsHaveFormulas library;;
- : (string, string) Core.Std._result =
Core.Std.Error "Not all output faults have formulas, check component Sensor"
```

#### 6.3.1.6 Invalid component library formula

If the formula is of the form (a, f), then the variables used in f are a subset of the variables in the union of input_flows, basic_events, and any formula variables defined.

Error Description: the variable “loa_1” is not defined in this component.

```
let library =
  [
   …
    {name         = "Sensor";
     faults       = ["ued"; "loa"];
     input_flows  = [];
     basic_events = ["sen_flt"; "sen_flt2"];
     event_info   = [(1.0e-6, 1.0)];
     output_flows = ["out"];
     formulas     = [(["out";"ued"], F["sen_flt"]);
                      (["out";"loa_1"], F["out";"ued"])]};
   …
  ];;
```

The check detects the error and gives the output below.

```
# checkLibrary_formulasMakeSense library;;
- : (string, string) Core.Std._result =
Core.Std.Error
 "Invalid formula in component Sensor, check formula [out,loa_1]"
```

### 6.3.2 Checks on the System Model

All error injections described within this sub-section are in the system model. Errors are highlighted in yellow.

### *6.3.2.1 Model instances have unique instance names.*

This check applies to the “instances” section of a model.

Error Description: the instance name “swa” is used twice in the model.

```
let gar_001 =
  { instances =
      [makeInstance ~i:"sensor" ~c:"Sensor" ~t:[("sen_flt_ued", 5.0)] ();
       makeInstance "riu" "RIU" ();
       makeInstance "swa" "Switch" ();
       makeInstance "swa" "Switch" ();
       makeInstance "gpm" "GPM" ();
      ];
    connections =
      [ (("riu",  "rin"),  ("sensor", "out"));
        (("swa", "swin"), ("riu", "out"));
        (("swb", "swin"), ("riu", "out"));
        (("gpm",  "gin1"), ("swa", "out"));
        (("gpm",  "gin2"), ("swb", "out"));
      ];
    top_fault = ("gpm", F["out";"ued"])
  } ;;
```

The check detects the error and gives the output below.

```
# checkModel_instanceNameUnique gar_001;;
- : (string, string) Core.Std._result =
Core.Std.Error "Model instance names are not unique"
```

### *6.3.2.2 Input flows have exactly one instance connected to them.*

The “input_flows” in the “connections” is always defined as an instance-inflow pair, such as ("gpm", "gin1"). Therefore, we need to check that ("gpm", "gin1") will not appear in the connection section more than once.

Error Description: the input flow (“gpm”, “gin1”) have two instances connected to it.

```
let gar_001 =
  {
   …
  connections =
  [ (("riu",   "rin"),  ("sensor", "out"));
    (("sw3a", "swin"), ("riu", "out"));
    (("sw3b", "swin"), ("riu", "out"));
    (("gpm",   "gin1"), ("swa", "out"));
    (("gpm",   "gin1"), ("swb", "out"));
  ];
    …
  } ;;
```

The check is able to detect the error, as shown below.

```
# checkModel_inputFlowUnique gar_bad;;
- : (string, string) Core.Std._result =
 Core.Std.Error
 "One of the input_flows in the model has more than one connection made to it."
```

### 6.3.3 Checks on the System Model with respect to the Component Library

All error injections described within this sub-section are in the component library and system model. Errors are highlighted in yellow.

#### *6.3.3.1 All names in a model correspond to actual components in the component library.*

When instancing components for the model, check whether the component is defined in the library.
Error Description: in the model, the component “DME” is not defined in the component library.

```
let gar_001 =
{ instances =
     [makeInstance ~i:"sensor" ~c:"DME" ~t:[("sen_flt_ued", 5.0)] ();
      makeInstance "riu" "RIU" ();
      makeInstance "swa" "Switch" ();
      makeInstance "swa" "Switch" ();
      makeInstance "gpm" "GPM" ();
      ];
     …
  };;
```

The check detects the error and gives the output below.

```
# checkModel_cnameInstanceIsDefinedInLibrary gar_001 library;;
- : (string, string) Core.Std._result =
Core.Std.Error
 "Invalid Component: this instantiation references a component that is not in the
library: DME"
```

#### *6.3.3.2 Elements of the form (a, x) in exposures of an instance make sense, i.e., there is a basic event named a in the component being instantiated.*

Error Description: in the model, “flt_ued” associated with the instance “sensor” is not a basic event of the component “Sensor” in the component library.

```
let library =
  [ …
    {name         = "Sensor";
     faults       = ["ued"; "loa"];
     input_flows  = ["sin"];
     basic_events = ["sen_flt_ued";"sen_flt_loa"];
     event_info   = [(1.0e-6, 1.0);(1.0e-5, 1.0)];
     output_flows = ["out"];
     formulas     = [FM(("out", "ued"), F("sen_flt_ued"));
                     FM(("out", "loa"),  F("sen_flt_loa"))]};
    …
];;

let gar_001 =
{ instances =
      [makeInstance ~i:"sensor" ~c:"Sensor" ~t:[("flt_ued", 5.0)] ();
       makeInstance "riu" "RIU" ();
       makeInstance "swa" "Switch" ();
       makeInstance "swa" "Switch" ();
```

```
        makeInstance "gpm" "GPM" ();
        ];
    …
} ;;
```

The check detects the error and gives the output below.

```
# checkModel_exposureOfBasicIsDefinedInLibrary gar_001 library;;
- : (string, string) Core.Std._result =
Core.Std.Error "Model attempts to change an invalid basic_event of a library
component"
```

### 6.3.3.3 *The connection information is a list of pairs consisting of an input flow and an instance.*

Error Description: in the model, the instance named “riu” is an instantiation of the “RIU” component; “riu” references “riu_in” in the connections as the input_flow, but is not defined in the RIU component of the library.

```
let library =
[    {name          = "Sensor";
      faults        = ["ued"; "loa"];
      input_flows   = ["rin"];
      basic_events = ["sen_flt_ued";"sen_flt_loa"];
      event_info    = [(1.0e-6, 1.0); (1.0e-5, 1.0)];
      output_flows = ["out"];
      formulas      = [(["out";"ued"], Or[F["rin";"ued"]; F["sen_flt_ued"]]);
                        (["out";"loa"], Or[F["rin";"loa"]; F["sen_flt_loa"]])]};

     {name          = "RIU";
      faults        = ["ued"; "loa"];
      input_flows   = ["rin"];
      basic_events = ["riu_flt_ued";"riu_flt_loa"];
      event_info    = [(2.0e-7, 1.0); (5.0e-6, 1.0)];
      output_flows = ["out"];
      formulas      = [(["out";"ued"], Or[F["rin";"ued"]; F["riu_flt_ued"]]);
                        (["out";"loa"], Or[F["rin";"loa"]; F["riu_flt_loa"]])]};

    {name          = "Switch";
     faults        = ["ued";];
     input_flows   = ["swin"];
     basic_events = ["sw_flt"];
     event_info    = [(1.0e-6, 1.0)];
     output_flows = ["out"];
     formulas      = [(["out";"ued"], Or[F["swin";"ued"];F["sw_flt"]])]};

     {name          = "GPM";
      faults        = ["ued"; "loa"];
      input_flows   = ["gin1";"gin2"];
      basic_events = ["gpm_flt_ued"; "gpm_flt_loa"];
      event_info    = [(2.0e-10, 1.0); (3.0e-5, 1.0)];
      output_flows = ["out"];
      formulas      = [(["out"; "ued"],
                          Or[F["gin1";"ued"];F["gin2";"ued"];F["gpm_flt_ued"]]);
                        (["out"; "loa"],
                          Or[And[F["gin1";"loa"];F["gin2";"loa"]; F["gpm_flt_loa"]]])]};

];;

let gar_001 =
{ instances =
      [makeInstance ~i:"sensor" ~c:"Sensor" ~t:[("sen_flt_ued", 5.0)] ();
       makeInstance "riu" "RIU" ();
       makeInstance "swa" "Switch" ();
       makeInstance "swb" "Switch" ();
       makeInstance "gpm" "GPM" ();
      ];
    connections =
      [(("riu", "rin_in"),("sensor", "out"));
       (("swa", "swin"),  ("riu", "out"));
       (("swb", "swin"),  ("riu", "out"));
       (("gpm", "gin1"),  ("swa", "out"));
       (("gpm", "gin1"),  ("swb", "out"));
      ];
     top_fault = ("gpm", F["out", "ued"])
} ;;
```

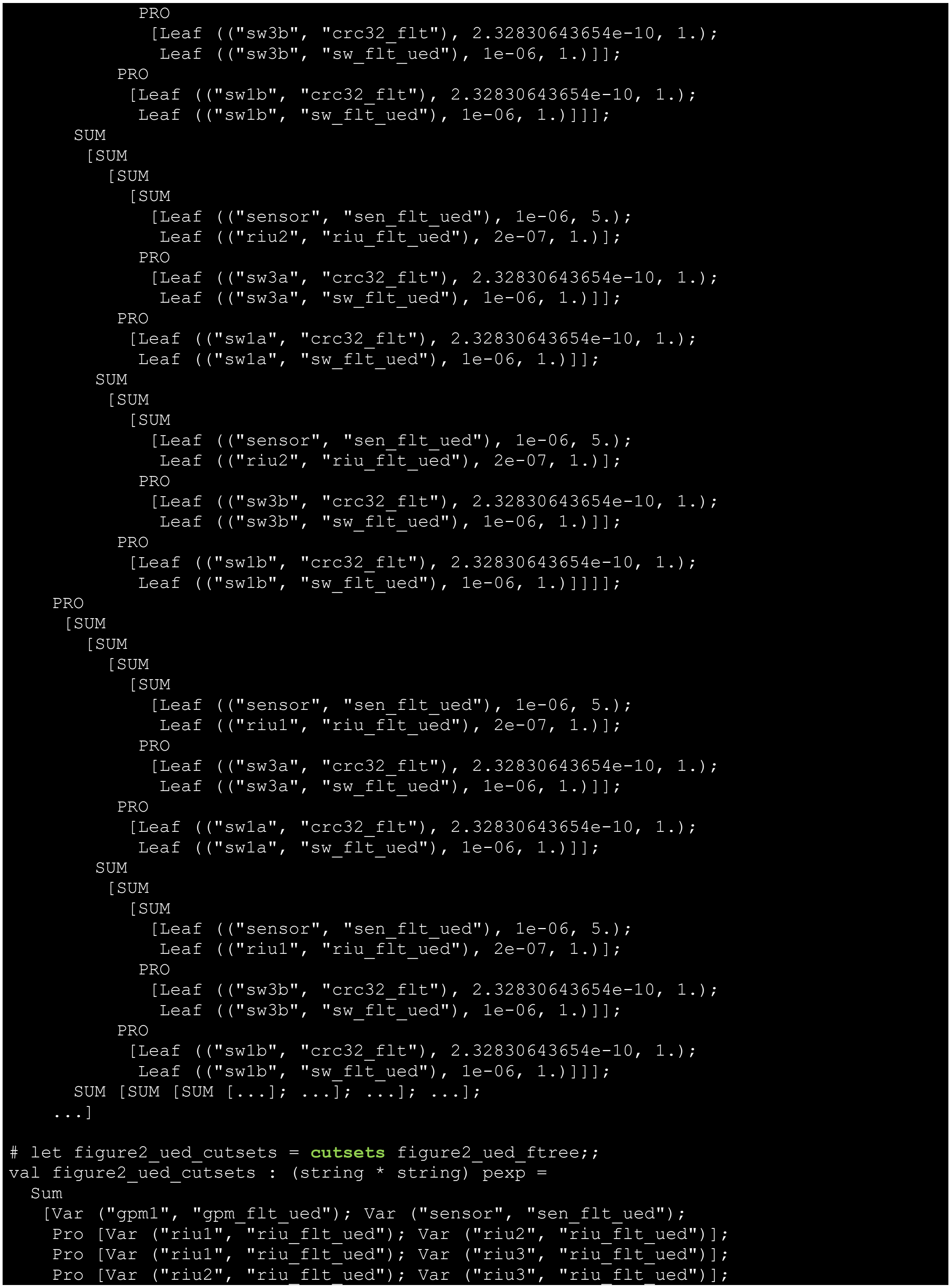
The check detects the error and gives the output below.

```
# checkModel_validConnections gar_bad2 library_good;;
- : (string, string) Core.Std._result =
Core.Std.Error
 "Invalid connection: this is not a valid component input from the library: (riu,
rin_in)"
```

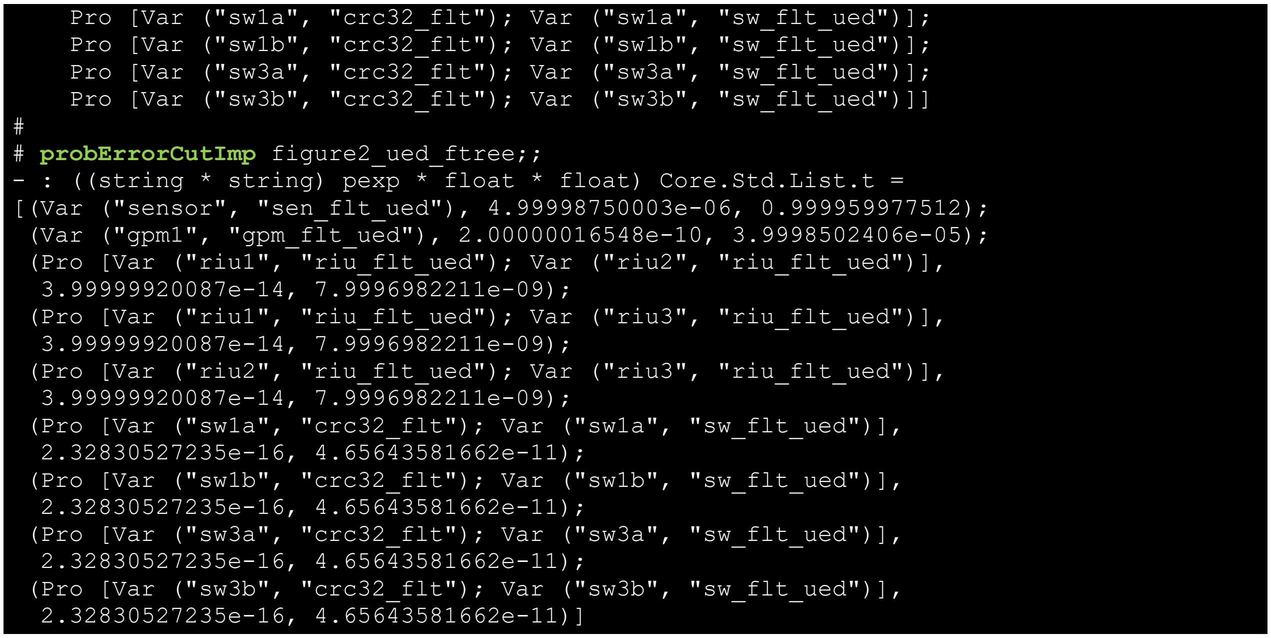
## 7 Analysis Capability

Our SOTERIA tool takes the modeling constructs described in section 6 and automatically generates the fault tree analysis. `model_to_ftree` generates the fault tree, cutsets lists the cutsets from the fault tree, and probErrorCutImp generates the cut set list along with its probability and ranked with an importance metric. Below is the output from calling these functions for the UED fault from section 6. The fault tree and cut sets agrees with the manually generated analysis in Figure 8. The raw output is difficult for a human to analyze, especially as the model gets large. To help the end-user, we added the capability to visualize the fault-tree and the model, which we describe in section 8.

```
# let figure2_ued_ftree = model_to_ftree library figure2_ued ;;
val figure2_ued_ftree : (string * string) ftree =
  SUM
   [Leaf (("gpm1", "gpm_flt_ued"), 2e-10, 1.);
    PRO
     [SUM
       [SUM
         [SUM
           [SUM
             [Leaf (("sensor", "sen_flt_ued"), 1e-06, 5.);
              Leaf (("riu1", "riu_flt_ued"), 2e-07, 1.)];
            PRO
             [Leaf (("sw3a", "crc32_flt"), 2.32830643654e-10, 1.);
              Leaf (("sw3a", "sw_flt_ued"), 1e-06, 1.)]];
          PRO
           [Leaf (("sw1a", "crc32_flt"), 2.32830643654e-10, 1.);
            Leaf (("sw1a", "sw_flt_ued"), 1e-06, 1.)]];
        SUM
         [SUM
           [SUM
             [Leaf (("sensor", "sen_flt_ued"), 1e-06, 5.);
              Leaf (("riu1", "riu_flt_ued"), 2e-07, 1.)];
```



```
            PRO
             [Leaf (("sw3b", "crc32_flt"), 2.32830643654e-10, 1.);
              Leaf (("sw3b", "sw_flt_ued"), 1e-06, 1.)]];
          PRO
           [Leaf (("sw1b", "crc32_flt"), 2.32830643654e-10, 1.);
            Leaf (("sw1b", "sw_flt_ued"), 1e-06, 1.)]]];
      SUM
       [SUM
         [SUM
           [SUM
             [Leaf (("sensor", "sen_flt_ued"), 1e-06, 5.);
              Leaf (("riu2", "riu_flt_ued"), 2e-07, 1.)];
            PRO
             [Leaf (("sw3a", "crc32_flt"), 2.32830643654e-10, 1.);
              Leaf (("sw3a", "sw_flt_ued"), 1e-06, 1.)]];
          PRO
           [Leaf (("sw1a", "crc32_flt"), 2.32830643654e-10, 1.);
            Leaf (("sw1a", "sw_flt_ued"), 1e-06, 1.)]];
        SUM
         [SUM
           [SUM
             [Leaf (("sensor", "sen_flt_ued"), 1e-06, 5.);
              Leaf (("riu2", "riu_flt_ued"), 2e-07, 1.)];
            PRO
             [Leaf (("sw3b", "crc32_flt"), 2.32830643654e-10, 1.);
              Leaf (("sw3b", "sw_flt_ued"), 1e-06, 1.)]];
          PRO
           [Leaf (("sw1b", "crc32_flt"), 2.32830643654e-10, 1.);
            Leaf (("sw1b", "sw_flt_ued"), 1e-06, 1.)]]]];
    PRO
     [SUM
       [SUM
         [SUM
           [SUM
             [Leaf (("sensor", "sen_flt_ued"), 1e-06, 5.);
              Leaf (("riu1", "riu_flt_ued"), 2e-07, 1.)];
            PRO
             [Leaf (("sw3a", "crc32_flt"), 2.32830643654e-10, 1.);
              Leaf (("sw3a", "sw_flt_ued"), 1e-06, 1.)]];
          PRO
           [Leaf (("sw1a", "crc32_flt"), 2.32830643654e-10, 1.);
            Leaf (("sw1a", "sw_flt_ued"), 1e-06, 1.)]];
        SUM
         [SUM
           [SUM
             [Leaf (("sensor", "sen_flt_ued"), 1e-06, 5.);
              Leaf (("riu1", "riu_flt_ued"), 2e-07, 1.)];
            PRO
             [Leaf (("sw3b", "crc32_flt"), 2.32830643654e-10, 1.);
              Leaf (("sw3b", "sw_flt_ued"), 1e-06, 1.)]];
          PRO
           [Leaf (("sw1b", "crc32_flt"), 2.32830643654e-10, 1.);
            Leaf (("sw1b", "sw_flt_ued"), 1e-06, 1.)]]]];
      SUM [SUM [SUM [...]; ...]; ...]; ...];
    ...]

# let figure2_ued_cutsets = cutsets figure2_ued_ftree;;
val figure2_ued_cutsets : (string * string) pexp =
  Sum
   [Var ("gpm1", "gpm_flt_ued"); Var ("sensor", "sen_flt_ued");
    Pro [Var ("riu1", "riu_flt_ued"); Var ("riu2", "riu_flt_ued")];
    Pro [Var ("riu1", "riu_flt_ued"); Var ("riu3", "riu_flt_ued")];
    Pro [Var ("riu2", "riu_flt_ued"); Var ("riu3", "riu_flt_ued")];
```



```
    Pro [Var ("sw1a", "crc32_flt"); Var ("sw1a", "sw_flt_ued")];
    Pro [Var ("sw1b", "crc32_flt"); Var ("sw1b", "sw_flt_ued")];
    Pro [Var ("sw3a", "crc32_flt"); Var ("sw3a", "sw_flt_ued")];
    Pro [Var ("sw3b", "crc32_flt"); Var ("sw3b", "sw_flt_ued")]]
#
# probErrorCutImp figure2_ued_ftree;;
- : ((string * string) pexp * float * float) Core.Std.List.t =
[(Var ("sensor", "sen_flt_ued"), 4.99998750003e-06, 0.999959977512);
 (Var ("gpm1", "gpm_flt_ued"), 2.00000016548e-10, 3.9998502406e-05);
 (Pro [Var ("riu1", "riu_flt_ued"); Var ("riu2", "riu_flt_ued")],
  3.99999920087e-14, 7.9996982211e-09);
 (Pro [Var ("riu1", "riu_flt_ued"); Var ("riu3", "riu_flt_ued")],
  3.99999920087e-14, 7.9996982211e-09);
 (Pro [Var ("riu2", "riu_flt_ued"); Var ("riu3", "riu_flt_ued")],
  3.99999920087e-14, 7.9996982211e-09);
 (Pro [Var ("sw1a", "crc32_flt"); Var ("sw1a", "sw_flt_ued")],
  2.32830527235e-16, 4.65643581662e-11);
 (Pro [Var ("sw1b", "crc32_flt"); Var ("sw1b", "sw_flt_ued")],
  2.32830527235e-16, 4.65643581662e-11);
 (Pro [Var ("sw3a", "crc32_flt"); Var ("sw3a", "sw_flt_ued")],
  2.32830527235e-16, 4.65643581662e-11);
 (Pro [Var ("sw3b", "crc32_flt"); Var ("sw3b", "sw_flt_ued")],
  2.32830527235e-16, 4.65643581662e-11)]
```

We further discuss an additional capability in our analysis which is the ability to handle loops in the model when generating fault trees. For this work, we referenced the NASA Fault Tree Handbook with Aerospace Applications, version 1.1, Section 5.4, Modeling Loops and Feedback (NASA, 2002). This section presented a space shuttle example where the orbiter sends a control signal to the main engine and the main engine provides a feedback signal back to the orbiter, as shown in Figure 11. A loop occurs because the failure of the orbiter depends on the failure of the main engine which depends on the failure of the orbiter (Figure 12).

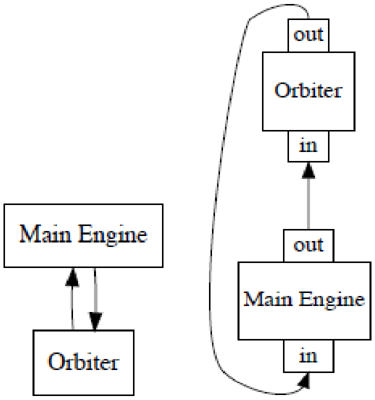


Figure 11. Visualization of the physical and functional architecture of the space shuttle example.

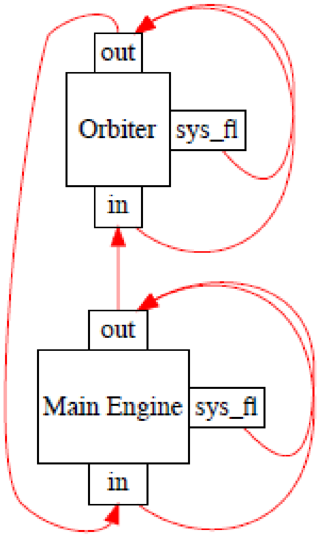
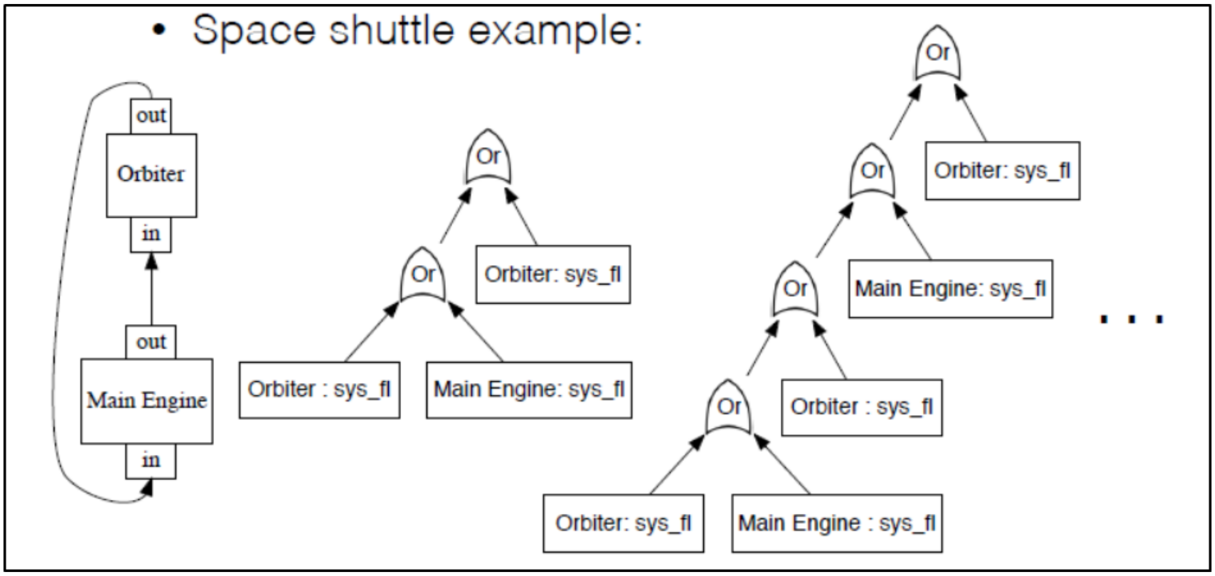


Figure 12. Visualization of the Orbiter & Main Engine feedback loop.



- Space shuttle example:

Figure 13. Prior to handling loops -- stack overflow.

To handle loops our algorithm first needs to detect them. Loops occur when we try to synthesize the fault tree. The synthesis proceeds by unwinding the fault definitions in each component and to the next connected component. Starting at the top-level fault, the algorithm unwinds the definitions until it reaches a fixpoint. A loop occurs when there is a cycle in the expanded fault definition. Note that we can legitimately expand out the definition of some faults more than once without there being a loop. Therefore, repetition must consider the context. We look for loops only along the path of ancestors of the fault tree being generated. This would depend on where you are in the tree.

Once a loop is discovered it needs to be broken. First, we break loops only in the context of conjunction and disjunction. Here is a simple example: when we come across a loop that has the formula A ∨ A ∨ A ∨ A ∨ …, we want to break the loop by replacing it with something. We note that A ∨ false = A, so we replace this type of loop (i.e., starting after the first disjunction) with false so that we can replace this loop with an A. Similarly, when we come across a loop that has A ∧ A ∧ A ∧ A ∧ …, we replace the loop with true so that we get A ∧ true = A and replace this loop with an A. From the systems that we have seen so far, we tend to see unwinding of the fault definitions to loops with these conjunctions and disjunctions, so it's likely that we will be able to break the loops in the context of conjunction and disjunction once the algorithm has expanded the formulas enough to reveal spots like these.

In addition, we break loops by replacing repeated faults with identity. For example, when we come across A ∧ B ∧ B ∧ B ∧ …, we replace the loop with I so we get A ∧ B ∧ I. Another way to think of this is if we discover a tree that we have seen before, we can replace it with an identity.

Finally, if there is a loop of the form Ain = Bout, Bout = Ain – in other words, the fault is just passed through the components with no additional basic events influencing the outputs of A or B – then we report an error. We could have replaced this sort of a loop with true and declare that a fault will always be passed through, but this will likely cause more confusion for the safety engineer when analyzing the resulting fault tree.

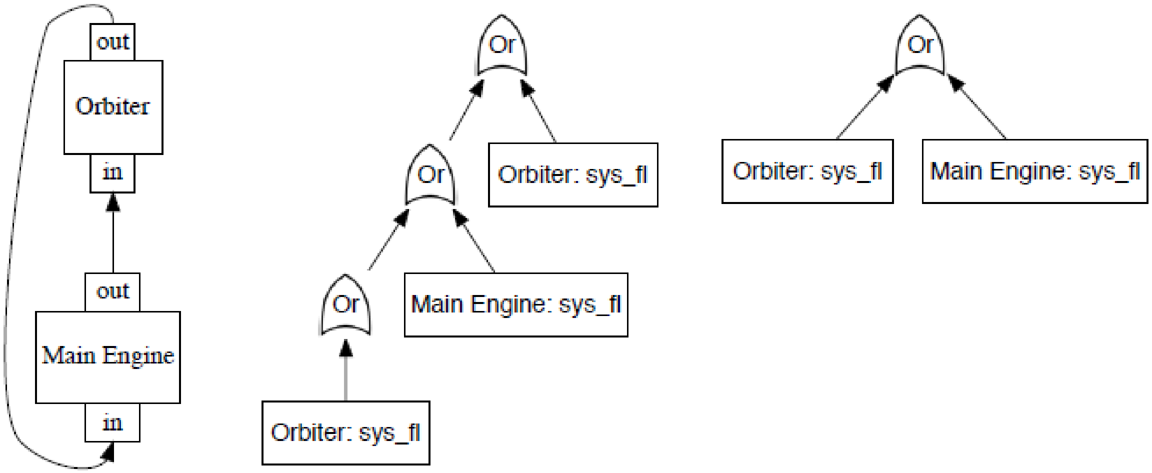


Figure 14. Fault tree synthesis with loops – space shuttle example fault tree and cut sets.

## 8 Visualization

We added a visualization capability to our model which automatically generates a view of the physical architecture, a view of the functional architecture, a view of the fault propagation through the architecture, and the fault tree. Raw output is difficult for a human to digest. We describe the visualization with the use of a very simple example – a system with 4 components, where data from a distance measuring equipment (DME) is sent through a switch to a GPM (general processing module) for processing, then back through the same switch to be displayed on a primary flight display (PFD). The library of components and model that describes this example are as follows:

```
let slibrary=
  [{name         = "GPM";
    faults       = ["ued";];
    input_flows  = ["gin"];
    basic_events = ["gpm_fl"];
    event_info   = [(2.0e-10, 1.0)];
    output_flows = ["out"];
    formulas     = [(["out"; "ued"], Or[F["gin";"ued"]; F["gpm_fl"]]); ]};

   {name         = "Switch";
    faults       = ["ued";];
    input_flows  = ["swin1"; "swin2"];
    basic_events = ["sw_fl"];
    event_info   = [(1.0e-6, 1.0)];
    output_flows = ["out1"; "out2"];
    formulas     = [(["out1"; "ued"], Or[F["swin1";"ued"]; F["sw_fl"]]);
                  (["out2"; "ued"], Or[F["swin2";"ued"]; F["sw_fl"]]);]};

   {name         = "DME";
    faults       = ["ued";];
    input_flows  = [];
    basic_events = ["dme_fl"];
    event_info   = [(1.0e-6, 1.0)];
    output_flows = ["out"];
    formulas     = [(["out"; "ued"], F["dme_fl"]);]};

   {name         = "PFD";
    faults       = ["ued";];
    input_flows  = ["pin"];
    basic_events = ["pfd_fl"];
    event_info   = [(2.0e-10, 1.0)];
    output_flows = ["out"];
    formulas     = [(["out"; "ued"], Or[F["pin";"ued"]; F["pfd_fl"]]); ]};
  ];;


let s_model =
  { instances =
      [makeInstance "gpm" "GPM" ();
       makeInstance "sw" "Switch" ();
       makeInstance "dme" "DME" ();
       makeInstance "pfd" "PFD" ();
      ];
    connections =
      [ (("sw", "swin1"), ("dme", "out"));
        (("sw", "swin2"), ("gpm", "out"));
        (("gpm", "gin"), ("sw", "out1"));
        (("pfd", "pin"), ("sw", "out2"));
      ];
    top_fault = ("pfd", F["out"; "ued"])
  } ;;
```

The top-level fault being analyzed is undetected erroneous data of the PFD.

Figure 15 is the visualization of the physical architecture of the component library and model from above. Automatically generating the physical architecture not only helps the end-user with communication and documentation, but it also helps him visually validate the instantiations and connections he created in the model.

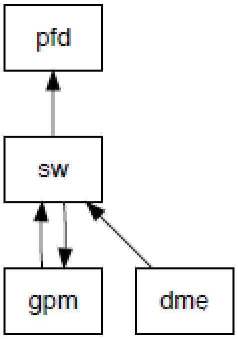


Figure 15. Visualization of the physical architecture.

Figure 16 shows the visualization of the functional architecture. This is a functional view in that the inputs and outputs of all the components are shown along with their connections to one another. The additional details gained by looking at the functional view over the physical view is that what appeared to be an infinite loop in the physical architecture (i.e., the double arrows between the SW and the GPM) is now further defined in the functional architecture. Functionally, the data is flowing from swin1 of SW → out1 of SW → gin of GPM → out of GPM → swin2 of SW → out2 of SW. Automatically generating the functional architecture helps the end-user visually validate the input/output connections.

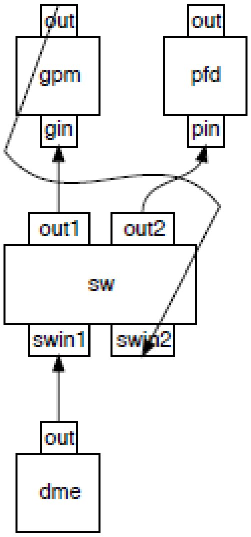


Figure 16. Visualization of the functional architecture.

Figure 17 shows the visualization of the fault propagation. The red edges highlight which part of the system influences the top-level fault. The internal faults of each component are also depicted with an appendage on the side of the component. For example, the component DME has an internal fault, dme_fl, as shown appended on the side. (Note that if this component had more than one fault defined in the library, then the visualization would generate more than one appendages.) The DME output is affected by the internal DME fault, dme_fl, as shown with the red edge, indicating that this fault can propagate to the top-level PFD fault (which, recall, is what this model is analyzing). Another example:

the GPM has an internal fault, gpm_fl; its output is affected by gpm_fl *and* its input, gin, both of which have an effect on the top-level fault. Automatically generating the fault propagation view helps the end-user see which part of the system affects the top-level fault being analyzed.

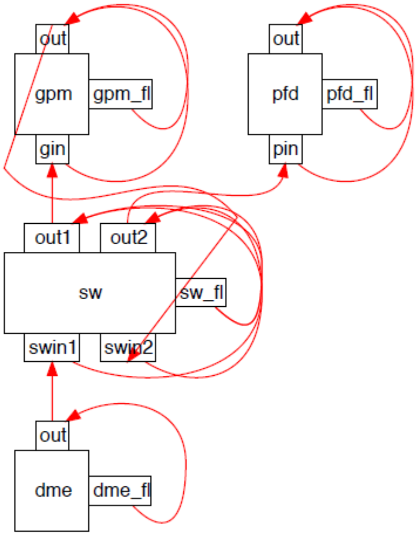


Figure 17. Visualization of the fault propagation.

The SOTERIA tool is now also enhanced with the ability to visualize an automatically synthesized fault tree, as shown in Figure 18.

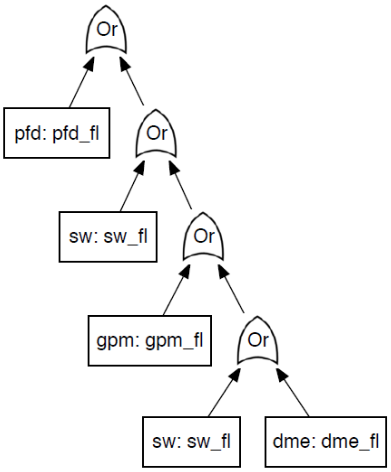
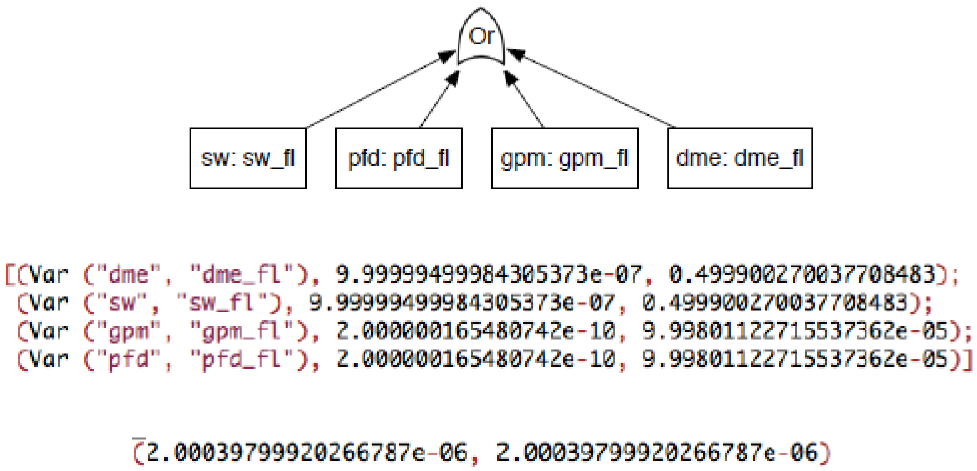


Figure 18. Visualization of the synthesized fault tree.

We further enhanced the qualitative and quantitative analysis results by automatically generating both a simplified fault tree and a list view of the cut sets. In this example, the simplified tree is equivalent to the cut sets. The cut set list is ordered by the importance measure (% contribution) with the biggest failure contributor listed first. The list view is consistent with outputs of popular commercial fault tree tools like Windchill FTA® so safety engineers will be familiar and comfortable with this view. Lastly, the top-level failure probability is calculated.



```
[(Var ("dme", "dme_fl"), 9.99999499984305373e-07, 0.49990027003770848З);
 (Var ("sw", "sw_fl"), 9.99999499984305373e-07, 0.499900270037708483);
 (Var ("gpm", "gpm_fl"), 2.00000165480742e-10, 9.99801122715537362e-05);
 (Var ("pfd", "pfd_fl"), 2.00000165480742e-10, 9.99801122715537362e-05)]
```

```
(2.00039799920266787e-06, 2.00039799920266787e-06)
```

Figure 19. Visualization off the cut sets, list view of the cut sets, and calculation of the top-level failure probability.

# 9 Functionally Integrated Distributed Systems & Other Examples

Here we demonstrate some more examples. The first two systems are inspired by real architectures from B777 and B787, each complex in its own way. We chose to analyze aircraft functions in each of these aircrafts that were interesting from a safety perspective. These are the kinds of problems/situations that engineers find difficult to solve in the sense that the hosted aircraft functions utilize shared resources and failure of functions are considered or contribute to “catastrophic” failure conditions. A Wheel brake and Landing Gear system was also defined for the purposes of stress testing our modeling and fault tree synthesis capabilities. This section includes some material in our paper (Manolios, Siu, Noorman, & Liao, 2017) submitted for publication. The failure probabilities used for the components in these examples are not intended to be representative of the actual equipment modeled, but merely example numbers to illustrate the calculations.

## 9.1 B777

The B777 architecture is interesting because it is the first platform to use IMA. B777 is also complex in that an IMA generally can host multiple applications that need to share system resources, such as computing time, communication bandwidth, and memory, where resource allocation is typically considered as a separate problem from the safety analysis problem. From a safety point of view, one hazard that engineers find hard to analyze and mitigate is the unintended erroneous position display during low RNP approach. The requirement for probability of failure is 1e-9, which corresponds to a catastrophic failure condition.

This section is organized as follows. First, we present the model and analyses of the original B777 sample architecture. Second, we consider an alternative configuration of the sample architecture, which changes the connections between input devices and Input/output Modules (IOMs). Third, we further consider another alternative configuration of the sample architecture, which, in addition to the changes in the previous alternative configuration, also changes the implementation of the Flight Management

Computers (FMC). Fourth, we present a technique to deal with communication bus in the system. Finally, we discuss some findings applicable to the system synthesis based on the study in this section.

### 9.1.1 Original Sample Architecture

A functionally integrated distributed architecture inspired by B777 is shown in Figure 20. In this sample architecture, we are interested in the top-level probability of loss of availability in a simplified Required Navigation Performance (RNP) scenario, where Primary Flight Display #1 (PFD #1) is the sink node of data flows under study. Most of the data flows are self-explanatory from the figure. We further describe the data flows between IOMs and FMCs as follows.

- The output from port A of IOM #1 goes to both port A of FMC #1 and port A of FMC #2.
- The output from port B of IOM #1 goes to both port B of FMC #1 and port B of FMC #2.
- The data flows from IOM #2 and IOM #3 to FMCs are defined similarly.
- Per a connection scheme from FMCs to electronic flight instrument system (EFIS) as described in Section 7.2 of ARINC 702A-4, the output from port A of FMC #1 goes to port A of IOM #4; the output from port B of FMC #2 goes to port B of IOM #4; the output from port A of FMC #2 goes to port A of IOM #5; and the output from port B of FMC #1 goes to port B of IOM #5.

Note that the communication bus is not explicitly modeled here. We will deal with buses later in Section 9.1.4.

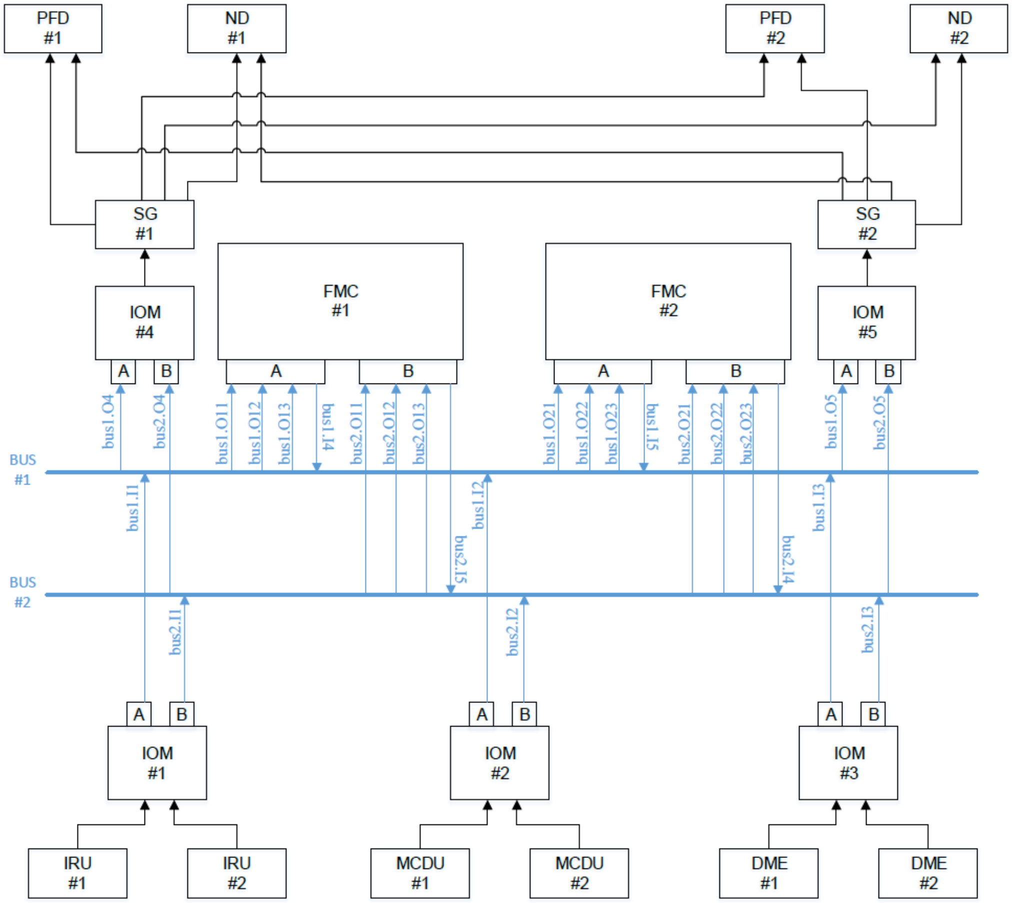


Figure 20. A functionally integrated distributed architecture inspired by B777

To model the sample architecture, we define a component library as follows.

```
let b777_library =
  [
   {name = "DME";
    faults = ["loa"];
    input_flows = [];
    basic_events = ["dme_fl"];
    event_info = [(1e-06, 1.)];
    output_flows = ["out"];
    formulas = [(["out"; "loa"], F ["dme_fl"])]};

   {name = "MCDU";
    faults = ["loa"];
    input_flows = [];
    basic_events = ["mcdu_fl"];
    event_info = [(1e-06, 1.)];
    output_flows = ["out"];
    formulas = [(["out"; "loa"], F ["mcdu_fl"])]};
   {name = "IRU";
    faults = ["loa"];
    input_flows = [];
    basic_events = ["iru_fl"];
    event_info = [(1e-06, 1.)];
    output_flows = ["out"];
    formulas = [(["out"; "loa"], F ["iru_fl"])]};

   {name = "IOM22";
    faults = ["loa"];
    input_flows = ["iom_in1"; "iom_in2"];
    basic_events = ["iom_fl"];
    event_info = [(1e-06, 1.)];
    output_flows = ["A"; "B"];
    formulas =
       [(["A"; "loa"],
         Or [And [F ["iom_in1"; "loa"]; F ["iom_in2"; "loa"]]; F ["iom_fl"]]);
        (["B"; "loa"], F ["A"; "loa"])]};

   {name = "IOM21";
    faults = ["loa"];
    input_flows = ["inA"; "inB"];
    basic_events = ["iom_fl"];
    event_info = [(1e-06, 1.)];
    output_flows = ["out"];
    formulas =
       [(["out"; "loa"],
         Or [And [F ["inA"; "loa"]; F ["inB"; "loa"]]; F ["iom_fl"]])]};

   {name = "FMC";
    faults = ["loa"];
    input_flows = ["inA1"; "inA2"; "inA3"; "inB1"; "inB2"; "inB3"];
    basic_events = ["fmc_fl"];
    event_info = [(2e-10, 1.)];
    output_flows = ["outA"; "outB"];
    formulas =
       [(["outA"; "loa"],
         Or
            [F ["fmc_fl"]; F ["inA1"; "loa"]; F ["inA2"; "loa"];
             F ["inA3"; "loa"]]);
        (["outB"; "loa"],
         Or
            [F ["fmc_fl"]; F ["inB1"; "loa"]; F ["inB2"; "loa"];
             F ["inB3"; "loa"]])]};

   {name = "SG";
    faults = ["loa"];
    input_flows = ["in"];
    basic_events = ["sg_fl"];
    event_info = [(1e-06, 1.)];
    output_flows = ["out"];
```

```
     formulas = [(["out"; "loa"], Or [F ["in"; "loa"]; F ["sg_fl"]])]};

    {name = "PFD";
     faults = ["loa"];
     input_flows = ["in1"; "in2"];
     basic_events = ["pfd_fl"];
     event_info = [(2e-10, 1.)];
     output_flows = ["out"];
     formulas =
        [(["out"; "loa"],
         Or [And [F ["in1"; "loa"]; F ["in2"; "loa"]]; F ["pfd_fl"]])]};

    {name = "ND";
     faults = ["loa"];
     input_flows = ["in1"; "in2"];
     basic_events = ["nd_fl"];
     event_info = [(2e-10, 1.)];
     output_flows = ["out"];
     formulas =
        [(["out"; "loa"],
         Or [And [F ["in1"; "loa"]; F ["in2"; "loa"]]; F ["nd_fl"]])]};
    {name = "BUS5_8";
     faults = ["loa"];
     input_flows = ["i1"; "i2"; "i3"; "i4"; "i5"];
     basic_events = [];
     event_info = [];
     output_flows = ["o11"; "o12"; "o13"; "o21"; "o22"; "o23"; "o4"; "o5"];
     formulas =
        [(["o11"; "loa"], F ["i1"; "loa"]); (["o21"; "loa"], F ["o11"; "loa"]);
         (["o12"; "loa"], F ["i2"; "loa"]); (["o22"; "loa"], F ["o12"; "loa"]);
         (["o13"; "loa"], F ["i3"; "loa"]); (["o23"; "loa"], F ["o13"; "loa"]);
         (["o4"; "loa"], F ["i4"; "loa"]); (["o5"; "loa"], F ["i5"; "loa"])]};

   ];;
```

Note that the component type FMC has two output data flows, outA and outB. In the component library, a separate formula is defined for each of the output data flows. More specifically, the output data flow associated with outA depends on input data flows from port A and FMC internal failure, and the output data flow associated with outB depends on input data flows from port B and FMC internal failure. With this implementation, the FMC ports could be used as independent sources with the caveat that an internal FMC failure would affect both ports. This is one possible implementation of the FMC operation; other implementations are also possible and are discuss later within this section. We will discuss another implementation in the alternative configuration of the sample architecture later.

The model of this sample architecture is presented as follows, where we instantiate the components involved in the safety scenario under study, define data flow connections, and specify the output of PFD #1 as the sink node associated with the top-level event in fault tree analysis. Note that an identified hazard with a failure condition of major or higher requires an analysis, such as an FTA, that shows compliance with an allowable average quantitative probability. For an example of a hazard that might involve more than one sink node associated with the top-level event refer to Section 9.3.

```
let b777_model =
  {instances =
      [makeInstance "iru1" "IRU" ();
       makeInstance "iru2" "IRU" ();
       makeInstance "mcdu1" "MCDU" ();
       makeInstance "mcdu2" "MCDU" ();
       makeInstance "dme1" "DME" ();
       makeInstance "dme2" "DME" ();
```

```
        makeInstance "iom1" "IOM22" ();
        makeInstance "iom2" "IOM22" ();
        makeInstance "iom3" "IOM22" ();
        makeInstance "fmc1" "FMC" ();
        makeInstance "fmc2" "FMC" ();
        makeInstance "iom4" "IOM21" ();
        makeInstance "iom5" "IOM21" ();
        makeInstance "sg1" "SG" ();
        makeInstance "sg2" "SG" ();
        makeInstance "nd1" "ND" ();
        makeInstance "nd2" "ND" ();
        makeInstance "pfd1" "PFD" ();
        makeInstance "pfd2" "PFD" ();
       ];
    connections =
       [ (("iom1", "iom_in1"), ("iru1", "out"));
         (("iom1", "iom_in2"), ("iru2", "out"));
         (("iom2", "iom_in1"), ("mcdu1", "out"));
         (("iom2", "iom_in2"), ("mcdu2", "out"));
         (("iom3", "iom_in1"), ("dme1", "out"));
         (("iom3", "iom_in2"), ("dme2", "out"));
         (("fmc1", "inA1"), ("iom1", "A"));
         (("fmc1", "inA2"), ("iom2", "A"));
         (("fmc1", "inA3"), ("iom3", "A"));
         (("fmc1", "inB1"), ("iom1", "B"));
         (("fmc1", "inB2"), ("iom2", "B"));
         (("fmc1", "inB3"), ("iom3", "B"));
         (("fmc2", "inA1"), ("iom1", "A"));
         (("fmc2", "inA2"), ("iom2", "A"));
         (("fmc2", "inA3"), ("iom3", "A"));
         (("fmc2", "inB1"), ("iom1", "B"));
         (("fmc2", "inB2"), ("iom2", "B"));
         (("fmc2", "inB3"), ("iom3", "B"));
         (("iom4", "inA"), ("fmc1", "outA"));
         (("iom4", "inB"), ("fmc2", "outB"));
         (("iom5", "inA"), ("fmc2", "outA"));
         (("iom5", "inB"), ("fmc1", "outB"));
         (("sg1", "in"), ("iom4", "out"));
         (("sg2", "in"), ("iom5", "out"));
         (("pfd1", "in1"), ("sg1", "out"));
         (("pfd1", "in2"), ("sg2", "out"));
         (("pfd2", "in1"), ("sg1", "out"));
         (("pfd2", "in2"), ("sg2", "out"));
         (("nd1", "in1"), ("sg1", "out"));
         (("nd1", "in2"), ("sg2", "out"));
         (("nd2", "in1"), ("sg1", "out"));
         (("nd2", "in2"), ("sg2", "out"));
       ];
    top_fault =("pfd1", F["out"; "loa"])
   } ;;
```

We used our tool to synthesize the fault tree and generate the minimum cut set for analysis. The cut sets and some comments describing each cut set are presented below.

| Output from tool | Comment |
|---|---|
| `Sum` | |
| ` [Var ("iom1", "iom_fl");` | Loss of IRU |
| `  Var ("iom2", "iom_fl");` | Loss of MCDU |
| `  Var ("iom3", "iom_fl");` | Loss of DME |
| `  Var ("pfd1", "pfd_fl");` | Loss of PFD |
| `  Pro [Var ("dme1", "dme_fl"); Var ("dme2", "dme_fl")];` | Loss of DME |
| `  Pro [Var ("fmc1", "fmc_fl"); Var ("fmc2", "fmc_fl")];` | Loss of FMC |
| `  Pro [Var ("iom4", "iom_fl"); Var ("iom5", "iom_fl")];` | Loss of display data |
| `  Pro [Var ("iom4", "iom_fl"); Var ("sg2", "sg_fl")];` | Loss of display data |
| `  Pro [Var ("iom5", "iom_fl"); Var ("sg1", "sg_fl")];` | Loss of display data |

| | |
|---|---|
| `    Pro [Var ("iru1", "iru_fl"); Var ("iru2", "iru_fl")];` | Loss of IRU |
| `    Pro [Var ("mcdu1", "mcdu_fl"); Var ("mcdu2", "mcdu_fl")];` | Loss of MCDU |
| `    Pro [Var ("sg1", "sg_fl"); Var ("sg2", "sg_fl")]]` | Loss of display data |

From the cut sets, we observe that for each type of input devices (i.e., IRU, MCDU, and DME), there are two ways to completely lose the associated input data; that is, either the associated IOM fails or both (redundant) instances of the input device fail. Take IRU as an example, which corresponds to the terms in the cut sets, `Var ("iom1", "iom_fl")` and `Pro [Var ("iru1", "iru_fl"); Var ("iru2", "iru_fl")]`, respectively.

Another interesting observation is regarding the total loss of display data (between FMC and PFD). From the cut sets, we see that there are four ways to completely lose the display data, i.e.,

(1) SG #1 and SG #2 both fail

(2) IOM #4 and IOM #5 both fail

(3) SG #1 and IOM #5 both fail

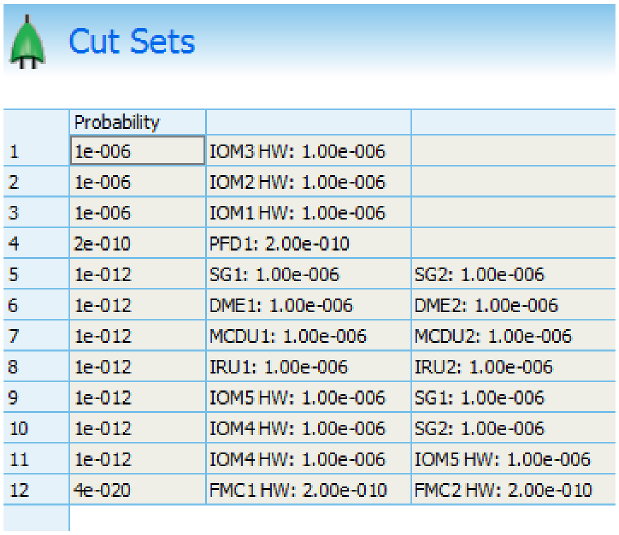
(4) SG #2 and IOM #4 both fail

Whereas (1) and (2) above are obvious since they correspond to failures of both redundant instances of a component, (3) and (4) are less obvious. In fact, the latter two scenarios could be omitted by users who derive cut sets manually, because (3) and (4) are not intuitive (especially in a larger system). In this example, our tool helps users to obtain the complete cut sets, based on which the users can further validate each term in the cut sets in the context of its physical meaning.

Our tool calculates the probability of the top-level event as:

(3.00020249934150288e-06, 3.00020249934150288e-06)

The analysis from the SOTERIA tool was compared to the output generated using Windchill FTA®, the commercial fault tree tool. The cut set report generated by Windchill FTA® (Figure 21), as well as the FTA probabilities calculated are consistent with SOTERIA.



Cut Sets

| | Probability | | |
|---|---|---|---|
| 1 | 1e-006 | IOM3 HW: 1.00e-006 | |
| 2 | 1e-006 | IOM2 HW: 1.00e-006 | |
| 3 | 1e-006 | IOM1 HW: 1.00e-006 | |
| 4 | 2e-010 | PFD1: 2.00e-010 | |
| 5 | 1e-012 | SG1: 1.00e-006 | SG2: 1.00e-006 |
| 6 | 1e-012 | DME1: 1.00e-006 | DME2: 1.00e-006 |
| 7 | 1e-012 | MCDU1: 1.00e-006 | MCDU2: 1.00e-006 |
| 8 | 1e-012 | IRU1: 1.00e-006 | IRU2: 1.00e-006 |
| 9 | 1e-012 | IOM5 HW: 1.00e-006 | SG1: 1.00e-006 |
| 10 | 1e-012 | IOM4 HW: 1.00e-006 | SG2: 1.00e-006 |
| 11 | 1e-012 | IOM4 HW: 1.00e-006 | IOM5 HW: 1.00e-006 |
| 12 | 4e-020 | FMC1 HW: 2.00e-010 | FMC2 HW: 2.00e-010 |

Figure 21. Cut Set Report from Windchill FTA for a functionally integrated distributed architecture inspired by B777

The SOTERIA tool also calculates the probability and the importance measure of each term in the cut set as shown below:

```
 [(Var ("iom1", "iom_fl"), 9.99999499984305373e-07, 0.33331066826448863);
  (Var ("iom2", "iom_fl"), 9.99999499984305373e-07, 0.33331066826448863);
  (Var ("iom3", "iom_fl"), 9.99999499984305373e-07, 0.33331066826448863);
  (Var ("pfd1", "pfd_fl"), 2.000000165480742e-10, 6.66621725006798845e-05);
  (Pro [Var ("dme1", "dme_fl"); Var ("dme2", "dme_fl")],
   9.99998999968860778e-13, 3.3331050160392335e-07);
  (Pro [Var ("iom4", "iom_fl"); Var ("iom5", "iom_fl")],
   9.99998999968860778e-13, 3.3331050160392335e-07);
  (Pro [Var ("iom4", "iom_fl"); Var ("sg2", "sg_fl")],
   9.99998999968860778e-13, 3.3331050160392335e-07);
  (Pro [Var ("iom5", "iom_fl"); Var ("sg1", "sg_fl")],
   9.99998999968860778e-13, 3.3331050160392335e-07);
  (Pro [Var ("iru1", "iru_fl"); Var ("iru2", "iru_fl")],
   9.99998999968860778e-13, 3.3331050160392335e-07);
  (Pro [Var ("mcdu1", "mcdu_fl"); Var ("mcdu2", "mcdu_fl")],
   9.99998999968860778e-13, 3.3331050160392335e-07);
  (Pro [Var ("sg1", "sg_fl"); Var ("sg2", "sg_fl")],
   9.99998999968860778e-13, 3.3331050160392335e-07);
  (Pro [Var ("fmc1", "fmc_fl"); Var ("fmc2", "fmc_fl")],
   4.00000066192299538e-20, 1.33324356032665548e-14)]
```

We observe that the terms corresponding to internal failures of IOM #1, IOM #2, and IOM #3 dominate the safety performance of the overall system (i.e., each account for 33.3% of the top-level failure probability). Moreover, the three IOMs also introduce potential single points of failure to data flows from input devices, e.g., failure of IOM #1 will lead to total loss of IRU data. Therefore, we next consider an alternative configuration of the connections between input devices and IOMs for the purpose of mitigating the above impacts of IOMs.

### 9.1.2 First Move: Change Connections in the Architecture

We consider an alternative configuration of the connections between input devices and IOMs as shown in Figure 22. More specifically, the connections between the second instance of each input device type and the IOMs are shuffled. The other connections in the architecture remain the same.

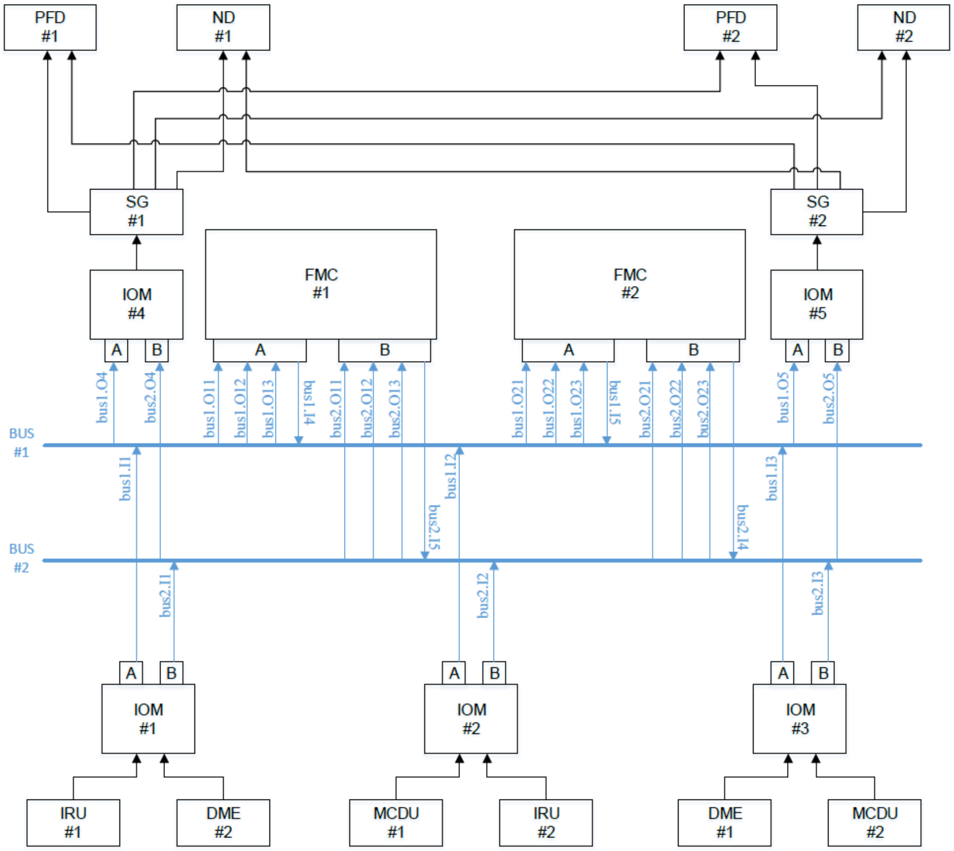


Figure 22. An alternative configuration of the sample architecture inspired by B777

In the original sample architecture (Figure 20), each of the three IOMs at the bottom of the figure takes redundant inputs from two instances of the same type of input device. In the new alternative architecture (Figure 22), each of the three IOMs takes inputs from two different types of input devices. Therefore, we need a new implementation of the IOMs. Accordingly, we added the following new component type to our library.

```
    {name = "IOM22_alt";
     faults = ["loa"];
     input_flows = ["iom_in1"; "iom_in2"];
     basic_events = ["iom_fl"];
     event_info = [(1e-06, 1.)];
     output_flows = ["A"; "B"];
     formulas =
        [(["A"; "loa"], Or [F ["iom_in1"; "loa"]; F ["iom_fl"]]);
         (["B"; "loa"], Or [F ["iom_in2"; "loa"]; F ["iom_fl"]])]};;
```

The model of the alternative sample architecture is as follows. Again, buses are not explicitly modeled in this example.

```
let b777_model_shuffle =
  { instances =
      [makeInstance "iru1" "IRU" ();
       makeInstance "iru2" "IRU" ();
       makeInstance "mcdu1" "MCDU" ();
       makeInstance "mcdu2" "MCDU" ();
       makeInstance "dme1" "DME" ();
       makeInstance "dme2" "DME" ();
```

```
        makeInstance "iom1" "IOM22_alt" ();    (* New type for this example *)
        makeInstance "iom2" "IOM22_alt" ();
        makeInstance "iom3" "IOM22_alt" ();
        makeInstance "fmc1" "FMC" ();
        makeInstance "fmc2" "FMC" ();
        makeInstance "iom4" "IOM21" ();
        makeInstance "iom5" "IOM21" ();
        makeInstance "sg1" "SG" ();
        makeInstance "sg2" "SG" ();
        makeInstance "nd1" "ND" ();
        makeInstance "nd2" "ND" ();
        makeInstance "pfd1" "PFD" ();
        makeInstance "pfd2" "PFD" ();
       ];
     connections =
       [ (("iom1", "iom_in1"), ("iru1", "out"));
          (("iom1", "iom_in2"), ("dme2", "out"));
          (("iom2", "iom_in1"), ("mcdu1", "out"));
          (("iom2", "iom_in2"), ("iru2", "out"));
          (("iom3", "iom_in1"), ("dme1", "out"));
          (("iom3", "iom_in2"), ("mcdu2", "out"));
          (("fmc1", "inA1"), ("iom1", "A"));
          (("fmc1", "inA2"), ("iom2", "A"));
          (("fmc1", "inA3"), ("iom3", "A"));
          (("fmc1", "inB1"), ("iom1", "B"));
          (("fmc1", "inB2"), ("iom2", "B"));
          (("fmc1", "inB3"), ("iom3", "B"));
          (("fmc2", "inA1"), ("iom1", "A"));
          (("fmc2", "inA2"), ("iom2", "A"));
          (("fmc2", "inA3"), ("iom3", "A"));
          (("fmc2", "inB1"), ("iom1", "B"));
          (("fmc2", "inB2"), ("iom2", "B"));
          (("fmc2", "inB3"), ("iom3", "B"));
          (("iom4", "inA"), ("fmc1", "outA"));
          (("iom4", "inB"), ("fmc2", "outB"));
          (("iom5", "inA"), ("fmc2", "outA"));
          (("iom5", "inB"), ("fmc1", "outB"));
          (("sg1", "in"), ("iom4", "out"));
          (("sg2", "in"), ("iom5", "out"));
          (("pfd1", "in1"), ("sg1", "out"));
          (("pfd1", "in2"), ("sg2", "out"));
          (("pfd2", "in1"), ("sg1", "out"));
          (("pfd2", "in2"), ("sg2", "out"));
          (("nd1", "in1"), ("sg1", "out"));
          (("nd1", "in2"), ("sg2", "out"));
          (("nd2", "in1"), ("sg1", "out"));
          (("nd2", "in2"), ("sg2", "out"));
       ];
     top_fault =("pfd1", F["out";"loa"])
   } ;;
```

The analysis provides the following cut sets.

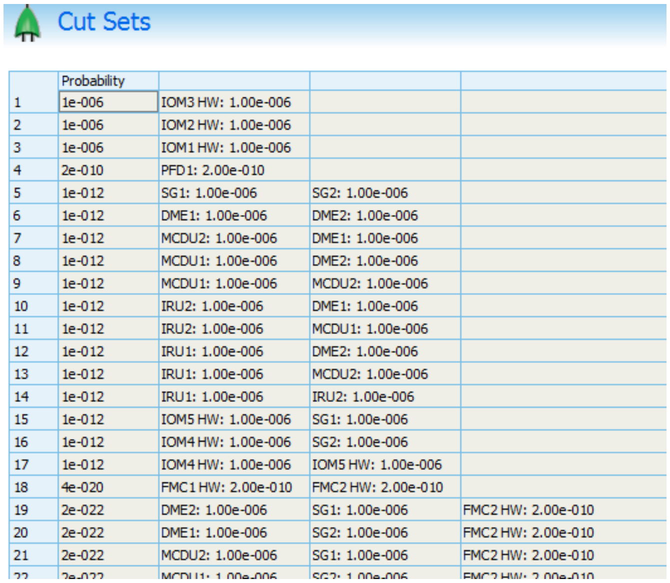
```
  Sum
   [Var ("iom1", "iom_fl"); Var ("iom2", "iom_fl"); Var ("iom3", "iom_fl");
    Var ("pfd1", "pfd_fl");
    Pro [Var ("dme1", "dme_fl"); Var ("dme2", "dme_fl")];
    Pro [Var ("dme1", "dme_fl"); Var ("iru2", "iru_fl")];
    Pro [Var ("dme1", "dme_fl"); Var ("mcdu2", "mcdu_fl")];
    Pro [Var ("dme2", "dme_fl"); Var ("iru1", "iru_fl")];
    Pro [Var ("dme2", "dme_fl"); Var ("mcdu1", "mcdu_fl")];
    Pro [Var ("fmc1", "fmc_fl"); Var ("fmc2", "fmc_fl")];
    Pro [Var ("iom4", "iom_fl"); Var ("iom5", "iom_fl")];
```

```
     Pro [Var ("iom4", "iom_fl"); Var ("sg2", "sg_fl")];
     Pro [Var ("iom5", "iom_fl"); Var ("sg1", "sg_fl")];
     Pro [Var ("iru1", "iru_fl"); Var ("iru2", "iru_fl")];
     Pro [Var ("iru1", "iru_fl"); Var ("mcdu2", "mcdu_fl")];
     Pro [Var ("iru2", "iru_fl"); Var ("mcdu1", "mcdu_fl")];
     Pro [Var ("mcdu1", "mcdu_fl"); Var ("mcdu2", "mcdu_fl")];
     Pro [Var ("sg1", "sg_fl"); Var ("sg2", "sg_fl")];
     Pro
      [Var ("dme1", "dme_fl"); Var ("fmc1", "fmc_fl"); Var ("iom4", "iom_fl")];
     Pro
      [Var ("dme1", "dme_fl"); Var ("fmc1", "fmc_fl"); Var ("sg1", "sg_fl")];
     Pro
      [Var ("dme1", "dme_fl"); Var ("fmc2", "fmc_fl"); Var ("iom5", "iom_fl")];
     Pro
      [Var ("dme1", "dme_fl"); Var ("fmc2", "fmc_fl"); Var ("sg2", "sg_fl")];
     Pro
      [Var ("dme2", "dme_fl"); Var ("fmc1", "fmc_fl"); Var ("iom5", "iom_fl")];
     Pro
      [Var ("dme2", "dme_fl"); Var ("fmc1", "fmc_fl"); Var ("sg2", "sg_fl")];
     Pro
      [Var ("dme2", "dme_fl"); Var ("fmc2", "fmc_fl"); Var ("iom4", "iom_fl")];
     Pro
      [Var ("dme2", "dme_fl"); Var ("fmc2", "fmc_fl"); Var ("sg1", "sg_fl")];
     Pro
      [Var ("fmc1", "fmc_fl"); Var ("iom4", "iom_fl"); Var ("iru1", "iru_fl")];
     Pro
      [Var ("fmc1", "fmc_fl"); Var ("iom4", "iom_fl");
       Var ("mcdu1", "mcdu_fl")];
     Pro [...]; ...]
```

The probability of the top-level event is:

(3.00020849929850045e-06, 3.00020849929950122e-06).

Similar to the original architecture, the analysis generated by the SOTERIA tool and the output generated using Windchill FTA® (Figure 23) are consistent.



Cut Sets

| | Probability | | | |
|---|---|---|---|---|
| 1 | 1e-006 | IOM3 HW: 1.00e-006 | | |
| 2 | 1e-006 | IOM2 HW: 1.00e-006 | | |
| 3 | 1e-006 | IOM1 HW: 1.00e-006 | | |
| 4 | 2e-010 | PFD1: 2.00e-010 | | |
| 5 | 1e-012 | SG1: 1.00e-006 | SG2: 1.00e-006 | |
| 6 | 1e-012 | DME1: 1.00e-006 | DME2: 1.00e-006 | |
| 7 | 1e-012 | MCDU2: 1.00e-006 | DME1: 1.00e-006 | |
| 8 | 1e-012 | MCDU1: 1.00e-006 | DME2: 1.00e-006 | |
| 9 | 1e-012 | MCDU1: 1.00e-006 | MCDU2: 1.00e-006 | |
| 10 | 1e-012 | IRU2: 1.00e-006 | DME1: 1.00e-006 | |
| 11 | 1e-012 | IRU2: 1.00e-006 | MCDU1: 1.00e-006 | |
| 12 | 1e-012 | IRU1: 1.00e-006 | DME2: 1.00e-006 | |
| 13 | 1e-012 | IRU1: 1.00e-006 | MCDU2: 1.00e-006 | |
| 14 | 1e-012 | IRU1: 1.00e-006 | IRU2: 1.00e-006 | |
| 15 | 1e-012 | IOM5 HW: 1.00e-006 | SG1: 1.00e-006 | |
| 16 | 1e-012 | IOM4 HW: 1.00e-006 | SG2: 1.00e-006 | |
| 17 | 1e-012 | IOM4 HW: 1.00e-006 | IOM5 HW: 1.00e-006 | |
| 18 | 4e-020 | FMC1 HW: 2.00e-010 | FMC2 HW: 2.00e-010 | |
| 19 | 2e-022 | DME2: 1.00e-006 | SG1: 1.00e-006 | FMC2 HW: 2.00e-010 |
| 20 | 2e-022 | DME1: 1.00e-006 | SG2: 1.00e-006 | FMC2 HW: 2.00e-010 |
| 21 | 2e-022 | MCDU2: 1.00e-006 | SG1: 1.00e-006 | FMC2 HW: 2.00e-010 |
| 22 | 2e-022 | MCDU1: 1.00e-006 | SG2: 1.00e-006 | FMC2 HW: 2.00e-010 |

Figure 23. Cut Set Report from Windchill FTA for an alternative configuration of the sample architecture inspired by B777

Although we changed the architecture, the dominant terms associated with internal failures of IOM #1, IOM #2, and IOM #3 remain in the cut sets, and the probability of top-level event did not improve. Some further thoughts reveal that we also need to change the implementation of FMCs. In the current implementation of the FMC, the output data flow associated with outA (resp., outB) of FMC only depends on input data flows from port A (resp., port B) and FMC internal failure. If any of the above IOMs fail, neither port A nor port B of an FMC will receive a complete set of input data, hence leading to failures of both output data flows. For example, if IOM #1 fails, then port A of an FMC will miss IRU data and port B of an FMC will miss DME data. Therefore, we next consider another alternative sample architecture, where a different implementation of FMCs is employed.

### 9.1.3 Second Move: Change an Implementation in the Architecture

The second alternative sample architecture is the same as the first one, except that the FMCs employ a different implementation. More specifically, each output data flow of an FMC will depend on input data flows from both port A and port B as well as FMC internal failure. Formally, we define this implementation of FMC as a new component type in our library as follows.

```
{name          = "FMC6_2_alt";
 faults        = ["loa"];
 input_flows   = ["inA1"; "inA2"; "inA3"; "inB1"; "inB2"; "inB3"];
 basic_events  = ["fmc_fl"];
 event_info    = [(2e-10, 1.)];
 output_flows  = ["outA"; "outB"];
 formulas      = [(["outA"; "loa"],
     Or[F ["fmc_fl"]; And [F ["inA1"; "loa"]; F ["inB2"; "loa"]];
        And [F ["inA2"; "loa"]; F ["inB3"; "loa"]];
        And [F ["inA3"; "loa"]; F ["inB1"; "loa"]]]);
    (["outB"; "loa"], F ["outA"; "loa"])]};
```

In the model, we instantiate two FMCs using the new component type FMC6_2_alt.

```
makeInstance "fmc1" "FMC6_2_alt" ();  (* New type for this example *)
makeInstance "fmc2" "FMC6_2_alt" ();
```

The rest of the model remains the same as the first alternative sample architecture. The cut sets and some comments describing each cut set are presented below.

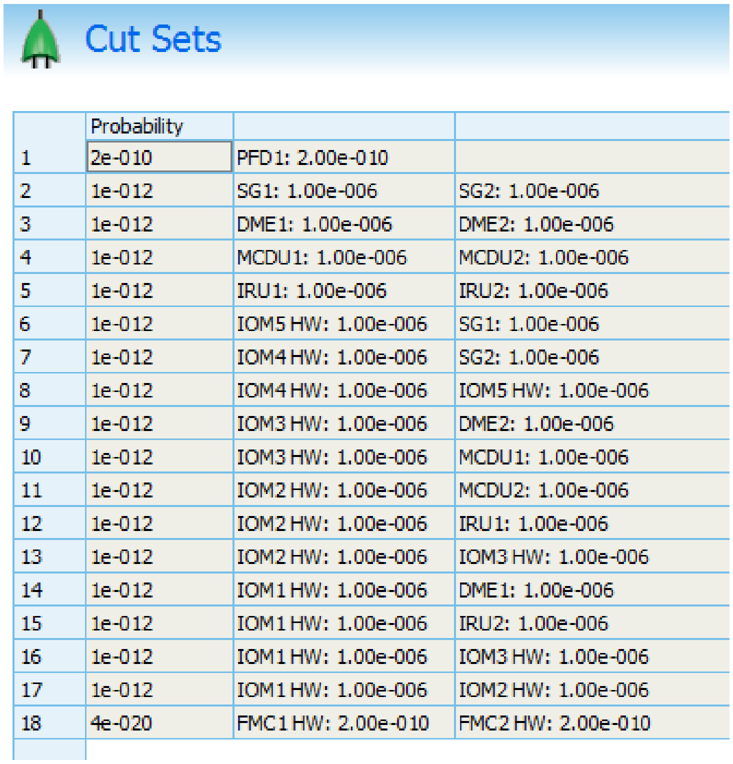
| Output from tool | Comment |
|---|---|
| Sum | |
| [Var ("pfd1", "pfd_fl"); | Loss of PFD |
| Pro [Var ("dme1", "dme_fl"); Var ("dme2", "dme_fl")]; | Loss of DME |
| Pro [Var ("dme1", "dme_fl"); Var ("iom1", "iom_fl")]; | Loss of DME |
| Pro [Var ("dme2", "dme_fl"); Var ("iom3", "iom_fl")]; | Loss of DME |
| Pro [Var ("fmc1", "fmc_fl"); Var ("fmc2", "fmc_fl")]; | Loss of FMC |
| Pro [Var ("iom1", "iom_fl"); Var ("iom2", "iom_fl")]; | Loss of IRU |
| Pro [Var ("iom1", "iom_fl"); Var ("iom3", "iom_fl")]; | Loss of DME |
| Pro [Var ("iom1", "iom_fl"); Var ("iru2", "iru_fl")]; | Loss of IRU |
| Pro [Var ("iom2", "iom_fl"); Var ("iom3", "iom_fl")]; | Loss of MCDU |
| Pro [Var ("iom2", "iom_fl"); Var ("iru1", "iru_fl")]; | Loss of IRU |

|  |  |
|---|---|
| `Pro [Var ("iom2", "iom_fl"); Var ("mcdu2", "mcdu_fl")];` | Loss of MCDU |
| `Pro [Var ("iom3", "iom_fl"); Var ("mcdu1", "mcdu_fl")];` | Loss of MCDU |
| `Pro [Var ("iom4", "iom_fl"); Var ("iom5", "iom_fl")];` | Loss of display data |
| `Pro [Var ("iom4", "iom_fl"); Var ("sg2", "sg_fl")];` | Loss of display data |
| `Pro [Var ("iom5", "iom_fl"); Var ("sg1", "sg_fl")];` | Loss of display data |
| `Pro [Var ("iru1", "iru_fl"); Var ("iru2", "iru_fl")];` | Loss of IRU |
| `Pro [Var ("mcdu1", "mcdu_fl"); Var ("mcdu2", "mcdu_fl")];` | Loss of MCDU |
| `Pro [Var ("sg1", "sg_fl"); Var ("sg2", "sg_fl")]]` | Loss of display data |

The probability of the top-level event is:

(2.15999973584375374e-10, 2.15999973584375374e-10)

The analysis generated by the SOTERIA tool and the output generated using Windchill FTA® (Figure 24) are consistent.



Cut Sets

|  | Probability |  |  |
|---|---|---|---|
| 1 | 2e-010 | PFD1: 2.00e-010 |  |
| 2 | 1e-012 | SG1: 1.00e-006 | SG2: 1.00e-006 |
| 3 | 1e-012 | DME1: 1.00e-006 | DME2: 1.00e-006 |
| 4 | 1e-012 | MCDU1: 1.00e-006 | MCDU2: 1.00e-006 |
| 5 | 1e-012 | IRU1: 1.00e-006 | IRU2: 1.00e-006 |
| 6 | 1e-012 | IOM5 HW: 1.00e-006 | SG1: 1.00e-006 |
| 7 | 1e-012 | IOM4 HW: 1.00e-006 | SG2: 1.00e-006 |
| 8 | 1e-012 | IOM4 HW: 1.00e-006 | IOM5 HW: 1.00e-006 |
| 9 | 1e-012 | IOM3 HW: 1.00e-006 | DME2: 1.00e-006 |
| 10 | 1e-012 | IOM3 HW: 1.00e-006 | MCDU1: 1.00e-006 |
| 11 | 1e-012 | IOM2 HW: 1.00e-006 | MCDU2: 1.00e-006 |
| 12 | 1e-012 | IOM2 HW: 1.00e-006 | IRU1: 1.00e-006 |
| 13 | 1e-012 | IOM2 HW: 1.00e-006 | IOM3 HW: 1.00e-006 |
| 14 | 1e-012 | IOM1 HW: 1.00e-006 | DME1: 1.00e-006 |
| 15 | 1e-012 | IOM1 HW: 1.00e-006 | IRU2: 1.00e-006 |
| 16 | 1e-012 | IOM1 HW: 1.00e-006 | IOM3 HW: 1.00e-006 |
| 17 | 1e-012 | IOM1 HW: 1.00e-006 | IOM2 HW: 1.00e-006 |
| 18 | 4e-020 | FMC1 HW: 2.00e-010 | FMC2 HW: 2.00e-010 |

**Figure 24. Cut Set Report from Windchill FTA for a second alternative configuration of the sample architecture inspired by B777**

Observe that the dominant terms associated with internal failures of IOM #1, IOM #2, and IOM #3 all disappear in the cut sets, and the probability of the top-level event improves significantly. The second alternative architecture effectively removes the potential single point of failure due to the IOMs. The probability and importance measure of each term in cut sets are also calculated.

```
[(Var ("pfd1", "pfd_fl"), 2.000000165480742e-10, 0.925926115773106);
 (Pro [Var ("dme1", "dme_fl"); Var ("dme2", "dme_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("dme1", "dme_fl"); Var ("iom1", "iom_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("dme2", "dme_fl"); Var ("iom3", "iom_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iom1", "iom_fl"); Var ("iom2", "iom_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iom1", "iom_fl"); Var ("iom3", "iom_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iom1", "iom_fl"); Var ("iru2", "iru_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iom2", "iom_fl"); Var ("iom3", "iom_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iom2", "iom_fl"); Var ("iru1", "iru_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iom2", "iom_fl"); Var ("mcdu2", "mcdu_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iom3", "iom_fl"); Var ("mcdu1", "mcdu_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iom4", "iom_fl"); Var ("iom5", "iom_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iom4", "iom_fl"); Var ("sg2", "sg_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iom5", "iom_fl"); Var ("sg1", "sg_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("iru1", "iru_fl"); Var ("iru2", "iru_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("mcdu1", "mcdu_fl"); Var ("mcdu2", "mcdu_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("sg1", "sg_fl"); Var ("sg2", "sg_fl")],
  9.99998999968860778e-13, 0.00462962556603385093);
 (Pro [Var ("fmc1", "fmc_fl"); Var ("fmc2", "fmc_fl")],
  4.00000066192299538e-20, 1.85185238476915286e-10)]
```

From the above results, the remaining single of point of failure in the cut sets is contributed by PFD #1, and its failure probability dominates the overall system safety performance (i.e., accounting for 92.5% of the top-level failure probability). Therefore, if we want to further improve the system safety performance, investigating an alternative configuration of the display(s) would be a logical next step. One potential solution is to consider reversionary modes of displays, which will be discussed later in this report.

### 9.1.4 Modeling Technique to Incorporate Communication Bus

The sample architecture under study involves two communication buses. In safety analysis, the probability of failure associated with bus is usually omitted, so we did not explicitly model buses in the above examples. On the other hand, when synthesizing architectures, modeling a bus may be necessary, e.g., when communication bandwidth is a constraint in the synthesis problem. We propose the following to model a bus. For reference, the input and output data flows associated with the buses are annotated in blue texts in Figure 22. In the component library, we formally define a component type BUS5_8 as follows.

```
    {name         = "BUS5_8";
     faults       = ["loa"];
     input_flows  = ["i1"; "i2"; "i3"; "i4"; "i5"];
     basic_events = [];
     event_info   = [];
     output_flows = ["o11"; "o12"; "o13"; "o21"; "o22"; "o23"; "o4"; "o5"];
     formulas     =
       [(["o11"; "loa"], F ["i1"; "loa"]); (["o21"; "loa"], F ["o11"; "loa"]);
        (["o12"; "loa"], F ["i2"; "loa"]); (["o22"; "loa"], F ["o12"; "loa"]);
        (["o13"; "loa"], F ["i3"; "loa"]); (["o23"; "loa"], F ["o13"; "loa"]);
        (["o4"; "loa"], F ["i4"; "loa"]); (["o5"; "loa"], F ["i5"; "loa"])] };
```

Note that “basic_events” is an empty set in BUS5_8, as we omit potential failures associated with the bus in safety analysis. Also, note that the data flows between IOMs and FMCs, which were (fully) captured by direct connections between IOMs and FMCs in the above examples, are now (partly) captured by the formulas in the bus component. In the model below, we instantiate two instances of BUS5_8 and specify the connections between IOMs, buses, and FMCs.

```
let b777_model_shuffle_w_bus =
  { instances =
      [makeInstance "iru1" "IRU" ();
       makeInstance "iru2" "IRU" ();
       makeInstance "mcdu1" "MCDU" ();
       makeInstance "mcdu2" "MCDU" ();
       makeInstance "dme1" "DME" ();
       makeInstance "dme2" "DME" ();
       makeInstance "iom1" "IOM22_alt" ();
       makeInstance "iom2" "IOM22_alt" ();
       makeInstance "iom3" "IOM22_alt" ();
       makeInstance "fmc1" "FMC6_2_alt" ();
       makeInstance "fmc2" "FMC6_2_alt" ();
        makeInstance "bus1" "BUS5_8" ();            (* Add bus *)
        makeInstance "bus2" "BUS5_8" ();
       makeInstance "iom4" "IOM21" ();
       makeInstance "iom5" "IOM21" ();
       makeInstance "sg1" "SG" ();
       makeInstance "sg2" "SG" ();
       makeInstance "nd1" "ND" ();
       makeInstance "nd2" "ND" ();
       makeInstance "pfd1" "PFD" ();
       makeInstance "pfd2" "PFD" ();
      ];
    connections =
      [ (("iom1", "iom_in1"), ("iru1", "out"));
          (("iom1", "iom_in2"), ("dme2", "out"));
          (("iom2", "iom_in1"), ("mcdu1", "out"));
          (("iom2", "iom_in2"), ("iru2", "out"));
          (("iom3", "iom_in1"), ("dme1", "out"));
          (("iom3", "iom_in2"), ("mcdu2", "out"));
          (("bus1", "i1"), ("iom1", "A"));
          (("bus1", "i2"), ("iom2", "A"));
          (("bus1", "i3"), ("iom3", "A"));
          (("bus2", "i1"), ("iom1", "B"));
          (("bus2", "i2"), ("iom2", "B"));
          (("bus2", "i3"), ("iom3", "B"));
          (("fmc1", "inA1"), ("bus1", "o11"));
          (("fmc1", "inA2"), ("bus1", "o12"));
          (("fmc1", "inA3"), ("bus1", "o13"));
          (("fmc1", "inB1"), ("bus2", "o11"));
          (("fmc1", "inB2"), ("bus2", "o12"));
          (("fmc1", "inB3"), ("bus2", "o13"));
          (("fmc2", "inA1"), ("bus1", "o21"));
          (("fmc2", "inA2"), ("bus1", "o22"));
          (("fmc2", "inA3"), ("bus1", "o23"));
          (("fmc2", "inB1"), ("bus2", "o21"));
          (("fmc2", "inB2"), ("bus2", "o22"));
          (("fmc2", "inB3"), ("bus2", "o23"));
          (("bus1", "i4"), ("fmc1", "outA"));
          (("bus1", "i5"), ("fmc2", "outA"));
          (("bus2", "i5"), ("fmc1", "outB"));
          (("bus2", "i4"), ("fmc2", "outB"));
          (("iom4", "inA"), ("bus1", "o4"));
          (("iom4", "inB"), ("bus2", "o4"));
          (("iom5", "inA"), ("bus1", "o5"));
          (("iom5", "inB"), ("bus2", "o5"));
          (("sg1", "in"), ("iom4", "out"));
          (("sg2", "in"), ("iom5", "out"));
          (("pfd1", "in1"), ("sg1", "out"));
          (("pfd1", "in2"), ("sg2", "out"));
          (("pfd2", "in1"), ("sg1", "out"));
          (("pfd2", "in2"), ("sg2", "out"));
          (("nd1", "in1"), ("sg1", "out"));
          (("nd1", "in2"), ("sg2", "out"));
          (("nd2", "in1"), ("sg1", "out"));
          (("nd2", "in2"), ("sg2", "out"));
      ];
    top_fault =("pfd1", F["out";"loa"])
  } ;;
```

As we expected, the cut sets and top-level probability of this model are the same as those of the second alternative architecture, because the bus does not inject new failure events in safety analysis.

There were no changes in the Windchill FTA® FTA generated to test the second alternative architecture; the 2nd and 3rd modifications are equivalent from a safety perspective, and thus the results discussed within the last section are applicable and consistent with the expected outcome.

### 9.1.5 Discussion

The two alternative sample architectures discussed above imply two types of potential moves in the system synthesis problem, i.e. changing connections between components and changing implementation (e.g., modes) of components. Two other types of potential moves are adding and deleting components from the architecture (which we did not discuss in this example). Further, by comparing the two alternative sample architectures above, we find that only moving the physical component(s) does not necessarily lead to improved system safety performance, and corresponding changes in the implementation of (other) components sometimes are also required to achieve desired results. Finally, the probability of failure in a bus is usually omitted in safety analysis, while explicit modeling of bus can be useful when synthesizing architectures. In a model with a bus that does not introduce new failure events, from a safety analysis perspective, the bus component simply plays the role of routing the data and essentially introduces intermediate variables to the fault tree synthesis algorithm. The intermediate variables relay the effect of failure propagation and will not have an impact on the final cut sets and top-level probability.

All architectures discussed within this section were also manually analyzed using Windchill FTA®. The cut set report generated by Windchill FTA®, as well as the FTA probabilities calculated were consistent with the analysis generated by the SOTERIA tool. This further proves the veracity of our tool. SOTERIA can also generate the cut set with the importance measure of each term, a feature that is useful for safety analysis.

## 9.2 B787

B787 is interesting because it is arguably “the most integrated, most supplier-based cockpit, using the largest display system and the most open architecture of any commercial aircraft developed by Boeing”

(Jensen, 2005). The displays and system integration is done by Rockwell Collins. The IMA, known as the Common Core System (CCS) is supplied by GE. The CCS provides the processing, network and I/O resources to the many aircraft functions. On the B787, Boeing has "set a goal to have as much onboard processing as possible performed by the CCS. In fact, the CCS hosts even the flight management system function. The CSS performs most of the processing for the B787's display system, as well. 'There is a very tight integration between our displays and the CCS,' says Collins' Irmen. 'The majority of the display applications are run on the general processing modules of the CCS. The display information is then sent over to our graphics generation module [GGM], housed in the CCS cabinets. It's sent using ARINC 661 [standard for flight deck display interface] over Ethernet, where it is formatted for display and sent to the displays over a pixel bus. Other systems, such as the CISS [configurable integrated surveillance system], also generate display information and send it to the GGMs over ARINC 661 for display'" (B787 Cockpit: Boeing's Bold Move, 2005).

The B787 has five head down displays (HDD) and two head up displays (HUD). The HDDs are the two PFDs and two NDs, one set of each for the pilot and the co-pilot. The display in the middle is the "aisle stand," which is for the two pilots to input information into the FMCs. In addition to these there are also two head-up displays for added situational awareness. These provide primary flight information (PFI) as well, but show additionally wind shear warnings and some takeoff cues for guidance when visibility is limited. When researching about the B787 cockpit, it would be hard to miss the endless articles about the coolness factor of the B787 cockpit, with the LCD HDDs in landscape format measuring 12 x 9.1", twice the size of the B777, and the two HUDs, positioned so that the pilots can see the necessary flight information while keeping their eyes outside the cockpit. See Figure 25.

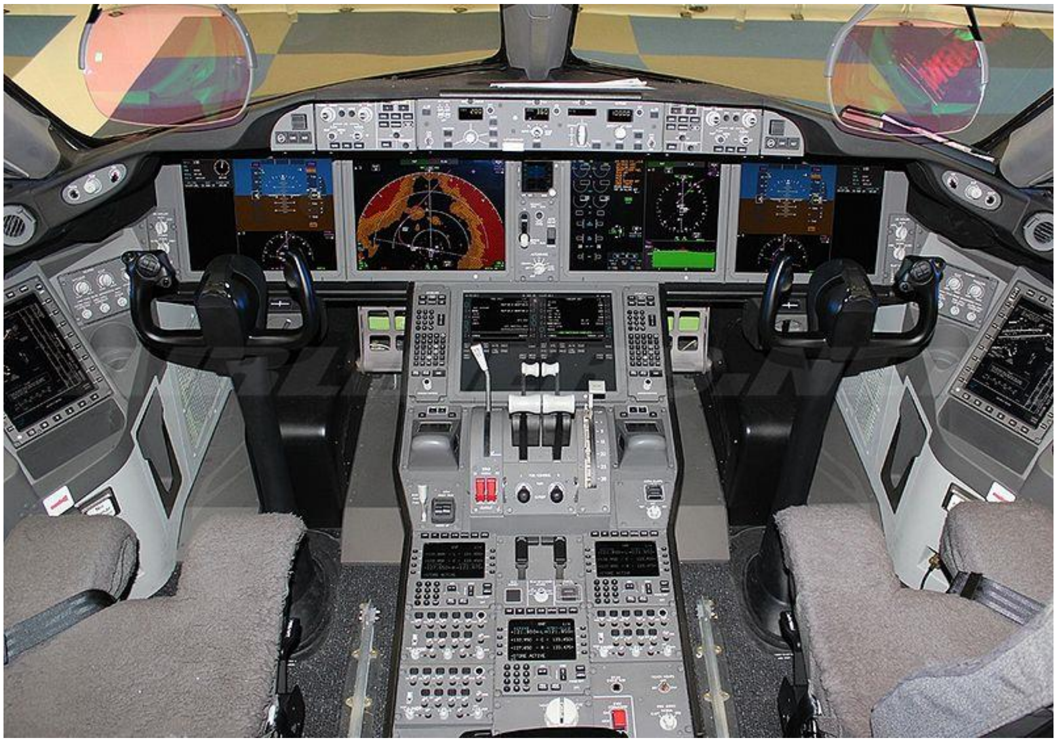


Figure 25. Boeing 787-8 Dreamliner (787-8 Dreamliner, AeroMexico [digital image], 2013)

The safety analysis of the B787 is complex in that a certification applicant can take credit for reversionary flight display, which is a secondary means to provide information initially presented on the PFD by transferring the information to an alternate display (FAA Advisory Circular: Installation of Electronic Display in Part 23 Airplanes, 2011). This is on top of analyzing the safety aspect with a dual FMC configuration. The FMCs transmit data to each other for comparison and validation. For example, if the computed positions between the FMCs differ by more than a set threshold, a message is issued to warm the crew (ARINC Characteristic 702A Advance Flight Management Computer System, 2014). The combination of reversionary flight display and dual FMCs make for a complex analysis problem; we would like for the modeling and fault tree synthesis capability of our tool to be able help an engineer solve a problem with this level of complexity. We analyzed, for instance, the loss of PFI which is influenced by the output of the FMCs. PFI refers to those functions or parameters that are required by the airworthiness and operational rules, such as airspeed, altitude, attitude, and heading (direction) (FAA Advisory Circular: Installation of Electronic Display in Part 23 Airplanes, 2011). FMCs provide functions such as navigation, flight planning, lateral and vertical guidance, performance optimization and prediction, air-ground data link, and pilot interfaces (ARINC Characteristic 702A Advance Flight Management Computer System, 2014). Lateral guidance/lateral steering, for example, generates the roll command and provides outputs related to various flight plans for display.

Section 8.4  (FAA Advisory Circular: Installation of Electronic Display in Part 23 Airplanes, 2011) gives requirements for reversionary display, such as:

- The reversionary flight information should be presented by an independent source and display to prevent complete loss of PFI due to a single failure.
- The reversionary configuration should have two independent displays that incorporate dual-independently powered AHRS and dual ADC subsystems that provide PFIs.
- The reversionary system response time should provide flight critical information on the MFD in less than one second after a single pilot action or an automatic operation.
- These displays should be powered such that any single failure of the power generation and distribution systems cannot remove the display of PFI from both displays.

Figure 26 shows a sample architecture inspired by the B787 IMA architecture. Note that the architecture contains a mixture of IMA and Line Replaceable Unit (LRU) components. While the B787 integrates most of its functionality into the CCS, some complex modules are LRUs, creating a sort of “hybrid” system. For example, the IRUs are LRUs. They communicate via A429, necessitating a data conversion module (DCM).

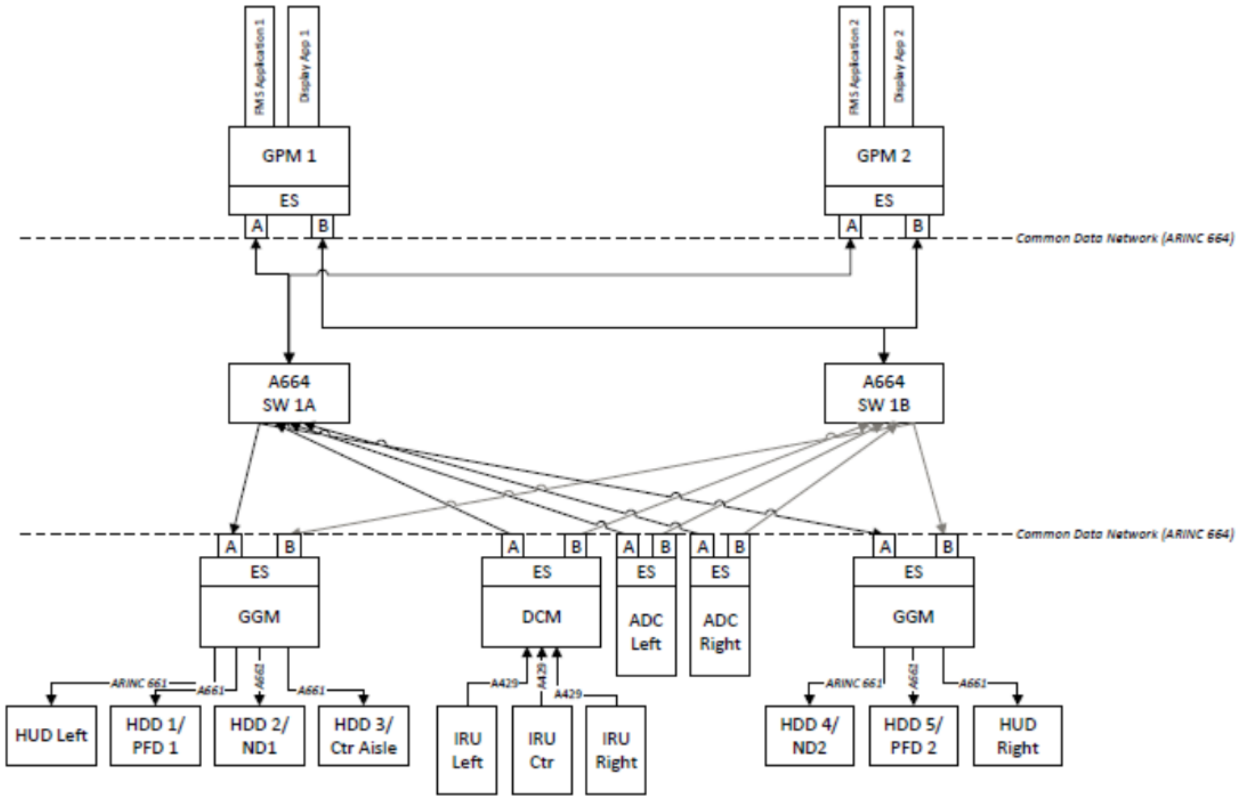


Figure 26. A functionally integrated distributed architecture inspired by B787.

The main goal of our modeling and fault tree synthesis tool is to aid the engineer in coming up with a better architecture while considering constraints determined by safety requirements. The model helps the engineer capture the logic of how the sensor data is processed. For example, it will capture how each display application decides what attitude to display. The source selection and voting will also be modeled to properly build the fault propagation model.

### 9.2.1 Exploring the Many Modes/Implementation

The IMA on the B787, also known as the Common Core System (CCS), can accommodate up to 100 applications (B787 Cockpit: Boeing's Bold Move, 2005). While this is a leap compared with a previous core processor like the C-130 AMP (Avionics Modernization Program) which incorporates about seven applications, it's not the increased number of applications that make the IMA analysis difficult, but the many possible modes and combination of modes that add complexity to the safety analysis. The safety model need to reflect how the system works and each mode may have an impact on the fault tree and cut set. It is also possible that 2 different implementations have no safety impact, i.e., do not help meet safety targets. Prior to the modeling capability and concepts we are developing on this program, what the safety engineer would do was construct the entire fault trees, manually keeping track of where different modes/implementation would impact the fault tree. This can be difficult with large systems, especially when there are multiple ways to build a fault tree. Here what we provide with our modeling is the ability to decompose the problem so that various modes/implementation are managed on the component level, thereby making it easier to keep track of the various modes and implementations. The fault tree, cut sets, and probabilities are done automatically.

The architecture shown in Figure 26 displays just 2 hosted applications: the FMS Application and the Display application. We analyzed the loss of Primary Flight Display (PFD). Here is a list of all possible modes that affect this analysis.

**3 modes for sensor selection:**

Though redundant sensors are common practice in Aviation, one reference on sensor set architecture from a safety perspective is Civil Avionics Systems, 2nd Edition, Chapter *4.7.4 Triplex Architecture* (Moir, Seabridge, & Jukes, 2013).

1. ***Triplex voting***, with 2 possible implementations
   (Here's an example where the implementation doesn't make a difference for the safety analysis)
   a. Median
   b. Average
2. ***Revert to duplex mode***, with 2 possible implementations
   (Here, again, the implementation doesn't make a difference for the safety analysis)
   a. Average
   b. Max or min, depending on which is safer
3. ***Revert to simplex mode / No voting***

**3 modes for selecting the 2 channels:**

Though dual channel is common practice in Avionics, one reference on multiple channels from a safety perspective is Civil Avionics Systems, 2nd Edition, Chapter *4.7.2 Duplex Architecture* (Moir, Seabridge, & Jukes, 2013).

1. ***Channel select logic (Ch A normal, Ch B alternate)***
   The sensor set is replicated to both channels. If one channel fails, there an alternative channel available. Both channels are functionally identical and have the same precision and performance characteristics. This mode offers high availability.
2. ***Channel cross monitor (Ch A & Ch B compare)***
   The sensor set is replicated to both channels, with an additional cross monitor comparison function. Output is only available if both channels agree. This mode offers high integrity.
3. ***Channel COM:MON feature***
   This is a more sophisticated, dual-dual mode. Each channel has a command (COM) and a monitor (MON) lane within it, the command lane being in control and the monitor lane checking for correctness, with cross monitor comparison function to compare the COM and MON on each channel. Two failure modes are possible. The cross-monitor function may produce a false warning, in which case the channel will be deemed to have failed even if the COM and MON lanes are themselves okay. The cross-monitor function may also fail it detect a difference between the COM and MON.

**2 modes in dual FMS:**

Here is a possible dual-FMS implementation, inspired by the Bombardier Challenger 605 – Navigation System Manual[3] and the Civil Avionics Systems, 2nd Edition, Chapter 11.2.10 *Typical FMS Architecture* (Moir, Seabridge, & Jukes, 2013). Dual-FMS system can operate in synchronized (SYNC) or independent (INDEP) modes. By default, dual-FMS installations power up in SYNC mode. When the two FMSs are synchronized, they share pilot entries. When the FMSs are operated in INDEP mode, all parameters must be manually entered in each FMS.

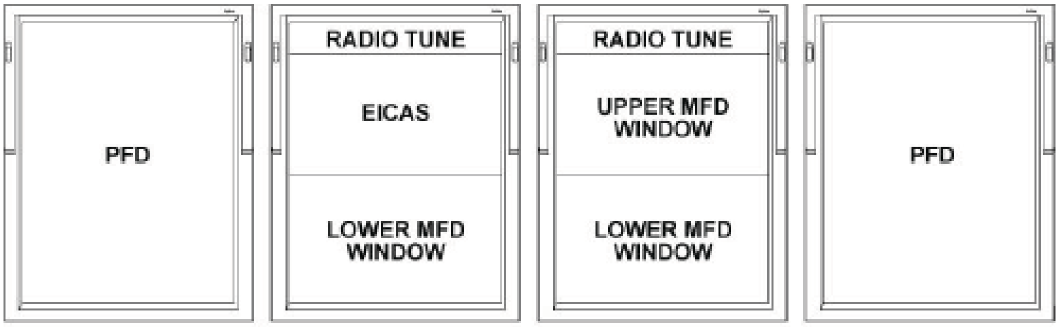
There may a third FMS which makes up a triple-FMS configuration, which could be synchronized to the captain side FMS as a hot-spare. We decided to focus on just 2 FMSs.

The following are possible FMS operations in dual-FMS implementation:

1. ***Normal dual mode***
   Each of the two FMSs interfaces with the Primary Flight Displays (PFDs) on their respective sides.
2. ***Revert to simplex mode***
   If FMS application 1 fails, the PFDs revert to displaying information from FMS Application 2. If FMS application 2 fails, the PFDs revert to displaying information from FMS Application 1.

**3 modes for Primary Flight Display:**

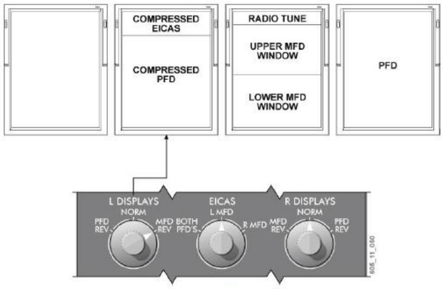
Here's a description of the Primary Flight Display and Multifunction Display from the Bombardier Challenger 605 – Flight Instruments Manual[4]. The two PFDs provide the pilots with the information necessary for the safe operation of the aircraft. They are integrated displays of attitude, air data, and navigation information. The two MFDs can be operated in three basic configurations: split window, chart window, and STAT window. The most common configuration for normal operation is the split window, which consists of four distinct windows: radio tuning window, upper window, lower window, and lower window overlay. During normal operation, the upper window of the left MFD displays the EICAS page, including engine indications, full time system indications, and CAS messages. The upper window of the right MFD displays the SUMMARY page, or the electronic checklist. The lower window of both the left and right MFDs can display synoptic pages or navigation information.



[3] The Bombardier manual is used as our reference because it was publicly available. In spirit, the dual-FMS functionality should be similar to that on the B787.
[4] In spirit, the reversionary display modes should be similar to that on the B787.

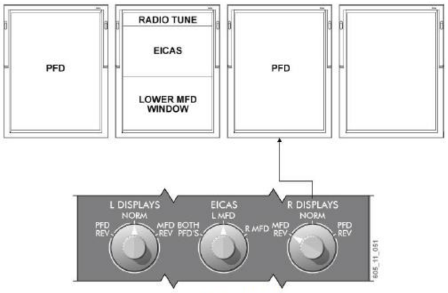
The reversionary mode is selected using knobs on the Reversionary Panel.

1. ***L-PFD Failure***
   Turning the L DISPLAY selector knob to MFD REV reverts the L-MFD to a compressed format, disables the radio tune, and turns off the L-PFD. The MFD includes a compressed EICAS page and a compressed PFD.



Reversion L–PFD Failure

2. ***R-PFD Failure***
   Turning the R DISPLAY selector knob to MFD Rev reverts the R-MFD to a PFD format, disables the radio tune, and turns off the R-PFD. Maintenance and checklist functionality are also lost.



Reversion R–PFD Failure

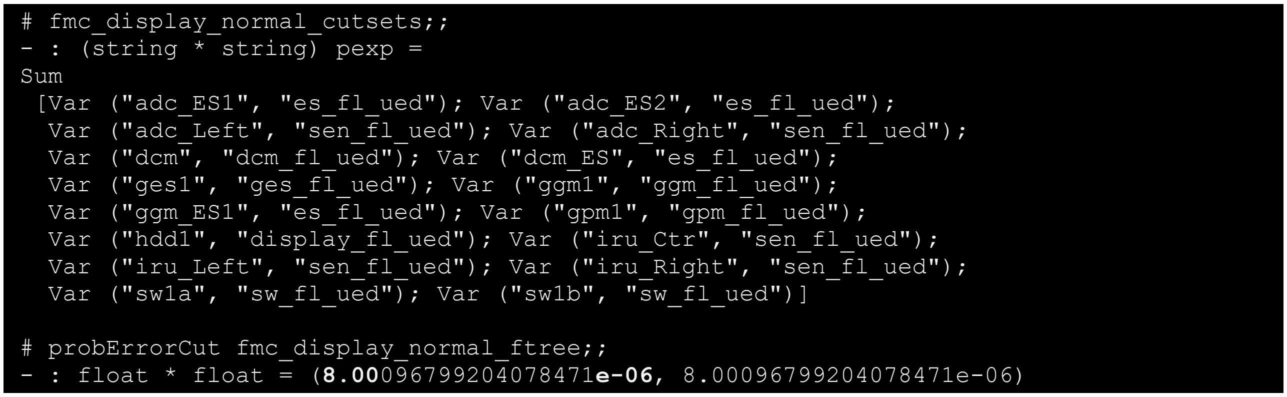
3. ***No reversion***

The point of going through all these available modes is to highlight that when creating a model, it is very important to know exactly how the system works and what's being modeled because the cut sets will be different. The modes also provide a set of feasible moves of component implementation in system synthesis.

The two scenarios analyzed by the SOTERIA tool were the following:

- UED for no sensor voting, cross channel monitoring, dual FMS, no reversionary display mode
- UED for triplex sensor voting, Ch A normal, dual FMS, no reversionary display mode

Below are the results for the integrity cut set and probability obtained using our modeling tool for no sensor voting, cross channel monitoring, dual FMS, no reversionary display mode:



```
# fmc_display_normal_cutsets;;
- : (string * string) pexp =
Sum
 [Var ("adc_ES1", "es_fl_ued"); Var ("adc_ES2", "es_fl_ued");
  Var ("adc_Left", "sen_fl_ued"); Var ("adc_Right", "sen_fl_ued");
  Var ("dcm", "dcm_fl_ued"); Var ("dcm_ES", "es_fl_ued");
  Var ("ges1", "ges_fl_ued"); Var ("ggm1", "ggm_fl_ued");
  Var ("ggm_ES1", "es_fl_ued"); Var ("gpm1", "gpm_fl_ued");
  Var ("hdd1", "display_fl_ued"); Var ("iru_Ctr", "sen_fl_ued");
  Var ("iru_Left", "sen_fl_ued"); Var ("iru_Right", "sen_fl_ued");
  Var ("sw1a", "sw_fl_ued"); Var ("sw1b", "sw_fl_ued")]

# probErrorCut fmc_display_normal_ftree;;
- : float * float = (8.00096799204078471e-06, 8.00096799204078471e-06)
```

Below are the initial results for the integrity cut set and probability for triplex sensor voting, Ch A normal, dual FMS, no reversionary display mode:

```
# fmc_display_triplexVote_ChA_cutsets;;
- : (string * string) pexp =
Sum
   [Var ("adc_ES1", "es_fl_ued"); Var ("adc_ES2", "es_fl_ued");
    Var ("adc_Left", "sen_fl_ued"); Var ("adc_Right", "sen_fl_ued");
    Var ("dcm_ES", "es_fl_ued"); Var ("ges1", "ges_fl_ued");
    Var ("ggm1", "ggm_fl_ued"); Var ("ggm_ES1", "es_fl_ued");
    Var ("gpm1", "gpm_fl_ued"); Var ("hdd1", "display_fl_ued");
    Var ("sw1a", "sw_fl_ued"); Var ("dcm", "dcm_fl_ued");
    Pro
     [Var ("iru_Ctr", "sen_fl_ued");
      Var ("iru_Left", "sen_fl_ued"); Var ("iru_Right", "sen_fl_ued")]]

# probErrorCut fmc_display_triplexVote_ChA_ftree;;
- : float * float = (3.00099549703923056e-06, 3.00099549703923056e-06)
```

The two samples discussed above were also manually analyzed using Windchill FTA® and the results were consistent.

## 9.3 Modeling Multiple Failures

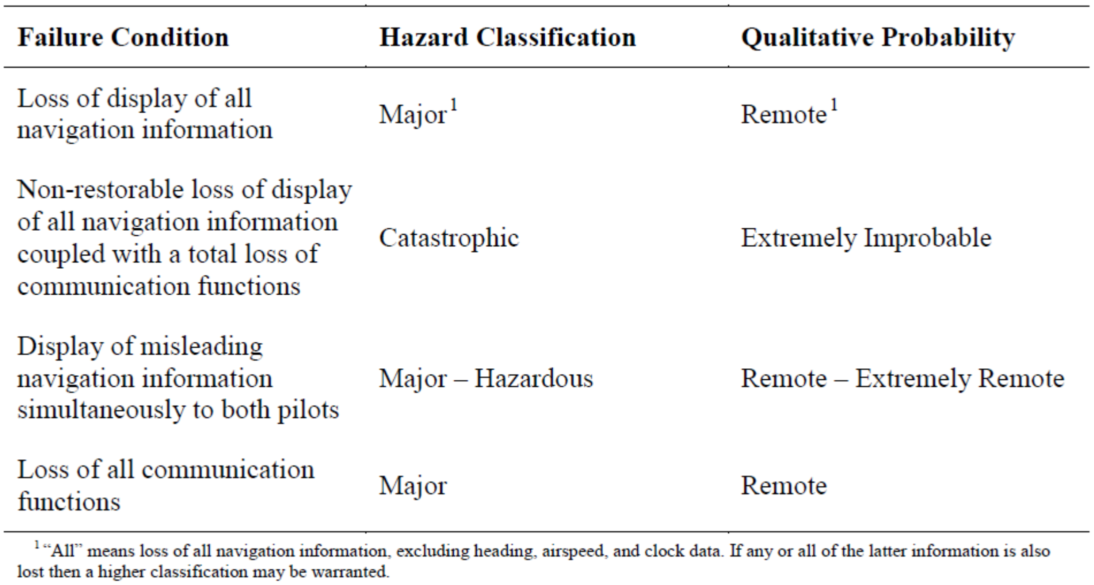
Up until now we have only analyzed one component failure at a time, e.g., the loss of a single display. The failure conditions provided by the FAA AC 25-11B (FAA Advisory Circular: Electronic Flight Displays, 2014) are more relevant to the functional capabilities of an aircraft, which is more realistic to the types of scenarios that an avionics safety engineer would have to analyze.

The FAA AC 25-11B provides guidance for showing compliance with design requirements of electronic flight deck displays, components, and systems installed in airplane. Chapter 4, in particular, talks about the safety aspects of electronic display systems. Here's a quote relevant to the examples we have chosen to model:

*"Using electronic displays and integrated modular avionics allows designers to integrate systems to a much higher degree than was practical with previous flight deck components. Although operating the airplane may become easier as a result of the integration, evaluating the conditions in which the display system could fail and determining the severity of the resulting failure effects may become more complex. The evaluation of the failure conditions should identify the display function and include all causes that could affect that function's display and display equipment."*

There is a series of tables that capture the failure conditions for display systems. Table 4-5 lists examples of safety objectives for navigation, which are interesting conditions for us to analyze with our models of an IMA hosting display and flight management applications.

**Table 4-5. Example Safety Objectives for Certain Navigation and Communication Failure Conditions**



| Failure Condition | Hazard Classification | Qualitative Probability |
|---|---|---|
| Loss of display of all navigation information | Major[1] | Remote[1] |
| Non-restorable loss of display of all navigation information coupled with a total loss of communication functions | Catastrophic | Extremely Improbable |
| Display of misleading navigation information simultaneously to both pilots | Major – Hazardous | Remote – Extremely Remote |
| Loss of all communication functions | Major | Remote |

[1] "All" means loss of all navigation information, excluding heading, airspeed, and clock data. If any or all of the latter information is also lost then a higher classification may be warranted.

To model a failure condition in Table 4-5, like "Loss of display of all navigation information," we need to add a dummy component to our library. Our model needs to instantiate this component and connect it with outputs from all the displays. This component has no basic events; its only purpose is to tie together the needed inputs for analysis.

```
(* Fictitious top-level sink node to take into account different modes *)

    {name         = "PILOT_4DU";
     faults       = ["loa"];
     input_flows  = ["in_pfd1"; "in_nd1"; "in_pfd2"; "in_nd2"];
     basic_events = [];
     event_info   = [];
     output_flows = ["out"];
     formulas     = [(["out"; "loa"],
                        And[F["in_pfd1";"loa"]; F["in_pfd2";"loa"]; F["in_nd1";"loa"]; F["in_nd2";"loa"]])]
    };
```

We first tried out this method of adding a dummy component following a textbook example to verify our tool's output. Then we applied the same method on the B777 and B787 inspired architectures.

Additionally, the validation examples allowed the following:

- Model realistic failure condition and ascertain that our modeling construct is powerful enough to model said failure conditions
- Evaluate what the end-user must do to model the failure condition and compare it with current practices e.g. hand analysis
- Collect any interesting scenarios to revisit with architecture synthesis

### 9.3.1 Textbook EFIS Example

The textbook Civil Avionics Systems (Moir, Seabridge, & Jukes, 2013), has an electronic flight instrument system (EFIS) example in Appendix B. We modeled this system using our tool to verify that adding a dummy component, which we call a "pilot" sink node, and ANDing all the display outputs in the formula, is the correct way of modeling the loss of all displays. Figure 27 is a picture of the architecture. Here's a paraphrase from the text of how the system works:

- There are 3 symbol generators (SGs) that formats images on to 4 display units (DUs). Each SG produces images to both PFD and ND at the same time.
- In normal operation SG#1 sources the images to the captain's side, PFD#1 and ND#1, while SG#2 produces images for the first officer's side, PFD#2 and ND#2. SG#3 is a hot spare and can take over the function of SG#1 or SG#2 if there is a failure in either one.
- Each SG gets its inputs from both the left-hand side and right-hand side sensors.
- If one DU fails, then the SG driving it will reconfigure its outputs so that the image on the adjacent DU will be a composite PFD/ND display
- If an SG fails, then the impacted DUs will select their inputs from SG#3, the hot spare.

For the sake of brevity, we will not include the independent integrated standby instrument (ISIS) in our model.

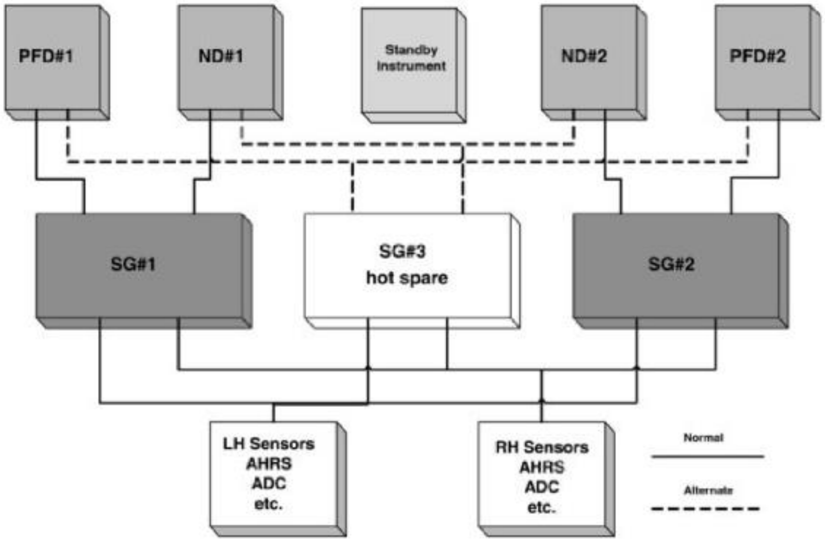


Figure 27. From Civil Avionics Systems (2$^{nd}$ edition) -- Typical EFIS architecture

Below is how we modeled the system. This model reuses the component library we developed for the B777 example, with the addition of three new components to match the textbook example. (Being able to reuse a component library from a previous example additionally demonstrates the broad applicability of our tool.)

```
(*
   Test model: Loss of primary flight data from Main Displays (i.e., PFDs and NDs)
   Figure B.1, p.540, in "Civil Avionics Systems" 2nd edition, by Ian Moir et. al.
*)
```

```
let test_model =
  { instances =
      [makeInstance "ahrs1" "IRU" ();
       makeInstance "ahrs2" "IRU" ();
       makeInstance "adc1" "DME" ();
       makeInstance "adc2" "DME" ();
       makeInstance "iom1" "IOM2_1_NO_INT_FAIL" (); (* New component type for this example *)
       makeInstance "iom2" "IOM2_1_NO_INT_FAIL" ();
       makeInstance "sg1" "SG2_1" ();              (* New component type for this example *)
       makeInstance "sg2" "SG2_1" ();
       makeInstance "sg3" "SG2_1" ();
       makeInstance "pfd1" "PFD" ();
       makeInstance "pfd2" "PFD" ();
       makeInstance "nd1" "ND" ();
       makeInstance "nd2" "ND" ();
       makeInstance "pilot" "PILOT_4DU" ();        (* New component type for this example *)
      ];
    connections =
      [   (("iom1", "inA"), ("ahrs1", "out"));
          (("iom1", "inB"), ("ahrs2", "out"));
          (("iom2", "inA"), ("adc1", "out"));
          (("iom2", "inB"), ("adc2", "out"));
          (("sg1", "in1"), ("iom1", "out"));
          (("sg2", "in1"), ("iom1", "out"));
          (("sg3", "in1"), ("iom1", "out"));
          (("sg1", "in2"), ("iom2", "out"));
          (("sg2", "in2"), ("iom2", "out"));
          (("sg3", "in2"), ("iom2", "out"));
          (("pfd1", "in1"), ("sg1", "out"));
          (("nd1", "in1"), ("sg1", "out"));
          (("pfd2", "in1"), ("sg2", "out"));
          (("nd2", "in1"), ("sg2", "out"));
          (("pfd1", "in2"), ("sg3", "out"));
          (("nd1", "in2"), ("sg3", "out"));
          (("pfd2", "in2"), ("sg3", "out"));
          (("nd2", "in2"), ("sg3", "out"));
          (("pilot", "in_pfd1"), ("pfd1", "out"));
          (("pilot", "in_pfd2"), ("pfd2", "out"));
          (("pilot", "in_nd1"), ("nd1", "out"));
          (("pilot", "in_nd2"), ("nd2", "out"));
      ];
    top_fault = ("pilot", F["out";"loa"])
  } ;;
```

The top-level fault being analyzed is loss of availability of the dummy component, “pilot.” Notice in the connections the pilot takes as input the 4 DUs.

The cut sets obtained by our tool match the results from the text (Figure 28), minus the ISIS which we did not include in our model.

| Output from tool | Comment |
|---|---|
| Sum | |
| [Pro [Var ("adc1", "dme_fl"); Var ("adc2", "dme_fl")]; | Loss of Speed and Altitude |
| Pro [Var ("ahrs1", "iru_fl"); Var ("ahrs2", "iru_fl")]; | Loss of Altitude |
| Pro [Var ("sg1", "sg_fl"); Var ("sg2", "sg_fl"); Var ("sg3", "sg_fl")]; | Loss of all SGs |
| Pro [Var ("nd1", "nd_fl"); Var ("nd2", "nd_fl"); Var ("pfd1", "pfd_fl"); Var ("pfd2", "pfd_fl")]; | Loss of all DUs |
| Pro [Var ("nd1", "nd_fl"); Var ("pfd1", "pfd_fl"); Var ("sg2", "sg_fl"); Var ("sg3", "sg_fl")]; | Loss of opposite side DUs & SGs |
| Pro [Var ("nd2", "nd_fl"); Var ("pfd2", "pfd_fl"); Var ("sg1", "sg_fl"); Var ("sg3", "sg_fl")]] | Loss of opposite side DUs & SGs |

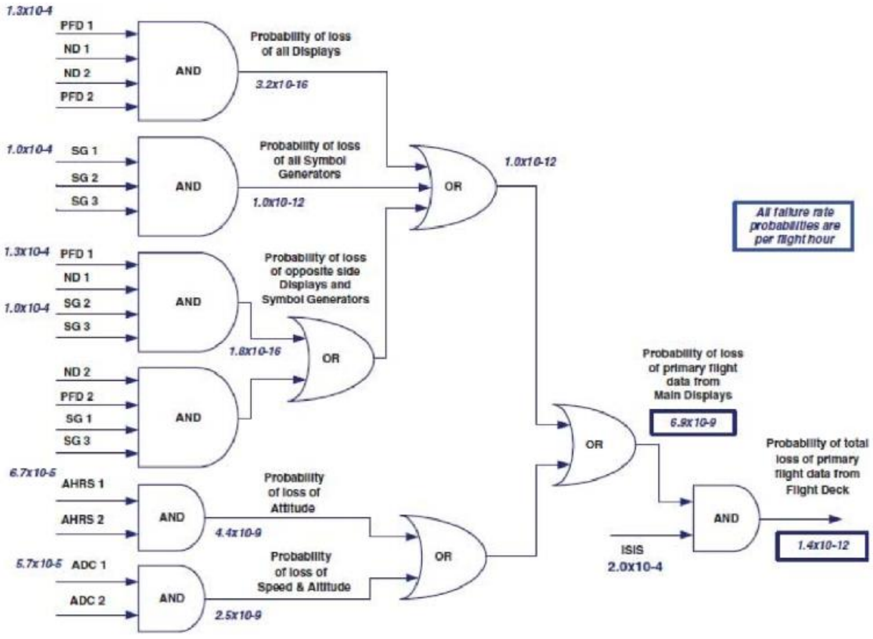

1.3x10-4
PFD 1
ND 1
ND 2
PFD 2
AND
Probability of loss of all Displays
3.2x10-16
1.0x10-4
SG 1
SG 2
SG 3
AND
Probability of loss of all Symbol Generators
1.0x10-12
OR
1.0x10-12
All failure rate probabilities are per flight hour
1.3x10-4
PFD 1
ND 1
1.0x10-4
SG 2
SG 3
AND
Probability of loss of opposite side Displays and Symbol Generators
1.8x10-16
OR
ND 2
PFD 2
SG 1
SG 3
AND
Probability of loss of primary flight data from Main Displays
6.9x10-9
OR
Probability of total loss of primary flight data from Flight Deck
6.7x10-5
AHRS 1
AHRS 2
AND
Probability of loss of Attitude
4.4x10-9
AND
1.4x10-12
ISIS
2.0x10-4
5.7x10-5
ADC 1
ADC 2
AND
Probability of loss of Speed & Altitude
2.5x10-9
OR

Figure 28. From Civil Avionics Systems (2nd edition) -- EFIS cut sets.

### 9.3.2 B777: Loss of Navigation Display

We also modeled the loss of display of all navigation information using our B777 model following the same method of adding a dummy pilot sink node. This model is illustrated in Figure 22 and instantiates components from our B777 component library. In the model connections, the pilot component takes inputs from the 4 displays. For this example, each output flow of the FMC depends on input data from both port A and port B, i.e. the formula for outA is the following:

```
(inA1 AND inB2) OR (inA2 AND inB3) OR (inA3 AND inB1) OR (FMC internal fault)
=
OR (inA1 AND inB2, inA2 AND inB3, inA3 AND inB1, FMC internal fault).
```

```
 (*
  Consider loss of primary flight data from Main Displays (i.e., PFDs and NDs).
  Add pilot top-level sink node.
*)

let b777_model_shuffle_w_bus_pilot =
  { instances = List.append b777_model_shuffle_w_bus.instances [(makeInstance "pilot" "PILOT_4DU" ());];

    connections = List.append b777_model_shuffle_w_bus.connections
      [   (("pilot", "in_pfd1"), ("pfd1", "out"));
          (("pilot", "in_pfd2"), ("pfd2", "out"));
          (("pilot", "in_nd1"), ("nd1", "out"));
          (("pilot", "in_nd2"), ("nd2", "out"));
      ];
    top_fault = ("pilot", F["out";"loa"])
  } ;;
```

The resulting cut sets for loss of display of all navigation information are as follows. Note that the cut set is the same as loss of a single display, except the loss of PFD#1 is now replaced with loss of all 4 displays (highlighted below). This is consistent with our intuition since the logic for the FMC did not change. The top-level probability is now much smaller since this new analysis accounts for the fault-tolerant design of the architecture. Notice, too, that creating a new model for this updated analysis was very simple – with just a few lines the end-user is able to instantiate a new model from a previously defined component library. All he had to do was add the connections and the new cut sets were automatically generated.

| Loss of display of all navigation information | Comment |
|---|---|
| Sum | |
| [Pro [Var ("dme1", "dme_fl"); Var ("dme2", "dme_fl")]; | Loss of DME |
| Pro [Var ("dme1", "dme_fl"); Var ("iom1", "iom_fl")]; | Loss of DME |
| Pro [Var ("dme2", "dme_fl"); Var ("iom3", "iom_fl")]; | Loss of DME |
| Pro [Var ("fmc1", "fmc_fl"); Var ("fmc2", "fmc_fl")]; | Loss of FMC |
| Pro [Var ("iom1", "iom_fl"); Var ("iom2", "iom_fl")]; | Loss of IRU |
| Pro [Var ("iom1", "iom_fl"); Var ("iom3", "iom_fl")]; | Loss of DME |
| Pro [Var ("iom1", "iom_fl"); Var ("iru2", "iru_fl")]; | Loss of IRU |
| Pro [Var ("iom2", "iom_fl"); Var ("iom3", "iom_fl")]; | Loss of MCDU |
| Pro [Var ("iom2", "iom_fl"); Var ("iru1", "iru_fl")]; | Loss of IRU |
| Pro [Var ("iom2", "iom_fl"); Var ("mcdu2", "mcdu_fl")]; | Loss of MCDU |
| Pro [Var ("iom3", "iom_fl"); Var ("mcdu1", "mcdu_fl")]; | Loss of MCDU |
| Pro [Var ("iom4", "iom_fl"); Var ("iom5", "iom_fl")]; | Loss of display data |
| Pro [Var ("iom4", "iom_fl"); Var ("sg2", "sg_fl")]; | Loss of display data |
| Pro [Var ("iom5", "iom_fl"); Var ("sg1", "sg_fl")]; | Loss of display data |
| Pro [Var ("iru1", "iru_fl"); Var ("iru2", "iru_fl")]; | Loss of IRU |
| Pro [Var ("mcdu1", "mcdu_fl"); Var ("mcdu2", "mcdu_fl")]; | Loss of MCDU |
| Pro [Var ("sg1", "sg_fl"); Var ("sg2", "sg_fl")]; | Loss of display data |
| Pro [Var ("nd1", "nd_fl"); Var ("nd2", "nd_fl"); Var ("pfd1", "pfd_fl"); Var ("pfd2", "pfd_fl")]] | Loss of all displays |

| Loss of PFD#1 | Comment |
|---|---|
| Sum | |
| [Var ("pfd1", "pfd_fl"); | Loss of PFD |
| Pro [Var ("dme1", "dme_fl"); Var ("dme2", "dme_fl")]; | Loss of DME |
| Pro [Var ("dme1", "dme_fl"); Var ("iom1", "iom_fl")]; | Loss of DME |
| Pro [Var ("dme2", "dme_fl"); Var ("iom3", "iom_fl")]; | Loss of DME |
| Pro [Var ("fmc1", "fmc_fl"); Var ("fmc2", "fmc_fl")]; | Loss of FMC |
| Pro [Var ("iom1", "iom_fl"); Var ("iom2", "iom_fl")]; | Loss of IRU |
| Pro [Var ("iom1", "iom_fl"); Var ("iom3", "iom_fl")]; | Loss of DME |
| Pro [Var ("iom1", "iom_fl"); Var ("iru2", "iru_fl")]; | Loss of IRU |
| Pro [Var ("iom2", "iom_fl"); Var ("iom3", "iom_fl")]; | Loss of MCDU |
| Pro [Var ("iom2", "iom_fl"); Var ("iru1", "iru_fl")]; | Loss of IRU |
| Pro [Var ("iom2", "iom_fl"); Var ("mcdu2", "mcdu_fl")]; | Loss of MCDU |
| Pro [Var ("iom3", "iom_fl"); Var ("mcdu1", "mcdu_fl")]; | Loss of MCDU |
| Pro [Var ("iom4", "iom_fl"); Var ("iom5", "iom_fl")]; | Loss of display data |
| Pro [Var ("iom4", "iom_fl"); Var ("sg2", "sg_fl")]; | Loss of display data |
| Pro [Var ("iom5", "iom_fl"); Var ("sg1", "sg_fl")]; | Loss of display data |
| Pro [Var ("iru1", "iru_fl"); Var ("iru2", "iru_fl")]; | Loss of IRU |
| Pro [Var ("mcdu1", "mcdu_fl"); Var ("mcdu2", "mcdu_fl")]; | Loss of MCDU |
| Pro [Var ("sg1", "sg_fl"); Var ("sg2", "sg_fl")]] | Loss of display data |

### 9.3.3 B787: Loss of Navigation Display

We also modeled the loss of display of all navigation information using our B787 model using a dummy pilot sink node. We describe the final analysis by building the examples from loss of one display, then 2 displays on the same side, then all displays on the same side, then, loss of 2 displays one on each side, and finally loss of all displays at the same time. The B787 model we will analyze has triplex IRU sensor voting, duplex ADC sensor voting, Ch A is normal and Ch B is available as an alternate, dual FMS, and reversionary display mode (Figure 26).

Below are the cut sets for loss of just one of the displays on the left side, HDD1 (left side PFD). Notice that loss of one display on the left side is dependent on the GPM, GPM ES, GGM, and GGM ES on the left side only.

| Loss of HDD1 (Left PFD) | Comments |
|---|---|
| `Sum` | |
| `[Var ("dcm", "dcm_fl_loa");` | `Loss of DCM` |
| `Var ("dcm_ES", "es_fl_loa");` | `Loss of DCM ES` |
| `Var ("ges1", "ges_fl_loa");` | `Loss of L. GPM ES` |
| `Var ("ggm1", "ggm_fl_loa");` | `Loss of L. GGM` |
| `Var ("ggm_ES1", "es_fl_loa");` | `Loss of L. GGM ES` |
| `Var ("gpm1", "gpm_fl_loa");` | `Loss of L. GPM` |
| `Var ("hdd1", "display_fl_loa");` | `Loss of L. PFD` |
| `Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_ES2", "es_fl_loa")];` | `Loss of both ADC ESs` |
| `Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_Right", "sen_fl_loa")];` | `Loss of ADC L. ES & ADC R.` |
| `Pro [Var ("adc_ES2", "es_fl_loa"); Var ("adc_Left", "sen_fl_loa")];` | `Loss of ADC R. ES & ADC L.` |
| `Pro [Var ("adc_Left", "sen_fl_loa"); Var ("adc_Right", "sen_fl_loa")];` | `Loss of both ADCs` |
| `Pro [Var ("iru_Ctr", "sen_fl_loa");`<br>`Var ("iru_Left", "sen_fl_loa"); Var ("iru_Right", "sen_fl_loa")]` | `Loss of all IRUs` |
| `Pro [Var ("sw1a", "sw_fl_loa"); Var ("sw1b", "sw_fl_loa")];]` | `Loss of both SWs` |

For completeness, below is a list of the cut sets with their probabilities in order of importance.

| Loss of HDD1 | Probability | Importance |
|---|---|---|
| `[(Var ("dcm", "dcm_fl_loa"));` | 9.99995000017e-06 | 0.999909808559 |
| `(Var ("ggm1", "ggm_fl_loa"));` | 2.00000016548e-10 | 1.9998297817e-05 |
| `(Var ("gpm1", "gpm_fl_loa"));` | 2.00000016548e-10 | 1.9998297817e-05 |
| `(Var ("hdd1", "display_fl_loa"));` | 2.00000016548e-10 | 1.9998297817e-05 |
| `(Var ("dcm_ES", "es_fl_loa"));` | 1.00000008274e-10 | 9.99914890849e-06 |
| `(Var ("ges1", "ges_fl_loa"));` | 1.00000008274e-10 | 9.99914890849e-06 |
| `(Var ("ggm_ES1", "es_fl_loa"));` | 1.00000008274e-10 | 9.99914890849e-06 |
| `(Pro [Var ("adc_Left", "sen_fl_loa"); Var ("adc_Right","sen_fl_loa")]);` | 9.99998999969e-13 | 9.9991380817e-08 |
| `(Pro [Var ("sw1a", "sw_fl_loa"); Var ("sw1b", "sw_fl_loa")]);` | 9.99998999969e-13 | 9.9991380817e-08 |
| `(Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_Right", "sen_fl_loa")]);` | 9.99999582725e-17 | 9.99914390876e-12 |
| `(Pro [Var ("adc_ES2", "es_fl_loa"); Var ("adc_Left", "sen_fl_loa")]);` | 9.99999582725e-17 | 9.99914390876e-12 |
| `(Pro [Var ("iru_Ctr", "sen_fl_loa");`<br>`Var ("iru_Left", "sen_fl_loa"); Var ("iru_Right", "sen_fl_loa")]);` | 9.99998499954e-19 | 9.99913308197e-14 |
| `(Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_ES2", "es_fl_loa")]);` | 1.00000016548e-20 | 9.99914973582e-16 |
| `]` | | |

Below are the cut sets for loss of 2 displays: HDD1 (left side PFD) and HDD2 (left side ND). To model this, we created a dummy sink node which took inputs from the 2 displays. The only thing that changed from the previous analysis is that now the cut set contains the loss of HDD1 ANDed with HDD2.

| Loss of HDD1 & HDD2 | Comments |
|---|---|
| `Sum` | |
| `[Var ("dcm", "dcm_fl_loa");` | `Loss of DCM` |
| `Var ("dcm_ES", "es_fl_loa");` | `Loss of DCM ES` |
| `Var ("ges1", "ges_fl_loa");` | `Loss of L. GPM ES` |
| `Var ("ggm1", "ggm_fl_loa");` | `Loss of L. GGM` |
| `Var ("ggm_ES1", "es_fl_loa");` | `Loss of L. GGM ES` |
| `Var ("gpm1", "gpm_fl_loa");` | `Loss of L. GPM` |
| `Pro [Var ("hdd1", "display_fl_loa"); Var ("hdd2", "display_fl_loa")];` | `Loss of HDD1 & HDD2` |
| `Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_ES2", "es_fl_loa")];` | `Loss of both ADC ESs` |
| `Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_Right", "sen_fl_loa")];` | `Loss of ADC L. ES & ADC R.` |
| `Pro [Var ("adc_ES2", "es_fl_loa"); Var ("adc_Left", "sen_fl_loa")];` | `Loss of ADC R. ES & ADC L.` |
| `Pro [Var ("adc_Left", "sen_fl_loa"); Var ("adc_Right", "sen_fl_loa")];` | `Loss of both ADCs` |
| `Pro [Var ("sw1a", "sw_fl_loa"); Var ("sw1b", "sw_fl_loa")];` | `Loss of both SWs` |
| `Pro [Var ("iru_Ctr", "sen_fl_loa");`<br>`Var ("iru_Left", "sen_fl_loa"); Var ("iru_Right", "sen_fl_loa")]]` | `Loss of all IRUs` |

Continuing with the buildup of the examples, below are the cut sets for loss of all left-hand side displays. Our tool makes it very easy to model this – we simply took the model from the previous example and modified it by adding one more input to the dummy sink node. Similar to the change in the previous example, the cut set now differs with the AND of all left-hand side displays: HDD1 AND HDD2 AND HUD_Left.

| Loss of HDD1 & HDD2 & HUD_Left | Comments |
|---|---|
| `Sum` | |
| `[Var ("dcm", "dcm_fl_loa");` | `Loss of DCM` |
| `Var ("dcm_ES", "es_fl_loa");` | `Loss of DCM ES` |
| `Var ("ges1", "ges_fl_loa");` | `Loss of GPM1 ES` |
| `Var ("ggm1", "ggm_fl_loa");` | `Loss of GGM1` |
| `Var ("ggm_ES1", "es_fl_loa");` | `Loss of GGM1 ES` |
| `Var ("gpm1", "gpm_fl_loa");` | `Loss of GPM1` |
| `Pro [Var ("hdd1", "display_fl_loa"); Var ("hdd2", "display_fl_loa");` | `Loss of HDD1 & HDD2 & HUD L.` |
| `Var ("hud_Left", "display_fl_loa")];` | |
| `Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_ES2", "es_fl_loa")];` | `Loss of both ADC ESs` |
| `Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_Right", "sen_fl_loa")];` | `Loss of ADC L. ES & ADC R.` |
| `Pro [Var ("adc_ES2", "es_fl_loa"); Var ("adc_Left", "sen_fl_loa")];` | `Loss of ADC R. ES & ADC L.` |
| `Pro [Var ("adc_Left", "sen_fl_loa"); Var ("adc_Right", "sen_fl_loa")];` | `Loss of both ADCs` |
| `Pro [Var ("sw1a", "sw_fl_loa"); Var ("sw1b", "sw_fl_loa")];` | `Loss of both SWs` |
| `Pro [Var ("iru_Ctr", "sen_fl_loa");` | `Loss of all IRUs` |
| `Var ("iru_Left", "sen_fl_loa"); Var ("iru_Right", "sen_fl_loa")]]` | |

Now let's look at the cut sets for two displays, but on opposite sides. Below are the cut sets for loss of ND on the left-hand side and on the right-hand side. Again, our tool makes it very easy to model this because we can take one of the previous models and modify the inputs to the dummy sink node. The interesting thing to note in the analysis is that every cut set is an AND of a left-hand side component with a right-hand side component. The end-user need only to modify the model in one place and then use our tool to automatically generate the cut sets, compared to having to modify multiple branches in the fault tree by hand for a system like this. Another thing to note is the presence of the DCM ES in all the cut sets so far, which says that this is a potential single point of failure in the way this system is designed. This will be an interesting scenario to test our synthesis algorithm later to see if the synthesis will have enough moves to resolve something like this.

| Loss of HDD2 & HDD4 (opposite side Nav Displays) | Comments |
|---|---|
| `Sum` | |
| `[Var ("dcm", "dcm_fl_loa");` | `Loss of DCM` |
| `Var ("dcm_ES", "es_fl_loa");` | `Loss of DCM ES` |
| `Pro [Var ("ges1", "ges_fl_loa"); Var ("ges2", "ges_fl_loa")];` | `Loss of L. GPM ES & R. GPM ES` |
| `Pro [Var ("ges1", "ges_fl_loa"); Var ("ggm2", "ggm_fl_loa")];` | `Loss of L. GPM ES & R. GGM` |
| `Pro [Var ("ges1", "ges_fl_loa"); Var ("ggm_ES2", "es_fl_loa")];` | `Loss of L. GPM ES & R. GGM ES` |
| `Pro [Var ("ges1", "ges_fl_loa"); Var ("gpm2", "gpm_fl_loa")];` | `Loss of L. GPM ES & R. Nav Disp` |
| `Pro [Var ("ges1", "ges_fl_loa"); Var ("hdd4", "display_fl_loa")];` | `Loss of L. GPM ES & R. GPM2 ES` |
| `Pro [Var ("ges2", "ges_fl_loa"); Var ("ggm1", "ggm_fl_loa")];` | `Loss of R. GPM ES & L. GGM` |
| `Pro [Var ("ges2", "ges_fl_loa"); Var ("ggm_ES1", "es_fl_loa")];` | `Loss of R. GPM ES & L. GGM ES` |
| `Pro [Var ("ges2", "ges_fl_loa"); Var ("gpm1", "gpm_fl_loa")];` | `Loss of R. GPM ES & L. GPM` |
| `Pro [Var ("ges2", "ges_fl_loa"); Var ("hdd2", "display_fl_loa")];` | `Loss of R. GPM ES & L. Nav Disp` |
| `Pro [Var ("ggm1", "ggm_fl_loa"); Var ("ggm2", "ggm_fl_loa")];` | `Loss of L. GGM & R. GGM` |
| `Pro [Var ("ggm1", "ggm_fl_loa"); Var ("ggm_ES2", "es_fl_loa")];` | `Loss of L. GGM & R. GGM ES` |
| `Pro [Var ("ggm1", "ggm_fl_loa"); Var ("gpm2", "gpm_fl_loa")];` | `Loss of L. GGM & R. GPM` |
| `Pro [Var ("ggm1", "ggm_fl_loa"); Var ("hdd4", "display_fl_loa")];` | `Loss of L. GGM & R. Nav Disp` |
| `Pro [Var ("ggm2", "ggm_fl_loa"); Var ("ggm_ES1", "es_fl_loa")];` | `Loss of R. GGM & L. GGM ES` |
| `Pro [Var ("ggm2", "ggm_fl_loa"); Var ("gpm1", "gpm_fl_loa")];` | `Loss of R. GGM & L. GPM` |
| `Pro [Var ("ggm2", "ggm_fl_loa"); Var ("hdd2", "display_fl_loa")];` | `Loss of R. GGM & L. Nav Disp` |
| `Pro [Var ("ggm_ES1", "es_fl_loa"); Var ("ggm_ES2", "es_fl_loa")];` | `Loss of L. GGM ES & R. GGM ES` |

|  |  |
|---|---|
| Pro [Var ("ggm_ES1", "es_fl_loa"); Var ("gpm2", "gpm_fl_loa")]; | Loss of L. GGM ES & R. GPM |
| Pro [Var ("ggm_ES1", "es_fl_loa"); Var ("hdd4", "display_fl_loa")]; | Loss of L. GGM ES & R. Nav Disp |
| Pro [Var ("ggm_ES2", "es_fl_loa"); Var ("gpm1", "gpm_fl_loa")]; | Loss of R. GGM ES & L. GPM |
| Pro [Var ("ggm_ES2", "es_fl_loa"); Var ("hdd2", "display_fl_loa")]; | Loss of R. GGM ES & L. Nav Disp |
| Pro [Var ("gpm1", "gpm_fl_loa"); Var ("gpm2", "gpm_fl_loa")]; | Loss of L. GPM & R. GPM |
| Pro [Var ("gpm1", "gpm_fl_loa"); Var ("hdd4", "display_fl_loa")]; | Loss of L. GPM & R. Nav Disp |
| Pro [Var ("gpm2", "gpm_fl_loa"); Var ("hdd2", "display_fl_loa")]; | Loss of L. GPM & L. Nav Disp |
| Pro [Var ("hdd2", "display_fl_loa");Var ("hdd4", "display_fl_loa")]; | Loss of L. & R. Nav Disp |
| Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_ES2", "es_fl_loa")]; | Loss of L. & R. ADC ES |
| Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_Right", "sen_fl_loa")]; | Loss of L. ADC ES & R. ADC |
| Pro [Var ("adc_ES2", "es_fl_loa"); Var ("adc_Left", "sen_fl_loa")]; | Loss of R. ADC ES & L. ADC |
| Pro [Var ("adc_Left","sen_fl_loa");Var ("adc_Right", "sen_fl_loa")]; | Loss of L. ADC & R. ADC |
| Pro [Var (...); ...]; ...] | and more … |

Finally, below are the cut sets for loss of all seven displays. Similar to the previous example, the cut sets are ANDs of a left-hand side component with a right-hand side component. The added difference here from the previous example is in the cut sets for loss of display components, where the sets are made up of loss of all displays on one side with the loss of one other component from the other side (i.e., loss of all right-hand side displays with the loss of the left-hand side GPM ES.

| Loss of All Displays | Comments |
|---|---|
| Sum |  |
| [Var ("dcm", "dcm_fl_loa"); | Loss of DCM |
| Var ("dcm_ES", "es_fl_loa"); | Loss of DCM ES |
| Pro [Var ("ges1", "ges_fl_loa"); Var ("ges2", "ges_fl_loa")]; | Loss of L. GPM ES & R. GPM ES |
| Pro [Var ("ges1", "ges_fl_loa"); Var ("ggm2", "ggm_fl_loa")]; | Loss of L. GPM ES & R. GGM |
| Pro [Var ("ges1", "ges_fl_loa"); Var ("ggm_ES2", "es_fl_loa")]; | Loss of L. GPM ES & R. GGM ES |
| Pro [Var ("ges1", "ges_fl_loa"); Var ("gpm2", "gpm_fl_loa")]; | Loss of L. GPM ES & R. GPM ES |
| Pro [Var ("ges2", "ges_fl_loa"); Var ("ggm1", "ggm_fl_loa")]; | Loss of R. GPM ES & L. GGM |
| Pro [Var ("ges2", "ges_fl_loa"); Var ("ggm_ES1", "es_fl_loa")]; | Loss of R. GPM ES & L. GGM ES |
| Pro [Var ("ges2", "ges_fl_loa"); Var ("gpm1", "gpm_fl_loa")]; | Loss of R. GPM ES & L. GPM |
| Pro [Var ("ggm1", "ggm_fl_loa"); Var ("ggm2", "ggm_fl_loa")]; | Loss of L. GGM & R. GGM |
| Pro [Var ("ggm1", "ggm_fl_loa"); Var ("ggm_ES2", "es_fl_loa")]; | Loss of L. GGM & R. GGM ES |
| Pro [Var ("ggm1", "ggm_fl_loa"); Var ("gpm2", "gpm_fl_loa")]; | Loss of L. GGM & R. GPM |
| Pro [Var ("ggm2", "ggm_fl_loa"); Var ("ggm_ES1", "es_fl_loa")]; | Loss of R. GGM & L. GGM ES |
| Pro [Var ("ggm2", "ggm_fl_loa"); Var ("gpm1", "gpm_fl_loa")]; | Loss of R. GGM & L. GPM |
| Pro [Var ("ggm_ES1", "es_fl_loa"); Var ("ggm_ES2", "es_fl_loa")]; | Loss of L. GGM ES & R. GGM ES |
| Pro [Var ("ggm_ES1", "es_fl_loa"); Var ("gpm2", "gpm_fl_loa")]; | Loss of L. GGM ES & R. GPM |
| Pro [Var ("ggm_ES2", "es_fl_loa"); Var ("gpm1", "gpm_fl_loa")]; | Loss of R. GGM ES & L. GPM |
| Pro [Var ("gpm1", "gpm_fl_loa"); Var ("gpm2", "gpm_fl_loa")]; | Loss of L. GPM & R. GPM |
| Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_ES2", "es_fl_loa")]; | Loss of L. & R. ADC ES |
| Pro [Var ("adc_ES1", "es_fl_loa"); Var ("adc_Right", "sen_fl_loa")]; | Loss of L. ADC ES & R. ADC |
| Pro [Var ("adc_ES2", "es_fl_loa"); Var ("adc_Left", "sen_fl_loa")]; | Loss of R. ADC ES & L. ADC |
| Pro [Var ("adc_Left","sen_fl_loa");Var ("adc_Right", "sen_fl_loa")]; | Loss of L. ADC & R. ADC |
| Pro [Var ("sw1a", "sw_fl_loa"); Var ("sw1b", "sw_fl_loa")]; | Loss of L. SW & R. SW |
| Pro [Var ("iru_Ctr", "sen_fl_loa");<br>Var ("iru_Left", "sen_fl_loa"); Var ("iru_Right", "sen_fl_loa")]; | Loss of Ctr Aisle Disp, L. IRU & R. IRU |
| Pro [Var ("ges1", "ges_fl_loa"); Var ("hdd4", "display_fl_loa");<br>Var ("hdd5","display_fl_loa");Var ("hud_Right","display_fl_loa")]; | Loss of L. GPM ES, R. Nav Disp, R. Nav Disp, R. HUD |
| Pro [Var ("ges2", "ges_fl_loa"); Var ("hdd1", "display_fl_loa");<br>Var ("hdd2","display_fl_loa");Var ("hud_Left", "display_fl_loa")]; | Loss of R. GPM ES, L. PFD, L. Nav Disp, L. HUD |
| Pro [Var ("ggm1", "ggm_fl_loa"); Var ("hdd4", "display_fl_loa");<br>Var ("hdd5","display_fl_loa");Var ("hud_Right","display_fl_loa")]; | Loss of L. GGM, R. Nav Disp, R. PFD, R. HUD |
| Pro [Var ("ggm2", "ggm_fl_loa"); Var ("hdd1", "display_fl_loa"); Var (...);<br>...];<br>...] | and more … |

## 9.4 Wheel Brake and Landing Gear System

A Wheel brake and Landing Gear system was defined for the purposes of stress testing our modeling capabilities. This system represents a sample system architecture for Landing Gear, Wheel Brakes, and Spoiler Control based on publicly available information, which contain both IMA based components and federated components. The intent of this example is to provide a more complex architecture to model, not generate a real system architecture.

The following sources were used as reference material for the system description and analysis:

- SAE AIR6110 (2011)
- Formal Design and Safety Analysis for AIR6110 Wheel Brake System (2015)
- Spoilers aeronautics (Wikipedia)
- Lufthansa Flight 2904 Incident Description (Wikipedia)
- Another A320 WBS incident (AAIB (UK))

### 9.4.1 System Description

The Wheel Brake and Landing Gear system is an electronically controlled hydraulic system that is used to partially implement the following Aircraft functions:

- Provide Control on the Ground
    - Extend & Retract Landing Gear
        - Extend Landing Gear
        - Retract Landing Gear
        - Provide Landing Gear Status
    - Control Speed
        - Decelerate aircraft on the ground (stopping on the runway)
            - Provide Primary Stopping Force
            - Provide Secondary Stopping Force
            - Decrease Lift / Create Drag / Enhance Braking Effectiveness
            - Remove Forward Thrust
            - Transfer Stopping Forces to Structural Integrity Components
    - Control Direction
        - Provide Pilot Steering

The Landing Gear System consists of subsystems including the Wheel Braking System (WBS) that provide the primary stopping force for the aircraft.  There are other aircraft systems used to decelerate the aircraft in support of the Landing Gear’s WBS.

Figure 29 shows the complete Landing Gear, Wheel Brake, and Spoiler system architecture. For the purposes of this report, we initially modelled the system that handles deployment of the landing gear and deployment of the spoilers (IMA portion of the system), followed by the wheel braking system (Federated portion), and finished by combining both models into one. The green box on the diagram indicates the federated components that control the wheel braking function. The orange box shows the IMA portion of this example that control the landing gear and spoilers deployment functions.

#### *9.4.1.1 Landing Gear*

The Landing Gear System consists of two pairs of wheels (two left and two right). Each pair has one inner, and one outer wheel. Each pair is connected to a single strut connecting it to the fuselage.

Each wheel consists of a tire on a hub. Within each hub is a hydraulically actuated brake that is controlled by the Wheel Brake System (WBS). Each wheel has a single set of sensors as follows: Wheel Speed, Tire Temperature, and Tire Pressure.

**Gear Actuation**

Each strut has an actuator which extends and retracts the gear from the gear bay. Actuation is controlled by a push-button Landing Gear Switch in the cockpit. The switch illuminates green when both gear sensors indicate the gear are down and locked. The switch is not illuminated when both gear sensors indicate the gear is up and locked. The switch flashes green when the gear is neither locked up or down (indicating the gear is in transition).

**Weight on Wheels (on-ground) Detection**

Each strut has a WOW sensor that will activate (indicating the aircraft is on-ground) when the weight on the strut is greater than or equal to 12,000 lbs. (6 tons).

**Landing Gear Bay Doors**

Each landing gear bay has a door that opens when the gear is down (extracted). The doors are activated by the Landing Gear Switch. The door sensor must indicate open and locked before the landing gear is extracted down. Similarly, the landing gear must be up and locked before the bay door is closed.

#### *9.4.1.2 Spoiler System*

The spoiler system consists of hydraulically actuated plates on the top of each wing that extend to “spoil” the airflow over the wing. The purpose of spoilers is to reduce lift, create drag, and provide additional downward force by the aircraft on the landing gear to enhance braking effectiveness.

There are spoilers on each wing. Spoiler activation is controlled by applications hosted on a high integrity avionics platform IMA.

**Spoiler Command / Monitor Function**

One Spoiler Control Application has primary control of the spoilers. The other application monitors the primary application by reading the spoiler sensors. If erroneous spoiler control is detected, the secondary monitor application will send a command to the spoilers. The RIU detects when the secondary application is issuing a command to the spoilers and ignores the primary spoiler command. The pilots are notified of this condition so that they can take precautionary measures such as notifying air traffic control (ATC), apply more braking, and plan to utilize more of the runway.

**Dependency on the WOW sensors and Airspeed**

The spoilers are deployed automatically when both the left and right gear’s WOW sensors indicate the aircraft is on-ground, and the airspeed is greater than 40 knots.

If there is a detected fault of a WOW sensor, the spoiler system will rely on the remaining WOW sensor for activation.

#### *9.4.1.3 Wheel Brake System*

The WBS is used as the primary means to stop the aircraft on the runway in a variety of conditions. The system consists of hydraulically actuated brakes on each wheel. The hydraulic brakes are controlled by a Braking System Control Unit (BSCU) which activates a hydraulic meter valve on the wheel. The meter valve controls the hydraulic pressure for a wheel brake. The WBS supports both manual and automatic brake modes.

**Manual Braking Mode**

Manual braking is activated by the pilot applying the left and/or right brake pedals. The left pedal controls the left gear and the right pedal controls the right gear. The BSCU senses the brake pedal position and sends a command to the meter valve which controls the hydraulic pressure to the brakes on the wheel.

**Auto-Brake Mode**

Automatic braking mode is activated by a selector switch and braking is controlled by pair of BSCUs with no pilot or co-pilot brake pedal assistance. The pilot may select one of three auto-braking levels (Low, Medium, Max) depending on run-way length and conditions. If auto-braking is not selected, the pilot must manually apply brake pedals to engage the brakes. The Auto-Brake feature engages when the wheel rotation is equal to or greater than 72 knots. The pilot can override the auto brake at any time by manually applying either (left or right) brake pedal.

**Dependency on Spoiler System**

The WBS is most effective when the aircraft's Spoiler System is activated. The downward force of the spoilers help the landing gear by reducing tire skidding and increasing wheel brake effectiveness. If a failure to deploy one or more spoilers is detected, then Max auto braking will be used unless overridden by the pilot.

**Anti-Skid Feature**

The Anti-skid feature senses a lack of wheel rotation (less than 40 knots) and automatically reduces the braking force applied to the wheel until the wheel rotation increases (greater than 40 knots). Once the wheel rotation has increased, the commanded braking force is re-applied to the wheel. This feature is activated when the WOW sensor reads on-ground.

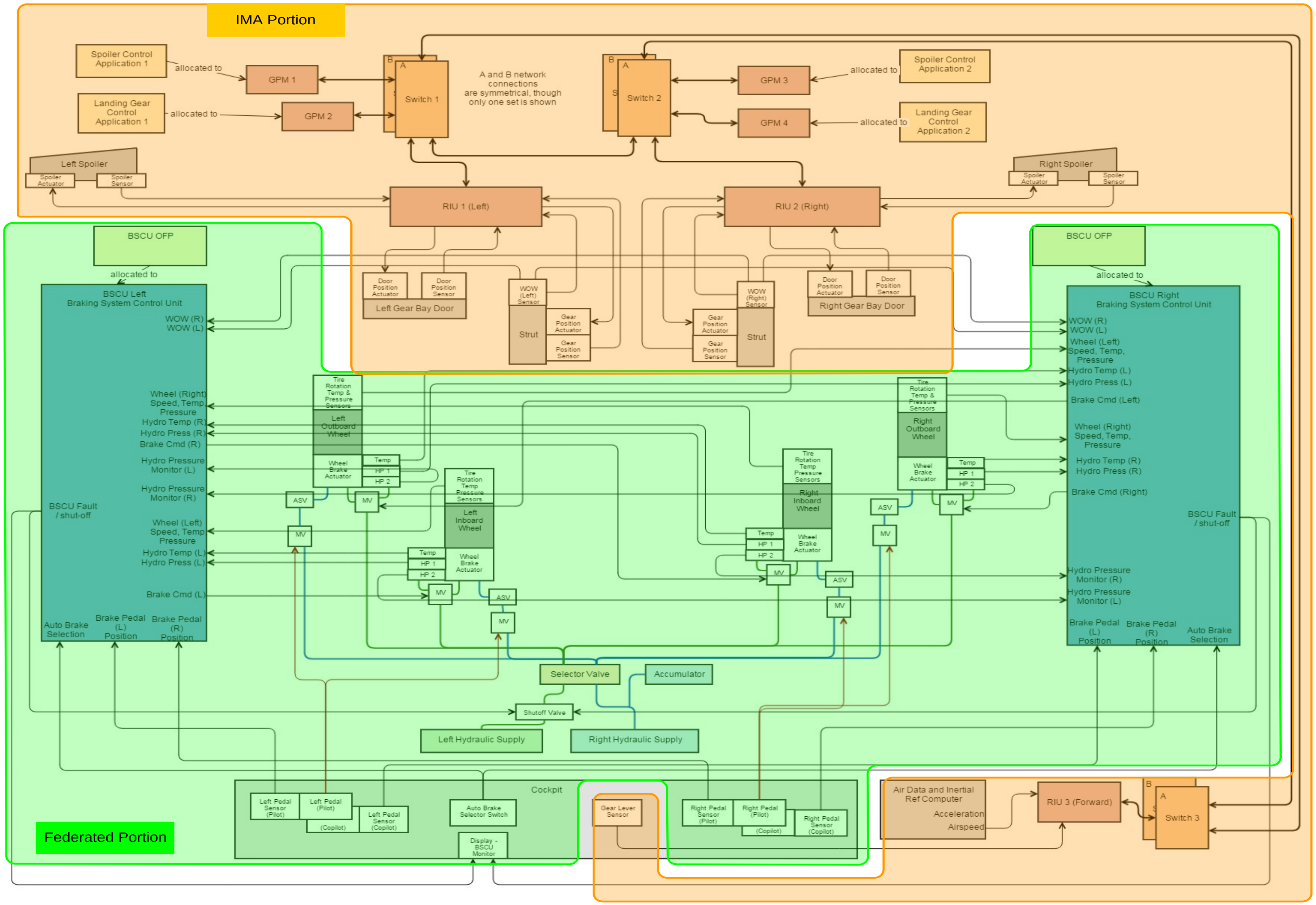


Figure 29. WBS and Land Gear Architecture Diagram

### 9.4.2 Landing Gear Deployment and Spoiler Deployment Modeling Work

The following sections contain comprehensive details of the exercise of modeling the Landing Gear Deployment and Spoiler Deployment functions in our tool. A briefer approach was taken in regards to the Wheel Brake function, and the combination of both models as the exercise of defining components, instances and connection is fully demonstrated while discussing the first two functions. A summary is provided at the end with all findings and observations.

#### *9.4.2.1 Component Models*

The modeling started with building component models for each of the components that support the landing gear deployment and spoiler deployment functions. The Sensors and Effectors (effectors are the actuators in this example) can be modeled generically as they all have the same basic failure propagation properties. Any differences in failure rates or exposure times can be specified at the point the components are instantiated in the model.

```
{name          = "Sen"; (*Sensor*)
  faults       = ["ued"; "loa"];
  input_flows  = [];
  basic_events = ["sen_flt_ued";"sen_flt_loa"];
  event_info   = [(1.0e-6, 1.0);(1.0e-5, 1.0)];
  output_flows = ["out"];
  formulas     = [(["out"; "ued"], F["sen_flt_ued"]);
                  (["out"; "loa"], F["sen_flt_loa"])]
};

{name          = "Eff"; (*Effector*)
  faults       = ["ued"; "loa"];
  input_flows  = ["in"];
  basic_events = ["eff_flt_ued";"eff_flt_loa"];
  event_info   = [(1.0e-6, 1.0);(1.0e-5, 1.0)];
  output_flows = ["out"]; (* output represents the action of effector *)
  formulas     = [(["out";"ued"], Or[ F["in";"ued"]; F["eff_flt_ued"]]) ;
                  (["out";"loa"], Or[ F["in";"loa"]; F["eff_flt_loa"]])]
};
```

Figure 30. Sensor and Effector component models

The RIUs are modeled based on the number of inputs and outputs similar to previous examples. The major difference in this example is that the RIUs have inputs from Sensors that are sent to A664 outputs and have A664 inputs that are sent to the Effectors connected to the RIUs. To simplify the overall system model, we also modeled the RIU A664 End System as part of the RIU model. The following inputs and output definitions were used for RIU 1 and RIU 2:

A664 Inputs / Outputs to other equipment:

1. Spoiler Actuator
2. Gear Door Actuator
3. Gear Position Actuator

A664 Outputs / Inputs from other equipment:

1. Spoiler Sensor
2. WOW Sensor

3. Gear Door Position Sensor
4. Gear Position Sensor

The A664 inputs to the RIU and the A664 outputs from the RIU need to be duplicated since there are A and B network channels. The A664 inputs also need additional duplication to represent the multiple application sources. The numbering scheme is captured in the comments embedded in the model shown in Figure 31. Each “rin<n>” is mapped to “outa<n>” and “outb<n>”. Each “out<n>” is driven by inputs from each copy of the application and each A664 channel. This system uses source selection between the A and B A664 channels (e.g. both must be lost to have loss of function; either being erroneous can cause undetected erroneous functionality). This system also relies on the RIU to perform the redundancy management between the commands from each copy of the controlling application. In this example, the same source selection logic was applied for the application redundancy (both must be lost to have loss of function; either being erroneous can cause undetected erroneous functionality).

```
{name          = "RIU_i4o3"; (* includes the RIU End System *)
  faults       = ["ued"; "loa"];
  input_flows  = ["rin1";"rin2";"rin3";"rin4";
                 (* ina<n>/inb<n> are A664 inputs for each RIU output
                  * to other equipment; "x"/"y" represent the different
                  * applications *)
                 "ina1x";"inb1x";"ina2x";"inb2x";"ina3x";"inb3x";
                 "ina1y";"inb1y";"ina2y";"inb2y";"ina3y";"inb3y"];
  basic_events = ["riu_flt_ued";"riu_flt_loa"];
  event_info   = [(1.0e-6, 1.0);(1.0e-5, 1.0)];
  output_flows = ["out1";"out2";"out3";
                 (* outa<n>/outb<n> are A664 outputs for each RIU input
                  * from other equipment*)
                 "outa1";"outb1";"outa2";"outb2";"outa3";"outb3";
                 "outa4";"outb4"];
  formulas     = [(* A664 Outputs - Note the patterns show up*)
                 (["outa1"; "ued"], Or[F["rin1"; "ued"];F["riu_flt_ued"]]);
                 (["outa1"; "loa"], Or[F["rin1"; "loa"];F["riu_flt_loa"]]);
                 (["outb1"; "ued"], Or[F["rin1"; "ued"];F["riu_flt_ued"]]);
                 (["outb1"; "loa"], Or[F["rin1"; "loa"];F["riu_flt_loa"]]);
                 (["outa2"; "ued"], Or[F["rin2"; "ued"];F["riu_flt_ued"]]);
                 (["outa2"; "loa"], Or[F["rin2"; "loa"];F["riu_flt_loa"]]);
                 (["outb2"; "ued"], Or[F["rin2"; "ued"];F["riu_flt_ued"]]);
                 (["outb2"; "loa"], Or[F["rin2"; "loa"];F["riu_flt_loa"]]);
                 (["outa3"; "ued"], Or[F["rin3"; "ued"];F["riu_flt_ued"]]);
                 (["outa3"; "loa"], Or[F["rin3"; "loa"];F["riu_flt_loa"]]);
                 (["outb3"; "ued"], Or[F["rin3"; "ued"];F["riu_flt_ued"]]);
                 (["outb3"; "loa"], Or[F["rin3"; "loa"];F["riu_flt_loa"]]);
                 (["outa4"; "ued"], Or[F["rin4"; "ued"];F["riu_flt_ued"]]);
                 (["outa4"; "loa"], Or[F["rin4"; "loa"];F["riu_flt_loa"]]);
                 (["outb4"; "ued"], Or[F["rin4"; "ued"];F["riu_flt_ued"]]);
                 (["outb4"; "loa"], Or[F["rin4"; "loa"];F["riu_flt_loa"]]);
                 (* RIU outputs to other equipment *)
                 (* Modeled so that RIU only listens to one source.
                  * Either could be erroneous; both need to be lost  *)
                 (["out1";"ued"],Or[F["ina1x";"ued"];F["inb1x";"ued"];
                   F["ina1y";"ued"];F["inb1y";"ued"];F["riu_flt_ued"]]);
                 (["out1";"loa"],Or[And[F["ina1x";"loa"];F["inb1x";"loa"];F["ina1y";"loa"];
                   F["inb1y";"loa"]];F["riu_flt_loa"]]);
                 (["out2";"ued"],Or[F["ina2x";"ued"];F["inb2x";"ued"];F["ina2y";"ued"];
                   F["inb2y";"ued"];F["riu_flt_ued"]]);
                 (["out2";"loa"],Or[And [F["ina2x";"loa"];F["inb2x";"loa"];F["ina2y";"loa"];
                   F["inb2y";"loa"]];F["riu_flt_loa"]]);
                 (["out3";"ued"],Or[F["ina3x";"ued"];F["inb3x";"ued"];F["ina3y";"ued"];
                   F["inb3y";"ued"];F["riu_flt_ued"]]);
                 (["out3";"loa"],Or[And [F["ina3x";"loa"];F["inb3x";"loa"];F["ina3y";"loa"];
                   F["inb3y";"loa"]];F["riu_flt_loa"]]);
                ]
};
```

Inputs from sensors that are then send out the A664

Data from apps arriving on shared A664 network.

Simplifying assumption. The fault state/mode of the RIU applies to all its outputs

Figure 31. RIU 1 and 2 component model

RIU 3, shown in Figure 32, is quite a bit simpler due to the fact it only has inputs from sensors to output to the A664 network.

```
   {name         = "RIU_i2o0";
    faults       = ["ued"; "loa"];
    input_flows  = ["rin1";"rin2"];
    basic_events = ["riu_flt_ued";"riu_flt_loa"];
    event_info   = [(1.0e-6, 1.0);(1.0e-5, 1.0)];
    output_flows = [(* outa<n>/outb<n> are A664 outputs for each RIU input
                      * from other equipment*)
                    "outa1";"outb1";"outa2";"outb2"];
    formulas     = [(* A664 Outputs - Note the patterns show up*)
                    (["outa1"; "ued"], Or[F["rin1"; "ued"];F["riu_flt_ued"]]);
                    (["outa1"; "loa"], Or[F["rin1"; "loa"];F["riu_flt_loa"]]);
                    (["outb1"; "ued"], Or[F["rin1"; "ued"];F["riu_flt_ued"]]);
                    (["outb1"; "loa"], Or[F["rin1"; "loa"];F["riu_flt_loa"]]);
                    (["outa2"; "ued"], Or[F["rin2"; "ued"];F["riu_flt_ued"]]);
                    (["outa2"; "loa"], Or[F["rin2"; "loa"];F["riu_flt_loa"]]);
                    (["outb2"; "ued"], Or[F["rin2"; "ued"];F["riu_flt_ued"]]);
                    (["outb2"; "loa"], Or[F["rin2"; "loa"];F["riu_flt_loa"]]);
                    ]
  };
```

Figure 32. RIU 3 component model

During the modeling, we determined that the GPM component models needed to be specific to the applications they are hosting. This is because the failure propagation logic is dependent on how the application uses the inputs to generate the output commands.  In section 9.4.5 we follow up on this topic and discuss other possible modeling techniques.

The following inputs and output definitions were used for GPM 1 and 3, the GPMs hosting the Spoiler Control Application:

Spoiler Control Application Inputs:

1. WOW Left
2. WOW Right
3. Spoiler Sensor Left
4. Spoiler Sensor Right
5. Airspeed

Spoiler Control Application Outputs:

1. Spoiler Actuator Left
2. Spoiler Actuator Right

The A664 inputs to the GPM and the A664 outputs from the GPM need to be duplicated since there are A and B network channels. While working the failure propagation logic in the GPM we found there were subtle aspects of the system description that needed to be refined.  This is similar to other examples we have done and in our opinion shows some of the potential value of this type of modeling. This type of modeling forces the designer to think about the system at the data item level and usually results in the need to refine their understanding of how the system functions work. The following assumptions were captured during the modeling process of the Spoiler Control Application and were fed back to the systems description:

- Assume LOA means inability to deploy spoilers when needed for braking and UED means erroneous deployment of spoilers.
- Assume either WOW R and L indicating on ground means spoilers can be deployed. Either one being erroneous could cause UED for the spoiler output. Both need to be lost to lose spoiler output.
- Assume Spoiler Position Sensor is defined as follows: 1 = not-deployed, 0 is deployed. Loss of = stuck at 0, UED = stuck at 1.
- Assume Spoiler Position Sensor feedback is used in application to determine when to command actuator to move. Loss of Spoiler Position contributes to loss of ability to deploy. Undetected Erroneous Spoiler Position does not impact loss of ability to deploy or erroneous deploy.[5]

Figure 33 shows the component model used for GPM 1 and GPM 3.

[5] This is due to the way we defined LOA and UED for this sensor. We defined UED as stuck at 1; sensor at 1 means not deployed. If the function is to deploy the spoiler and the system always thinks the spoiler is not deployed (due to the stuck at 1 fault), the system still can drive the actuator. On the erroneous deployment side, the command to move the actuator also requires an actual application to see an aircraft state where the spoilers are needed.

```
{name          = "GPM_Spoiler";
  faults       = ["ued"; "loa"];
  input_flows  = ["gin1a";"gin2a";"gin3a";"gin4a";"gin1b";"gin2b";"gin3b";
                  "gin4b";"gin5a";"gin5b"];
  basic_events = ["gpm_flt_ued"; "gpm_flt_loa"];
  event_info   = [(3.0e-10, 1.0); (3.0e-5, 1.0)];
  output_flows = ["out1a";"out1b";"out2a";"out2b"];
  formulas     = [(["out1a"; "ued"],Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gin2a"; "ued"];
                    F["gin2b"; "ued"]; F["gin5a"; "ued"]; F["gin5b"; "ued"];F["gpm_flt_ued"]]);
                  (["out1b"; "ued"],Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gin2a"; "ued"];
                    F["gin2b"; "ued"]; F["gin5a"; "ued"]; F["gin5b"; "ued"];F["gpm_flt_ued"]]);
                  (["out2a"; "ued"],Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gin2a"; "ued"];
                    F["gin2b"; "ued"]; F["gin5a"; "ued"]; F["gin5b"; "ued"];F["gpm_flt_ued"]]);
                  (["out2b"; "ued"],Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gin2a"; "ued"];
                    F["gin2b"; "ued"]; F["gin5a"; "ued"]; F["gin5b"; "ued"];F["gpm_flt_ued"]]);
                  (["out1a"; "loa"],Or[And[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];
                    And[F["gin2a"; "loa"]; F["gin2b"; "loa"]]];
                    And[F["gin5a"; "loa"]; F["gin5b"; "loa"]];
                    And[F["gin3a"; "loa"]; F["gin3b"; "loa"]];F["gpm_flt_loa"]]);
                  (["out1b"; "loa"],Or[And[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];
                    And[F["gin2a"; "loa"]; F["gin2b"; "loa"]]];
                    And[F["gin5a"; "loa"]; F["gin5b"; "loa"]];
                    And[F["gin3a"; "loa"]; F["gin3b"; "loa"]];F["gpm_flt_loa"]]);
                  (["out2a"; "loa"],Or[And[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];
                    And[F["gin2a"; "loa"]; F["gin2b"; "loa"]]];
                    And[F["gin5a"; "loa"]; F["gin5b"; "loa"]];
                    And[F["gin4a"; "loa"]; F["gin4b"; "loa"]];F["gpm_flt_loa"]]);
                  (["out2b"; "loa"],Or[And[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];
                    And[F["gin2a"; "loa"]; F["gin2b"; "loa"]]];
                    And[F["gin5a"; "loa"]; F["gin5b"; "loa"]];
                    And[F["gin4a"; "loa"]; F["gin4b"; "loa"]];F["gpm_flt_loa"]]);
                  ]
};
```

Figure 33. GPM 1 and 3 component model

The following inputs and output definitions were used for GPM 2 and 4, the GPMs hosting the Landing Gear Control Application:

Landing Gear Control Application Inputs:

1. Gear Lever
2. Gear Door Position Sensor Left
3. Gear Door Position Sensor Right
4. Gear Position Sensor Left
5. Gear Position Sensor Right

Landing Gear Control Application Outputs:

1. Gear Door Actuator Left
2. Gear Door Actuator Right
3. Gear Position Actuator Left
4. Gear Position Actuator Right

The A664 inputs to the GPM and the A664 outputs from the GPM need to be duplicated since there are A and B network channels. The following assumptions were captured during the modeling process of the Landing Gear Control Application and were fed back to the systems description:

- Assume LOA means inability to deploy gear when needed and UED means erroneous deployment of gear.
- Assume Gear Door Position Sensor is defined as follows: 1 = open, 0 is closed. Loss of = stuck at 0, UED = stuck at 1.
- Assume Door Position Sensor Loss means gear can't be deployed (not sure door is open) and Door Position Sensor UED does not impact Erroneous Deployment.
- Assume Gear Position Sensor is defined as follows: 1 = up, 0 is down. Loss of = stuck at 0, UED = stuck at 1.
- Assume Gear Position Sensor feedback is used in application to determine when to command actuator to move. Loss of Gear Position contributes to loss of ability to deploy. Undetected erroneous Gear Position does not impact loss of ability to deploy or erroneous deploy.

Figure 34 shows the component model used for GPM 2 and GPM 4.

```
{name          = "GPM_Gear";
  faults       = ["ued"; "loa"];
  input_flows  = ["gin1a";"gin2a";"gin3a";"gin4a";"gin5a";"gin1b";"gin2b";
                  "gin3b";"gin4b";"gin5b"];
  basic_events = ["gpm_flt_ued"; "gpm_flt_loa"];
  event_info   = [(3.0e-10, 1.0); (3.0e-5, 1.0)];
  output_flows = ["out1a";"out1b";"out2a";"out2b";"out3a";"out3b";"out4a";
                  "out4b"];
  formulas     = [
                  (["out1a"; "ued"], Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gpm_flt_ued"]]);
                  (["out1b"; "ued"], Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gpm_flt_ued"]]);
                  (["out2a"; "ued"], Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gpm_flt_ued"]]);
                  (["out2b"; "ued"], Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gpm_flt_ued"]]);
                  (["out3a"; "ued"], Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gpm_flt_ued"]]);
                  (["out3b"; "ued"], Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gpm_flt_ued"]]);
                  (["out4a"; "ued"], Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gpm_flt_ued"]]);
                  (["out4b"; "ued"], Or[F["gin1a"; "ued"]; F["gin1b"; "ued"]; F["gpm_flt_ued"]]);
                  (["out1a"; "loa"],Or[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];F["gpm_flt_loa"]]);
                  (["out1b"; "loa"],Or[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];F["gpm_flt_loa"]]);
                  (["out2a"; "loa"],Or[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];F["gpm_flt_loa"]]);
                  (["out2b"; "loa"],Or[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];F["gpm_flt_loa"]]);
                  (["out3a"; "loa"],Or[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];
                    And[F["gin2a"; "loa"]; F["gin2b"; "loa"]];And[F["gin4a"; "loa"];
                    F["gin4b"; "loa"]];F["gpm_flt_loa"]]);
                  (["out3b"; "loa"],Or[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];
                    And[F["gin2a"; "loa"]; F["gin2b"; "loa"]];And[F["gin4a"; "loa"];
                    F["gin4b"; "loa"]];F["gpm_flt_loa"]]);
                  (["out4a"; "loa"],Or[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];
                    And[F["gin3a"; "loa"]; F["gin3b"; "loa"]];And[F["gin5a"; "loa"];
                    F["gin5b"; "loa"]];F["gpm_flt_loa"]]);
                  (["out4b"; "loa"],Or[And[F["gin1a"; "loa"]; F["gin1b"; "loa"]];
                    And[F["gin3a"; "loa"]; F["gin3b"; "loa"]];And[F["gin5a"; "loa"];
                    F["gin5b"; "loa"]];F["gpm_flt_loa"]]);
                  ]
};
```

Figure 34. GPM 2 and 4 component model

The final components modeled were the A664 switches. There are six switches in the architecture (1A, 1B, 2A, 2B, 3A, 3B). A and B network connections are symmetrical, so there are no cross A and B channel links. The switch-to-switch connections shown are the trunk links that would be the same on both network A and B. The Wheel Brake portion of the architecture will be modeled in the future. Like other examples, the A664 switch component models are based on the number of inputs and outputs at the data item level. Since all data is duplicated on the A and B networks, the switch models can be instantiated on either channel and provide the correct data flows. For Switch 1 and Switch 2 it was determined that there were 19 signals that needed to be modeled as inputs and outputs. Four of the signals end up getting multicast to multiple destinations so they are modeled to flow to two different outputs each. These switch models also include the 32-bit CRC protection mechanism that we have modeled in previous examples.

Figure 35 shows the component model used for Switch 1 (A and B) and Switch 2 (A and B). In16 through in19 are the inputs that are multicast to two different outputs.

```
{name          = "Switch1";
  faults       = ["ued";"loa"];
  input_flows  = ["in01";"in02";"in03";"in04";"in05";"in06";"in07";"in08";"in09";"in10";
                  "in11";"in12";"in13";"in14";"in15";"in16";"in17";"in18";"in19"];
  basic_events = ["sw_flt_ued";"sw_flt_loa";"crc32_flt"];
  event_info   = [(1.0e-6, 1.0); (1.0e-5, 1.0); (2.**(-32.), 1.0)];
  output_flows = ["out01";"out02";"out03";"out04";"out05";"out06";"out07";
                  "out08";"out09";"out10";"out11";"out12";"out13";"out14";
                  "out15";"out16m1";"out17m1";"out18m1";"out19m1";
                  "out16m2";"out17m2";"out18m2";"out19m2"];
  formulas     = [(["out01";"ued"],Or[F["in01";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out02";"ued"],Or[F["in02";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out03";"ued"],Or[F["in03";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out04";"ued"],Or[F["in04";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out05";"ued"],Or[F["in05";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out06";"ued"],Or[F["in06";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out07";"ued"],Or[F["in07";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out08";"ued"],Or[F["in08";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out09";"ued"],Or[F["in09";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out10";"ued"],Or[F["in10";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out11";"ued"],Or[F["in11";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out12";"ued"],Or[F["in12";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out13";"ued"],Or[F["in13";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out14";"ued"],Or[F["in14";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out15";"ued"],Or[F["in15";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out16m1";"ued"],Or[F["in16";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out17m1";"ued"],Or[F["in17";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out18m1";"ued"],Or[F["in18";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out19m1";"ued"],Or[F["in19";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out16m2";"ued"],Or[F["in16";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out17m2";"ued"],Or[F["in17";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out18m2";"ued"],Or[F["in18";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out19m2";"ued"],Or[F["in19";"ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out01"; "loa"], Or[F["in01"; "loa"];F["sw_flt_loa"]]);
                  (["out02"; "loa"], Or[F["in02"; "loa"];F["sw_flt_loa"]]);
                  (["out03"; "loa"], Or[F["in03"; "loa"];F["sw_flt_loa"]]);
                  (["out04"; "loa"], Or[F["in04"; "loa"];F["sw_flt_loa"]]);
                  (["out05"; "loa"], Or[F["in05"; "loa"];F["sw_flt_loa"]]);
                  (["out06"; "loa"], Or[F["in06"; "loa"];F["sw_flt_loa"]]);
                  (["out07"; "loa"], Or[F["in07"; "loa"];F["sw_flt_loa"]]);
                  (["out08"; "loa"], Or[F["in08"; "loa"];F["sw_flt_loa"]]);
                  (["out09"; "loa"], Or[F["in09"; "loa"];F["sw_flt_loa"]]);
                  (["out10"; "loa"], Or[F["in10"; "loa"];F["sw_flt_loa"]]);
                  (["out11"; "loa"], Or[F["in11"; "loa"];F["sw_flt_loa"]]);
                  (["out12"; "loa"], Or[F["in12"; "loa"];F["sw_flt_loa"]]);
                  (["out13"; "loa"], Or[F["in13"; "loa"];F["sw_flt_loa"]]);
                  (["out14"; "loa"], Or[F["in14"; "loa"];F["sw_flt_loa"]]);
                  (["out15"; "loa"], Or[F["in15"; "loa"];F["sw_flt_loa"]]);
                  (["out16m1"; "loa"], Or[F["in16"; "loa"];F["sw_flt_loa"]]);
                  (["out17m1"; "loa"], Or[F["in17"; "loa"];F["sw_flt_loa"]]);
                  (["out18m1"; "loa"], Or[F["in18"; "loa"];F["sw_flt_loa"]]);
                  (["out19m1"; "loa"], Or[F["in19"; "loa"];F["sw_flt_loa"]]);
                  (["out16m2"; "loa"], Or[F["in16"; "loa"];F["sw_flt_loa"]]);
                  (["out17m2"; "loa"], Or[F["in17"; "loa"];F["sw_flt_loa"]]);
                  (["out18m2"; "loa"], Or[F["in18"; "loa"];F["sw_flt_loa"]]);
                  (["out19m2"; "loa"], Or[F["in19"; "loa"];F["sw_flt_loa"]]);
                  ]
};
```

Figure 35. Switch 1 and 2 component model

The model for Switch 3 is similar to the previous component model, but it only includes two input flows. Both inputs are multicast to different outputs.

Figure 36 shows the component model used for Switch 3 (A and B).

```
{name          = "Switch3";
  faults       = ["ued"; "loa"];
  input_flows  = ["in1"; "in2"];
  basic_events = ["sw_flt_ued"; "sw_flt_loa"; "crc32_flt"];
  event_info   = [(1.0e-6, 1.0); (1.0e-5, 1.0); (2.**(-32.), 1.0)];
  output_flows = ["out1m1"; "out1m2"; "out2m1"; "out2m2"];
  formulas     = [(["out1m1"; "ued"],Or[F["in1"; "ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out1m2"; "ued"],Or[F["in1"; "ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out2m1"; "ued"],Or[F["in2"; "ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out2m2"; "ued"],Or[F["in2"; "ued"];And[F["sw_flt_ued"];F["crc32_flt"]]]);
                  (["out1m1"; "loa"],Or[F["in1"; "loa"];F["sw_flt_loa"]]);
                          (["out1m2"; "loa"],Or[F["in1"; "loa"];F["sw_flt_loa"]]);
                          (["out2m1"; "loa"],Or[F["in2"; "loa"];F["sw_flt_loa"]]);
                          (["out2m2"; "loa"],Or[F["in2"; "loa"];F["sw_flt_loa"]]);
                          ]
};
```

Figure 36. Switch 3 component model

In addition to the components of the system, we determined we needed to create component models to represent the different failure conditions we were modeling. Like other examples we have done, these dummy components allow for us to calculate system level effects that are the combination of multiple outputs. In this example, we needed to be able to model a condition where any 1 of 4 effectors failed to operate correctly. We also needed to model a condition where two effectors were required to fail to operate to cause the failure condition.

Figure 37 shows the dummy component models used for connecting multiple effector outputs to support top-level failure condition determination.

```
{name          = "And2"; (* A AND B - both need to fail *)
  faults       = ["ued"; "loa"];
  input_flows  = ["a";"b"];
  basic_events = [];
  event_info   = [];
  output_flows = ["z"];
  formulas     = [ (["z"; "ued"], And[F["a"; "ued"]; F["b"; "ued"]]);
                   (["z"; "loa"], And[F["a"; "loa"]; F["b"; "loa"]])
                  ]
};
{name          = "Or4"; (* A OR B OR C OR D - Any 1 needs to fail *)
  faults       = ["ued"; "loa"];
  input_flows  = ["a";"b";"c";"d"];
  basic_events = [];
  event_info   = [];
  output_flows = ["z"];
  formulas     = [ (["z"; "ued"], Or[F["a"; "ued"]; F["b"; "ued"];F["c"; "ued"];F["d"; "ued"]]);
                   (["z"; "loa"], Or[F["a"; "loa"]; F["b"; "loa"];F["c"; "loa"];F["d"; "loa"]])
                  ]  };
```

Figure 37. Top-level logic component models

### 9.4.2.2 System Model

The system modeling involved instantiating each component model.

Figure 38 shows the component instantiations used in the system model.

```
(*Sensors*)
    makeInstance ~i:"SpoilerSenL" ~c:"Sen" ~l:[("sen_flt_ued", 5.0e-7)] ();
    makeInstance ~i:"WOWSenL" ~c:"Sen" ~l:[("sen_flt_ued", 2.0e-7)] ();
    makeInstance ~i:"GearDoorSenL" ~c:"Sen" ~l:[("sen_flt_ued", 6.0e-7)] ();
    makeInstance ~i:"GearPosSenL" ~c:"Sen" ~l:[("sen_flt_ued", 6.0e-7)] ();
    makeInstance ~i:"SpoilerSenR" ~c:"Sen" ~l:[("sen_flt_ued", 5.0e-7)] ();
    makeInstance ~i:"WOWSenR" ~c:"Sen" ~l:[("sen_flt_ued", 2.0e-7)] ();
    makeInstance ~i:"GearDoorSenR" ~c:"Sen" ~l:[("sen_flt_ued", 6.0e-7)] ();
    makeInstance ~i:"GearPosSenR" ~c:"Sen" ~l:[("sen_flt_ued", 6.0e-7)] ();
    makeInstance ~i:"GearLeverSen" ~c:"Sen" ~l:[("sen_flt_ued", 1.0e-6)] ();
    makeInstance ~i:"AirspeedSen" ~c:"Sen" ~l:[("sen_flt_ued", 1.0e-6)] ();
(*Effectors*)
    makeInstance ~i:"SpoilerEffL" ~c:"Eff" ~l:[("eff_flt_ued", 5.0e-7)] ();
    makeInstance ~i:"GearDoorEffL" ~c:"Eff" ~l:[("eff_flt_ued", 6.0e-7)] ();
    makeInstance ~i:"GearPosEffL" ~c:"Eff" ~l:[("eff_flt_ued", 6.0e-7)] ();
    makeInstance ~i:"SpoilerEffR" ~c:"Eff" ~l:[("eff_flt_ued", 5.0e-7)] ();
    makeInstance ~i:"GearDoorEffR" ~c:"Eff" ~l:[("eff_flt_ued", 6.0e-7)] ();
    makeInstance ~i:"GearPosEffR" ~c:"Eff" ~l:[("eff_flt_ued", 6.0e-7)] ();
(*IMA Components*)
    makeInstance "riu1" "RIU_i4o3" ();
    makeInstance "riu2" "RIU_i4o3" ();
    makeInstance "riu3" "RIU_i2o0" ();
    makeInstance "sw1a" "Switch1" ();
    makeInstance "sw1b" "Switch1" ();
    makeInstance "sw2a" "Switch1" ();
    makeInstance "sw2b" "Switch1" ();
    makeInstance "sw3a" "Switch3" ();
    makeInstance "sw3b" "Switch3" ();
    makeInstance "gpm1" "GPM_Spoiler" ();
    makeInstance "gpm2" "GPM_Gear" ();
    makeInstance "gpm3" "GPM_Spoiler" ();
    makeInstance "gpm4" "GPM_Gear" ();
(*Define top-level failure conditions*)
(*FC1 - Inability to deploy gear (loss of either) *)
    makeInstance "FC1" "Or4" ();
(*FC2 - Loss of Spoilers (loss of both) *)
    makeInstance "FC2" "And2" ();
```

Figure 38. Component Instantiations

The following figures show the connections required to support the system model.

```
(* Sensor to RIU *)
    (("riu1", "rin1"), ("SpoilerSenL", "out"));
    (("riu1", "rin2"), ("WOWSenL", "out"));
    (("riu1", "rin3"), ("GearDoorSenL","out"));
    (("riu1", "rin4"), ("GearPosSenL","out"));
    (("riu2", "rin1"), ("SpoilerSenR", "out"));
    (("riu2", "rin2"), ("WOWSenR", "out"));
    (("riu2", "rin3"), ("GearDoorSenR","out"));
    (("riu2", "rin4"), ("GearPosSenR","out"));
    (("riu3", "rin1"), ("GearLeverSen","out"));
    (("riu3", "rin2"), ("AirspeedSen","out"));
(* RIU to Effector *)
    (("SpoilerEffL","in"), ("riu1","out1"));
    (("GearDoorEffL","in"), ("riu1","out2"));
    (("GearPosEffL","in"), ("riu1","out3"));
    (("SpoilerEffR","in"), ("riu2","out1"));
    (("GearDoorEffR","in"), ("riu2","out2"));
    (("GearPosEffR","in"), ("riu2","out3"));
```

Figure 39. Connections – RIU to Sensor/Effector

```
(* A664 - Switch-Switch Trunk Links *)
    (("sw1a", "in09"), ("sw2a", "out03"));
    (("sw1a", "in10"), ("sw2a", "out05"));
    (("sw1a", "in11"), ("sw2a", "out07"));
    (("sw1a", "in12"), ("sw2a", "out20"));
    (("sw1a", "in13"), ("sw2a", "out21"));
    (("sw1a", "in14"), ("sw2a", "out22"));
    (("sw1a", "in15"), ("sw2a", "out23"));

    (("sw1b", "in09"), ("sw2b", "out03"));
    (("sw1b", "in10"), ("sw2b", "out05"));
    (("sw1b", "in11"), ("sw2b", "out07"));
    (("sw1b", "in12"), ("sw2b", "out20"));
    (("sw1b", "in13"), ("sw2b", "out21"));
    (("sw1b", "in14"), ("sw2b", "out22"));
    (("sw1b", "in15"), ("sw2b", "out23"));

    (("sw2a", "in09"), ("sw1a", "out04"));
    (("sw2a", "in10"), ("sw1a", "out06"));
    (("sw2a", "in11"), ("sw1a", "out08"));
    (("sw2a", "in12"), ("sw1a", "out20"));
    (("sw2a", "in13"), ("sw1a", "out21"));
    (("sw2a", "in14"), ("sw1a", "out22"));
    (("sw2a", "in15"), ("sw1a", "out23"));

    (("sw2b", "in09"), ("sw1b", "out04"));
    (("sw2b", "in10"), ("sw1b", "out06"));
    (("sw2b", "in11"), ("sw1b", "out08"));
    (("sw2b", "in12"), ("sw1b", "out20"));
    (("sw2b", "in13"), ("sw1b", "out21"));
    (("sw2b", "in14"), ("sw1b", "out22"));
    (("sw2b", "in15"), ("sw1b", "out23"));

    (("sw1a", "in01"), ("sw3a", "out1"));
    (("sw1a", "in02"), ("sw3a", "out3"));
    (("sw1b", "in01"), ("sw3b", "out1"));
    (("sw1b", "in02"), ("sw3b", "out3"));
    (("sw2a", "in01"), ("sw3a", "out2"));
    (("sw2a", "in02"), ("sw3a", "out4"));
    (("sw2b", "in01"), ("sw3b", "out2"));
    (("sw2b", "in02"), ("sw3b", "out4"));
```

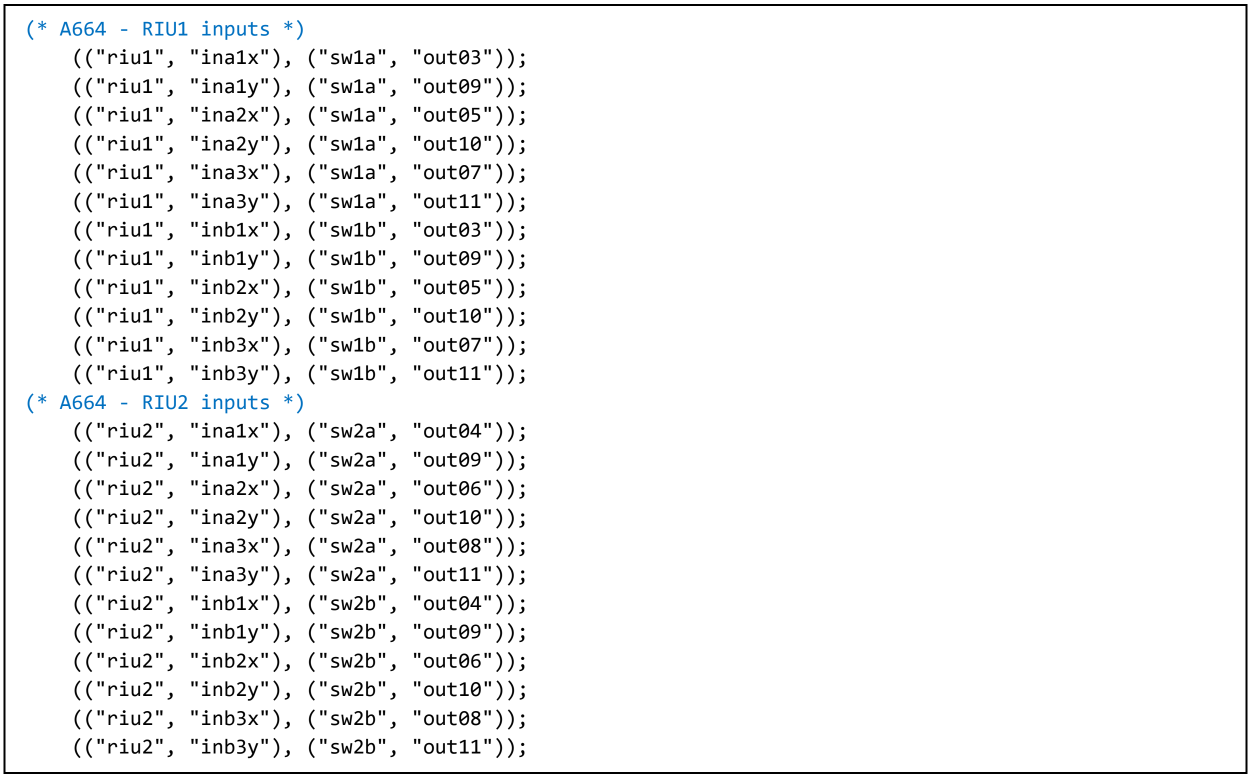
Figure 40. Connections – Switch-Switch Trunk Links

```
(* A664 - Switch 1A inputs *)
    (("sw1a", "in03"), ("gpm1", "out1a"));
    (("sw1a", "in04"), ("gpm1", "out2a"));
    (("sw1a", "in05"), ("gpm2", "out1a"));
    (("sw1a", "in06"), ("gpm2", "out2a"));
    (("sw1a", "in07"), ("gpm2", "out3a"));
    (("sw1a", "in08"), ("gpm2", "out4a"));
    (("sw1a", "in16"), ("riu1", "outa1"));
    (("sw1a", "in17"), ("riu1", "outa2"));
    (("sw1a", "in18"), ("riu1", "outa3"));
    (("sw1a", "in19"), ("riu1", "outa4"));
(* A664 - Switch 1B inputs *)
    (("sw1b", "in03"), ("gpm1", "out1b"));
    (("sw1b", "in04"), ("gpm1", "out2b"));
    (("sw1b", "in05"), ("gpm2", "out1b"));
    (("sw1b", "in06"), ("gpm2", "out2b"));
    (("sw1b", "in07"), ("gpm2", "out3b"));
    (("sw1b", "in08"), ("gpm2", "out4b"));
    (("sw1b", "in16"), ("riu1", "outb1"));
    (("sw1b", "in17"), ("riu1", "outb2"));
    (("sw1b", "in18"), ("riu1", "outb3"));
    (("sw1b", "in19"), ("riu1", "outb4"));
(* A664 - Switch 2A inputs *)
    (("sw2a", "in03"), ("gpm3", "out1a"));
    (("sw2a", "in04"), ("gpm3", "out2a"));
    (("sw2a", "in05"), ("gpm4", "out1a"));
    (("sw2a", "in06"), ("gpm4", "out2a"));
    (("sw2a", "in07"), ("gpm4", "out3a"));
    (("sw2a", "in08"), ("gpm4", "out4a"));
    (("sw2a", "in16"), ("riu2", "outa1"));
    (("sw2a", "in17"), ("riu2", "outa2"));
    (("sw2a", "in18"), ("riu2", "outa3"));
    (("sw2a", "in19"), ("riu2", "outa4"));
(* A664 - Switch 2B inputs *)
    (("sw2b", "in03"), ("gpm3", "out1b"));
    (("sw2b", "in04"), ("gpm3", "out2b"));
    (("sw2b", "in05"), ("gpm4", "out1b"));
    (("sw2b", "in06"), ("gpm4", "out2b"));
    (("sw2b", "in07"), ("gpm4", "out3b"));
    (("sw2b", "in08"), ("gpm4", "out4b"));
    (("sw2b", "in16"), ("riu2", "outb1"));
    (("sw2b", "in17"), ("riu2", "outb2"));
    (("sw2b", "in18"), ("riu2", "outb3"));
    (("sw2b", "in19"), ("riu2", "outb4"));
(* A664 - Switch 3A inputs *)
    (("sw3a", "in1"), ("riu3", "outa1"));
    (("sw3a", "in2"), ("riu3", "outa2"));
(* A664 - Switch 3B inputs *)
    (("sw3b", "in1"), ("riu3", "outb1"));
    (("sw3b", "in2"), ("riu3", "outb2"));
```

Figure 41. Connections – Switch-Input Links

```
(* A664 - GPM1 inputs *)
    (("gpm1", "gin1a"), ("sw1a", "out17"));
    (("gpm1", "gin2a"), ("sw1a", "out13"));
    (("gpm1", "gin3a"), ("sw1a", "out16"));
    (("gpm1", "gin4a"), ("sw1a", "out12"));
    (("gpm1", "gin5a"), ("sw1a", "out02"));
    (("gpm1", "gin1b"), ("sw1b", "out17"));
    (("gpm1", "gin2b"), ("sw1b", "out13"));
    (("gpm1", "gin3b"), ("sw1b", "out16"));
    (("gpm1", "gin4b"), ("sw1b", "out12"));
    (("gpm1", "gin5b"), ("sw1b", "out02"));
(* A664 - GPM2 inputs *)
    (("gpm2", "gin1a"), ("sw1a", "out01"));
    (("gpm2", "gin2a"), ("sw1a", "out18"));
    (("gpm2", "gin3a"), ("sw1a", "out14"));
    (("gpm2", "gin4a"), ("sw1a", "out19"));
    (("gpm2", "gin5a"), ("sw1a", "out15"));
    (("gpm2", "gin1b"), ("sw1b", "out01"));
    (("gpm2", "gin2b"), ("sw1b", "out18"));
    (("gpm2", "gin3b"), ("sw1b", "out14"));
    (("gpm2", "gin4b"), ("sw1b", "out19"));
    (("gpm2", "gin5b"), ("sw1b", "out15"));
(* A664 - GPM3 inputs *)
    (("gpm3", "gin1a"), ("sw2a", "out13"));
    (("gpm3", "gin2a"), ("sw2a", "out17"));
    (("gpm3", "gin3a"), ("sw2a", "out12"));
    (("gpm3", "gin4a"), ("sw2a", "out16"));
    (("gpm3", "gin5a"), ("sw2a", "out02"));
    (("gpm3", "gin1b"), ("sw2b", "out13"));
    (("gpm3", "gin2b"), ("sw2b", "out17"));
    (("gpm3", "gin3b"), ("sw2b", "out12"));
    (("gpm3", "gin4b"), ("sw2b", "out16"));
    (("gpm3", "gin5b"), ("sw2b", "out02"));
(* A664 - GPM4 inputs *)
    (("gpm4", "gin1a"), ("sw2a", "out01"));
    (("gpm4", "gin2a"), ("sw2a", "out14"));
    (("gpm4", "gin3a"), ("sw2a", "out18"));
    (("gpm4", "gin4a"), ("sw2a", "out15"));
    (("gpm4", "gin5a"), ("sw2a", "out19"));
    (("gpm4", "gin1b"), ("sw2b", "out01"));
    (("gpm4", "gin2b"), ("sw2b", "out14"));
    (("gpm4", "gin3b"), ("sw2b", "out18"));
    (("gpm4", "gin4b"), ("sw2b", "out15"));
    (("gpm4", "gin5b"), ("sw2b", "out19"));
```

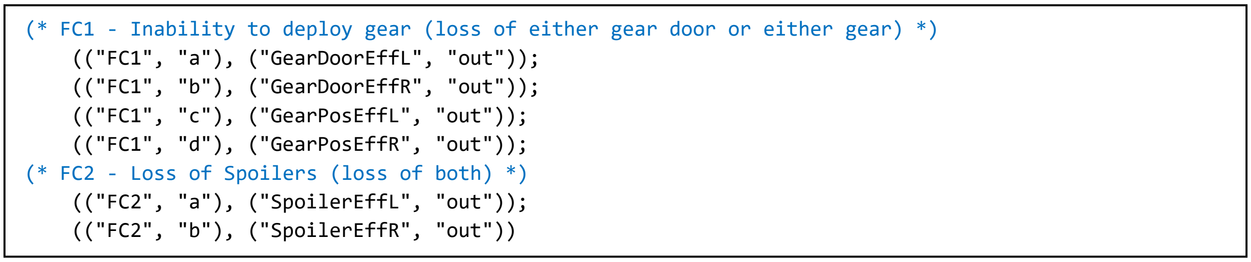
Figure 42. Connections – GPM-Input Links



```
(* A664 - RIU1 inputs *)
    (("riu1", "ina1x"), ("sw1a", "out03"));
    (("riu1", "ina1y"), ("sw1a", "out09"));
    (("riu1", "ina2x"), ("sw1a", "out05"));
    (("riu1", "ina2y"), ("sw1a", "out10"));
    (("riu1", "ina3x"), ("sw1a", "out07"));
    (("riu1", "ina3y"), ("sw1a", "out11"));
    (("riu1", "inb1x"), ("sw1b", "out03"));
    (("riu1", "inb1y"), ("sw1b", "out09"));
    (("riu1", "inb2x"), ("sw1b", "out05"));
    (("riu1", "inb2y"), ("sw1b", "out10"));
    (("riu1", "inb3x"), ("sw1b", "out07"));
    (("riu1", "inb3y"), ("sw1b", "out11"));
(* A664 - RIU2 inputs *)
    (("riu2", "ina1x"), ("sw2a", "out04"));
    (("riu2", "ina1y"), ("sw2a", "out09"));
    (("riu2", "ina2x"), ("sw2a", "out06"));
    (("riu2", "ina2y"), ("sw2a", "out10"));
    (("riu2", "ina3x"), ("sw2a", "out08"));
    (("riu2", "ina3y"), ("sw2a", "out11"));
    (("riu2", "inb1x"), ("sw2b", "out04"));
    (("riu2", "inb1y"), ("sw2b", "out09"));
    (("riu2", "inb2x"), ("sw2b", "out06"));
    (("riu2", "inb2y"), ("sw2b", "out10"));
    (("riu2", "inb3x"), ("sw2b", "out08"));
    (("riu2", "inb3y"), ("sw2b", "out11"));
```

Figure 43. Connections – RIU-A664 Input Links

The final connections are modeled to define the top-level failure conditions. Once these are in the model, we just needed to map the output “z” of the appropriate failure condition component to the top fault to get the results.  This shows how easy it is to model multiple failure conditions using the same exact system model.



```
(* FC1 - Inability to deploy gear (loss of either gear door or either gear) *)
    (("FC1", "a"), ("GearDoorEffL", "out"));
    (("FC1", "b"), ("GearDoorEffR", "out"));
    (("FC1", "c"), ("GearPosEffL", "out"));
    (("FC1", "d"), ("GearPosEffR", "out"));
(* FC2 - Loss of Spoilers (loss of both) *)
    (("FC2", "a"), ("SpoilerEffL", "out"));
    (("FC2", "b"), ("SpoilerEffR", "out"))
```

Figure 44. Connections – Define top-level failure conditions

### 9.4.3 Model Results and Comparison

We ran the model in our tool to generate the top-level probability result for each failure condition and the cut sets that contribute to the top fault.  The tool also generates some interesting graphic representations that we included here. In parallel, we built a fault tree for each failure condition using more traditional safety methods so we had some results to compare against what the tool generated.

The details are provided in the following sections, but in summary the results out of our modeling tool were very good relative to the human generated fault trees. In the case of failure condition 2 we discovered an error in the human generated fault tree (one of the sensor inputs was not included) by performing the comparison of the results. In both cases, though the top-level results matched, the model generate more cut sets than the hand fault trees. This highlights the fact that the safety engineer is more likely to make assumptions during the fault tree development process to simplify the tree where they know there is negligible impact to the top-level result. Since the model is looking at the actual data flows in the architecture, the resulting cut sets are more complete because the model doesn't make these assumptions.

Figure 45 is a visualization generated by the tool of the physical model. Figure 46 is a visualization generated by the tool of the individual connections within the model. This diagram isn't very practical to read in detail, but it does give a visual appreciation for the complexity of this model.

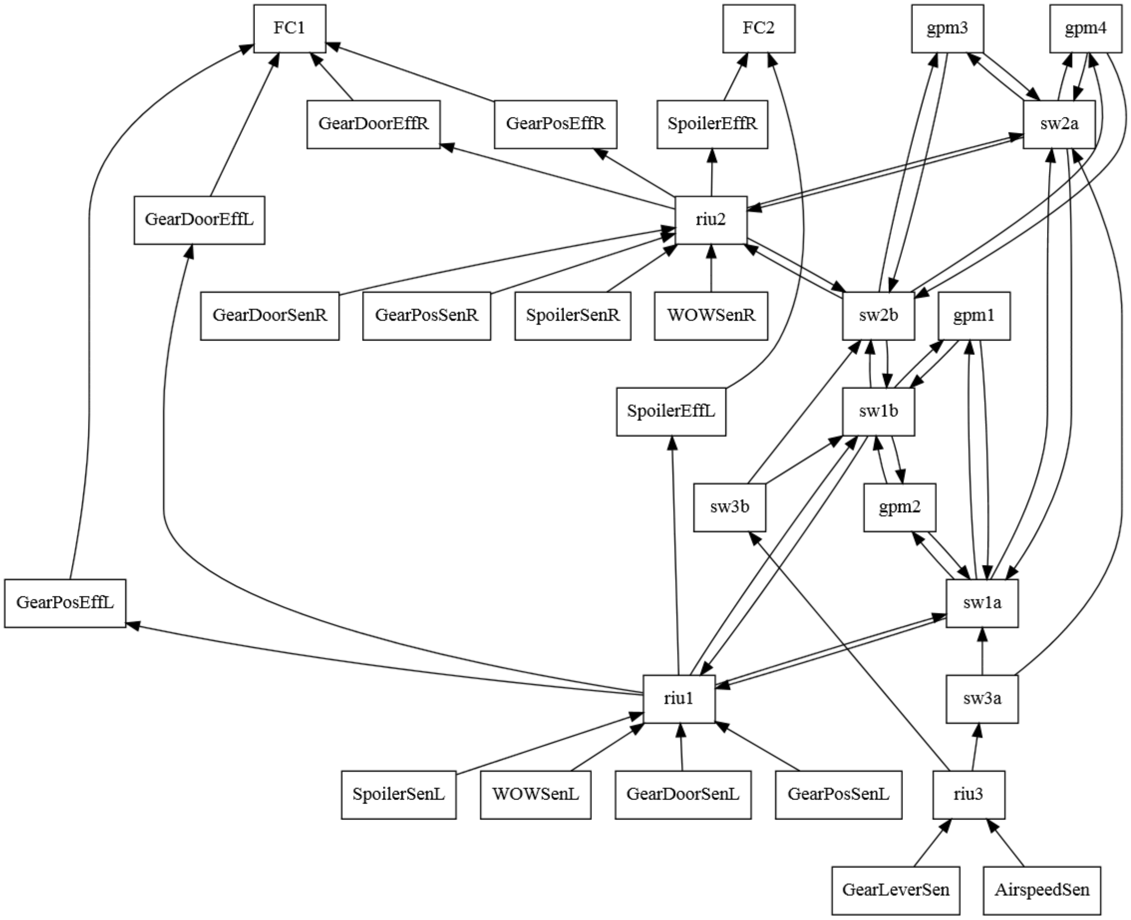


Figure 45. Visualization - physical model

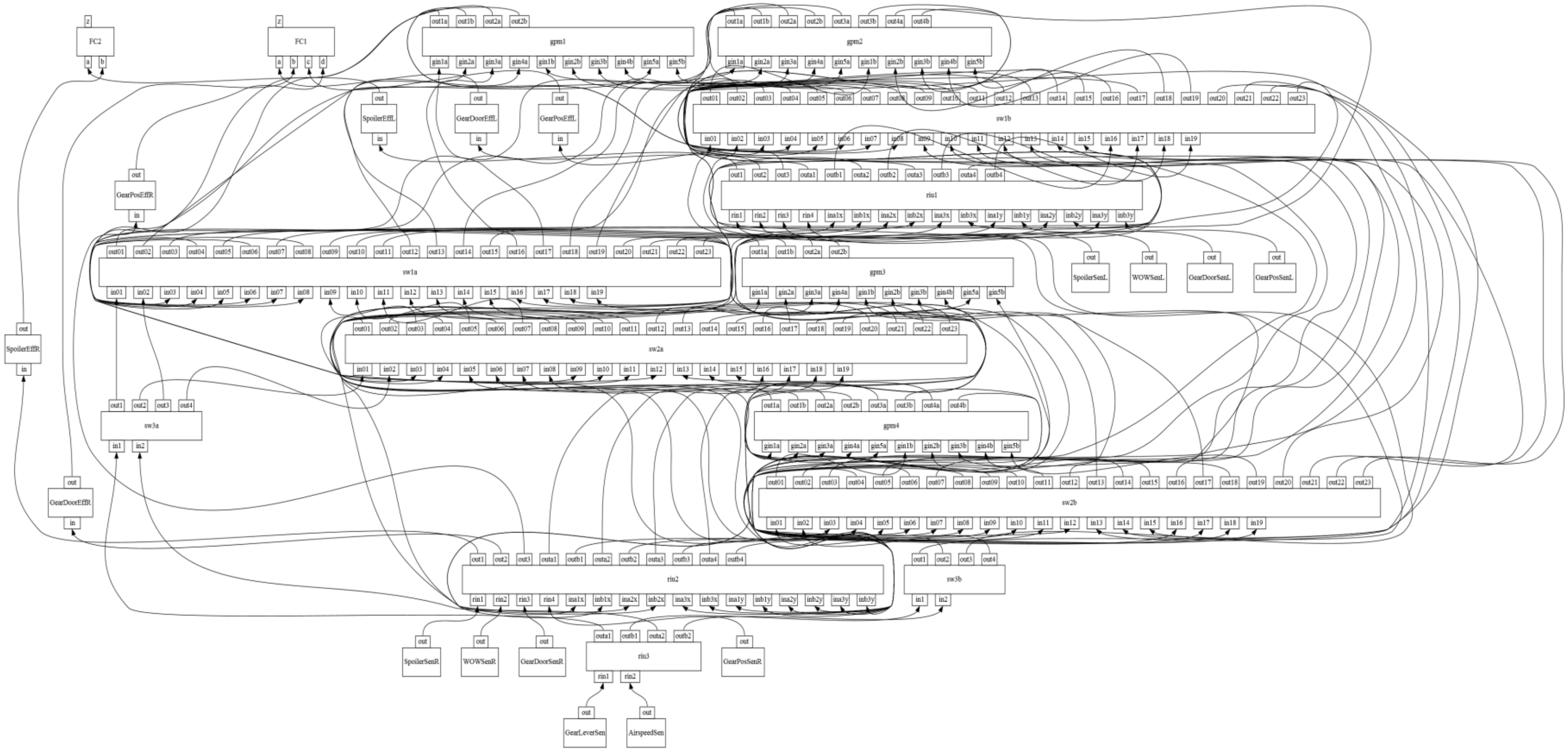


Figure 46. Visualization – connection model

#### 9.4.3.1 Failure Condition 1 – Loss of Landing Gear Deployment

The hand generated fault tree generated the following cut sets:

| ID | Event 1 | Event 2 |
|---|---|---|
| 1 | Left Door Actuator | |
| 2 | Right Door Actuator | |
| 3 | Left Door Sensor | |
| 4 | Right Door Sensor | |
| 5 | L. Gear Switch Push-button | |
| 6 | Left L. Gear Actuator | |
| 7 | Right L. Gear Actuator | |
| 8 | Left L. Gear Sensor | |
| 9 | Right L. Gear Sensor | |
| 10 | RIU1 | |
| 11 | RIU2 | |
| 12 | RIU3 | |
| 13 | GPM2 | GPM4 |
| 14 | Switch1A | Switch1B |
| 15 | Switch2A | Switch2B |
| 16 | Switch3A | Switch3B |

The tool generated the following cut sets based on the model:

| ID | Event 1 | Event 2 | Event 3 |
|---|---|---|---|
| 1 | ("GearDoorEffL","eff_flt_loa"), | | |
| 2 | ("GearDoorEffR","eff_flt_loa"), | | |
| 3 | ("GearDoorSenL","sen_flt_loa"), | | |
| 4 | ("GearDoorSenR","sen_flt_loa"), | | |
| 5 | ("GearLeverSen","sen_flt_loa"), | | |
| 6 | ("GearPosEffL","eff_flt_loa"), | | |
| 7 | ("GearPosEffR","eff_flt_loa"), | | |
| 8 | ("GearPosSenL","sen_flt_loa"), | | |
| 9 | ("GearPosSenR","sen_flt_loa"), | | |
| 10 | ("riu1","riu_flt_loa"), | | |
| 11 | ("riu2","riu_flt_loa"), | | |
| 12 | ("riu3","riu_flt_loa"), | | |
| 13 | ("gpm2","gpm_flt_loa"); | ("gpm4","gpm_flt_loa")], | |
| 14 | ("sw1a","sw_flt_loa"); | ("sw1b","sw_flt_loa")], | |
| 15 | ("sw2a","sw_flt_loa"); | ("sw2b","sw_flt_loa")], | |
| 16 | ("sw3a","sw_flt_loa"); | ("sw3b","sw_flt_loa")], | |

|  |  |  |  |
|---|---|---|---|
| 17 | ("gpm2","gpm_flt_loa"); | ("sw1a","sw_flt_loa"); | ("sw2b","sw_flt_loa")], |
| 18 | ("gpm2","gpm_flt_loa"); | ("sw1b","sw_flt_loa"); | ("sw2a","sw_flt_loa")], |
| 19 | ("gpm4","gpm_flt_loa"); | ("sw1a","sw_flt_loa"); | ("sw2b","sw_flt_loa")], |
| 20 | ("gpm4","gpm_flt_loa"); | ("sw1b","sw_flt_loa"); | ("sw2a","sw_flt_loa")], |
| 21 | ("gpm2","gpm_flt_loa"); | ("sw2a","sw_flt_loa"); | ("sw3b","sw_flt_loa")], |
| 22 | ("gpm2","gpm_flt_loa"); | ("sw2b","sw_flt_loa"); | ("sw3a","sw_flt_loa")], |
| 23 | ("gpm4","gpm_flt_loa"); | ("sw1a","sw_flt_loa"); | ("sw3b","sw_flt_loa")], |
| 24 | ("gpm4","gpm_flt_loa"); | ("sw1b","sw_flt_loa"); | ("sw3a","sw_flt_loa")], |
| 25 | ("sw1a","sw_flt_loa"); | ("sw2a","sw_flt_loa"); | ("sw3b","sw_flt_loa")], |
| 26 | ("sw1a","sw_flt_loa"); | ("sw2b","sw_flt_loa"); | ("sw3a","sw_flt_loa")], |
| 27 | ("sw1a","sw_flt_loa"); | ("sw2b","sw_flt_loa"); | ("sw3b","sw_flt_loa")], |
| 28 | ("sw1b","sw_flt_loa"); | ("sw2a","sw_flt_loa"); | ("sw3a","sw_flt_loa")], |
| 29 | ("sw1b","sw_flt_loa"); | ("sw2a","sw_flt_loa"); | ("sw3b","sw_flt_loa")], |
| 30 | ("sw1b","sw_flt_loa"); | ("sw2b","sw_flt_loa"); | ("sw3a","sw_flt_loa")], |

Cut sets 1-16 are the same between the model and the hand generated fault tree. The additional cut sets 17-30 identified by the model show that the model is more accurate than the hand generated fault tree. Both analyses resulted in the same top-probability of 1.2E-4 per flight hour (pfh), so these additional cut sets don't impact the result within the accuracy that safety is concerned.

### *9.4.3.2 Failure Condition 2 – Loss of Spoiler Deployment*

The hand generated fault tree generated the following cut sets:

| **ID** | **Event 1** | **Event 2** | **Event 3** | **Event 4** |
|---|---|---|---|---|
| 1 | Airspeed Sensor |  |  |  |
| 2 | RIU3 |  |  |  |
| 3 | GPM1 | GPM3 |  |  |
| 4 | RIU1 | Right Spoiler Actuator |  |  |
| 5 | Left Spoiler Actuator | RIU2 |  |  |
| 6 | Left Spoiler Actuator | Right Spoiler Actuator |  |  |
| 7 | Left Spoiler Actuator | Right Spoiler Sensor |  |  |
| 8 | Left Spoiler Sensor | RIU2 |  |  |
| 9 | Left Spoiler Sensor | Right Spoiler Actuator |  |  |
| 10 | RIU1 | RIU2 |  |  |
| 11 | RIU1 | Right Spoiler Sensor |  |  |
| 12 | RIU1 | Right WOW Sensor |  |  |
| 13 | RIU2 | Left WOW Sensor |  |  |
| 14 | Switch3A | Switch3B |  |  |
| 15 | Left WOW Sensor | Right WOW Sensor |  |  |
| 16 | Left Spoiler Sensor | Right Spoiler Sensor |  |  |
| 17 | Switch1A | GPM3 | Switch1B |  |

| | | | | |
|---|---|---|---|---|
| 18 | GPM1 | Switch2A | Switch2B | |
| 19 | RIU2 | Switch1A | Switch1B | |
| 20 | RIU1 | Switch2A | Switch2B | |
| 21 | Left Spoiler Sensor | Switch2A | Switch2B | |
| 22 | Left Spoiler Actuator | Switch2A | Switch2B | |
| 23 | Right Spoiler Actuator | Switch1A | Switch1B | |
| 24 | Right Spoiler Sensor | Switch1A | Switch1B | |
| 25 | Switch1A | Switch2A | Switch1B | Switch2B |

The tool generated a total of 73 cut sets which include 48 cut sets not identified by the hand fault tree. Of these 48, 8 are 3rd order cut sets and 40 are 4th order cut sets. Since the driving cut sets are first order cut sets, these additional cut sets have no mathematical impact on the top gate result.

Both analyses resulted in the same top-probability of 2.0E-5 pfh, so these additional cut sets don't impact the result within the accuracy that safety is concerned.

The following tables includes the additional 3rd order cut sets based on the model:

| ID | Event 1 | Event 2 | Event 3 |
|---|---|---|---|
| 1 | ("WOWSenL", "sen_flt_loa") | ("sw2a", "sw_flt_loa") | ("sw2b", "sw_flt_loa")] |
| 2 | ("WOWSenR", "sen_flt_loa") | ("sw1a", "sw_flt_loa") | ("sw1b", "sw_flt_loa")] |
| 3 | ("gpm1", "gpm_flt_loa") | ("sw2a", "sw_flt_loa") | ("sw3b", "sw_flt_loa")] |
| 4 | ("gpm1", "gpm_flt_loa") | ("sw2b", "sw_flt_loa") | ("sw3a", "sw_flt_loa")] |
| 5 | ("gpm3", "gpm_flt_loa") | ("sw1a", "sw_flt_loa") | ("sw3b", "sw_flt_loa")] |
| 6 | ("gpm3", "gpm_flt_loa") | ("sw1b", "sw_flt_loa") | ("sw3a", "sw_flt_loa")] |
| 7 | ("sw1a", "sw_flt_loa") | ("sw2a", "sw_flt_loa") | ("sw3b", "sw_flt_loa")] |
| 8 | ("sw1b", "sw_flt_loa") | ("sw2b", "sw_flt_loa") | ("sw3a", "sw_flt_loa")] |

The additional 4th order cut sets are omitted from the report.

### 9.4.4 Wheel Brake Model and the Combination of Both Models

The following failure conditions related to the braking function and the combined model were identified and analyzed:

- FC3 – Total Loss of Wheel Braking
- FC4 – Symmetric Partial Loss of Braking
- FC5 – Asymmetric Partial Loss of Braking
- FC6 – Partial Loss of Wheel Braking in combination with Loss of Spoilers

The component models needed include generic types, such as sensors and effectors, and specific types such as a BSCU, a selector valve, a hydraulic source, and effectors with multiple inputs to use for the brake actuator, meter and shutoff valve.

The instances and connections were modeled similar to other examples, refer to Section 9.4.2.2. The connections in the federated portion of this system were simpler to model than the connections in the IMA portion. Connection lines on the diagram (Figure 29) and the model connections map 1:1.

The dummy logic gates needed to define the top faults are the following:

- FC3 – Loss of all 4 brakes (A * B * C * D)
- FC4 – Loss 2 brakes, but only one on each side ((A + B) * (C + D))
- FC5 – Loss of any single brake (A + B + C + D)
- FC6 – FC2 in combination with FC4. (FC2 + FC4)

where A, B, C, and D represent each of the 4 brake actuators.

```
(* FC6 - Partial Loss of Braking in combination with Loss of Spoilers Mapping*)
(("FC6", "a"), ("FC4", "z"));
(("FC6", "b"), ("FC2", "z"))
```

#### 9.4.4.1 *Results*

Failure condition 3 initially created a stack overflow performance issue with our SOTERIA tool, which we were able to resolve. This was likely caused by the logic equation getting very large with the 4-input AND gate at the top. The probability obtained for failure condition 3 is 1.2E-4phf. There were 30 cut sets identified (1st order – 12, 2nd order – 4, 3rd order 14). The following are the top cutsets:

| ID | Event 1 | Event 2 | Probability |
|---|---|---|---|
| 1 | GearDoorEffL--eff_flt_loa | | 1.00E-05 |
| 2 | GearDoorEffR--eff_flt_loa | | 1.00E-05 |
| 3 | GearDoorSenL--sen_flt_loa | | 1.00E-05 |
| 4 | GearDoorSenR--sen_flt_loa | | 1.00E-05 |
| 5 | GearLeverSen--sen_flt_loa | | 1.00E-05 |
| 6 | GearPosEffL--eff_flt_loa | | 1.00E-05 |
| 7 | GearPosEffR--eff_flt_loa | | 1.00E-05 |
| 8 | GearPosSenL--sen_flt_loa | | 1.00E-05 |
| 9 | GearPosSenR--sen_flt_loa | | 1.00E-05 |
| 10 | riu1--riu_flt_loa | | 1.00E-05 |
| 11 | riu2--riu_flt_loa | | 1.00E-05 |
| 12 | riu3--riu_flt_loa | | 1.00E-05 |

The probability obtained for failure condition 4 is 1.4E-9 pfh. There were 485 cut sets identified (1st order – 1; 2nd order – 4; 3rd order 224; higher order – 256). The following are the top cut sets:

| ID | Event 1 | Event 2 | Probability |
|---|---|---|---|
| 1 | SelectorValve--flt_loa | | 1.00E-09 |
| 2 | LInWBAct--eff_flt_loa | RInWBAct--eff_flt_loa | 1.00E-10 |

|  |  |  |  |
|---|---|---|---|
| 3 | LInWBAct--eff_flt_loa | ROutWBAct--eff_flt_loa | 1.00E-10 |
| 4 | LOutWBAct--eff_flt_loa | RInWBAct--eff_flt_loa | 1.00E-10 |
| 5 | LOutWBAct--eff_flt_loa | ROutWBAct--eff_flt_loa | 1.00E-10 |

The probability obtained for failure condition 5 is 4.0E-5 pfh. There were 107 cut sets identified (1st order – 5; 2nd order – 72; 3rd order 28; higher order – 2) The following are the top cut sets:

| ID | Event 1 | Event 2 | Probability |
|---|---|---|---|
| 1 | LInWBAct--eff_flt_loa |  | 1.00E-05 |
| 2 | LOutWBAct--eff_flt_loa |  | 1.00E-05 |
| 3 | RInWBAct--eff_flt_loa |  | 1.00E-05 |
| 4 | ROutWBAct--eff_flt_loa |  | 1.00E-05 |
| 5 | SelectorValve--flt_loa |  | 1.00E-09 |

It can be noted that the driving cut sets are first order cut sets, second order cut sets can have some contribution but in these examples, additional cut sets after the first 5 have no mathematical impact on the top gate result.

The probability obtained for failure condition 6 is 2.0E-5 pfh. There were 558 cut sets identified (1st order – 3; 2nd order – 18; 3rd order 259; higher order – 289). The contribution from FMC2 was 2.05e-5 and the contribution from FMC4 was 1.4e-9. We noted that the activity to combine two existing models was rather quick, it took about 30 minutes. This shows the flexibility of the tool and effectiveness at allowing combination of existing models.

All the results were validated by cut set review.

### 9.4.5 Discussion

The modeling of the Wheel Brake and Landing Gear System led to the question of how we account for applications with different computing logic. In this example, the applications were physically located on different GPMs, but we identified some new considerations that would allow modeling different applications on the same GPM. We determined there are three options for modeling Applications:

- ○ Option 1 (currently used for this example) – Develop a unique GPM model for each application.
    - Disadvantage is that we can’t easily move applications from GPM to GPM
- ○ Option 2 – Develop a generic GPM model to capture GPM faults and have downstream models for the applications that model the logic.
    - This is more complex to model and still needs to handle the I/O routing to the GPM to move with the application
- ○ Option 3 - Develop a method of recognizing repeated events across different instantiations within the tool.
    - Each application could model the GPM failures, but the tool would be able to recognize repeated GPM failures

- Effectively each Application is modeled as a separate GPM even though they could be on the same physical hardware.
- This method requires changes to the tool.

While building the GPM fault models, we found that we needed to be specific about the states of different data items to ensure that they fit within the LOA and UED flows that are currently allowed in the tool. One example is the states for a discrete signal. If a discrete can be High or Low, we shouldn't need to worry about how those states feed LOA or UED at the system level. Rather, we should only need to consider the state based on the logic in one of the elements included in the model. Future tool changes to allow mixing UED and LOA could also be used to allow new states to allow certain data items. Another example of mixing UED and LOA is discussed in section 9.5.

When developing the network connections for the model, we noticed that this was a very pain-staking task, but it seems like a good candidate to provide a method to generate these connection definitions based on existing system development tools. This is outside the scope of the current SOTERIA project, but it does highlight an area for future work that would help streamline the development of models.

When running the model for failure condition 2, we identified a performance issue with the cut set generation algorithm. The logic for this specific failure condition led us to make changes to the tool to more efficiently determine the cut sets.

## 9.5 Models with Fault Dependencies

We introduce another interesting example where the top-level availability (LOA) is dependent on both LOA and UED (integrity) basic events. We kept the example simple so we can focus on the capability of the tool to use intermediate variables. We considered the impact of sensor UED on loss of ability to vote.

Here is a description of the data flows: Two sensors are fed into separate remote interface units (RIUs). Data from each sensor is routed to the network channel A and B via the RIU. Once the data has traversed the network to the GPM, redundancy management is used to pass a single copy of each sensor's data to the voting logic in the GPM. In this example, the GPM compares (by voting) the 2 sensor values and checks their reading for equality. Loss of integrity (or undetected erroneous data) occurs when the data is equal, but erroneous. In other words, both sensors' values are corrupted in a way that the comparison does not detect the erroneous data. Loss of availability occurs when both copies of at least one sensor propagate LOA and, since the values are voted, when either one of the sensor values is erroneous.

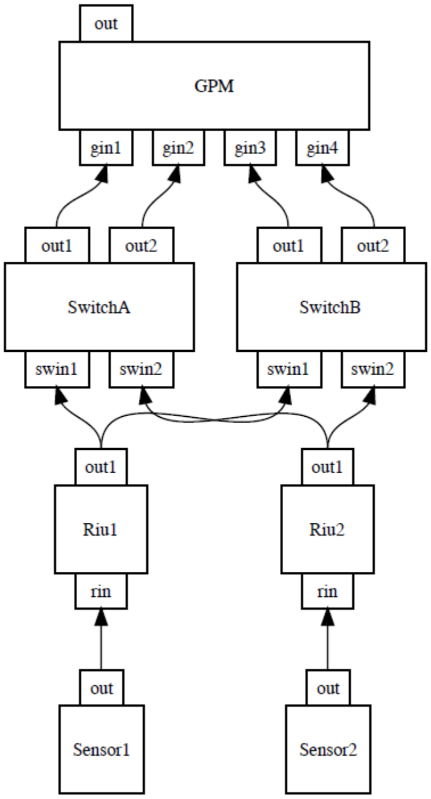
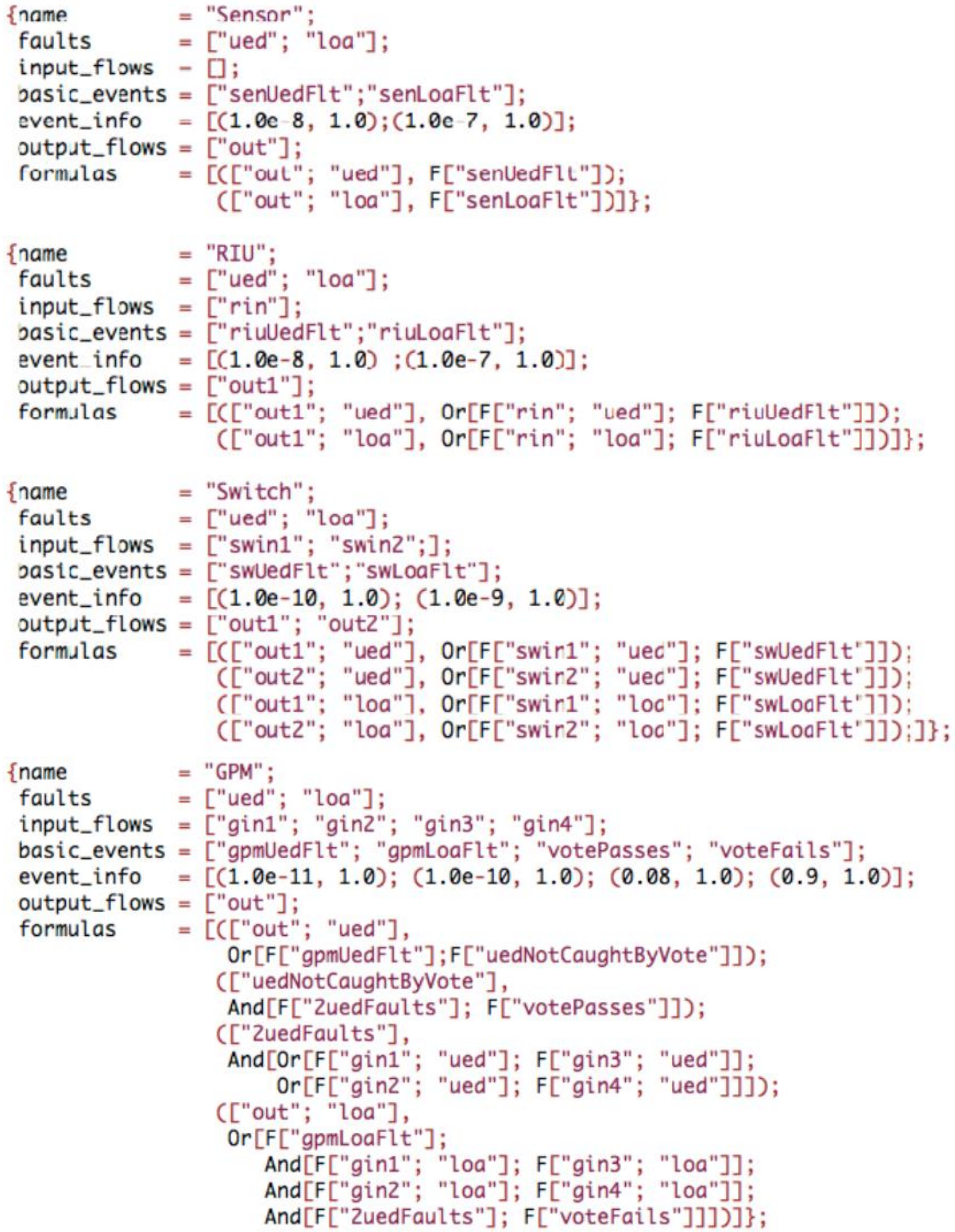


Figure 47 Functional architecture

Figure 47 shows the architecture. The GPM has 4 inputs:

- gin1 is from Sensor1 → RIU1 → SwitchA → GPM
- gin2 is from Sensor2 → RIU2 → SwitchA → GPM
- gin3 is from Sensor1 → RIU1 → SwitchB → GPM
- gin4 is from Sensor1 → RIU2 → SwitchB → GPM

In the GPM, the propagation logic of LOA refers to UED logic. Here is the component library.



```
{name          = "Sensor";
 faults        = ["ued"; "loa"];
 input_flows   = [];
 basic_events = ["senUedFlt";"senLoaFlt"];
 event_info    = [(1.0e-8, 1.0);(1.0e-7, 1.0)];
 output_flows = ["out"];
 formulas      = [(["out"; "ued"], F["senUedFlt"]);
                  (["out"; "loa"], F["senLoaFlt"])]};

{name          = "RIU";
 faults        = ["ued"; "loa"];
 input_flows   = ["rin"];
 basic_events = ["riuUedFlt";"riuLoaFlt"];
 event_info    = [(1.0e-8, 1.0) ;(1.0e-7, 1.0)];
 output_flows = ["out1"];
 formulas      = [(["out1"; "ued"], Or[F["rin"; "ued"]; F["riuUedFlt"]]);
                  (["out1"; "loa"], Or[F["rin"; "loa"]; F["riuLoaFlt"]])]};

{name          = "Switch";
 faults        = ["ued"; "loa"];
 input_flows   = ["swin1"; "swin2";];
 basic_events = ["swUedFlt";"swLoaFlt"];
 event_info    = [(1.0e-10, 1.0); (1.0e-9, 1.0)];
 output_flows = ["out1"; "out2"];
 formulas      = [(["out1"; "ued"], Or[F["swin1"; "ued"]; F["swUedFlt"]]);
                  (["out2"; "ued"], Or[F["swin2"; "ued"]; F["swUedFlt"]]);
                  (["out1"; "loa"], Or[F["swin1"; "loa"]; F["swLoaFlt"]]);
                  (["out2"; "loa"], Or[F["swin2"; "loa"]; F["swLoaFlt"]]);]};

{name          = "GPM";
 faults        = ["ued"; "loa"];
 input_flows   = ["gin1"; "gin2"; "gin3"; "gin4"];
 basic_events = ["gpmUedFlt"; "gpmLoaFlt"; "votePasses"; "voteFails"];
 event_info    = [(1.0e-11, 1.0); (1.0e-10, 1.0); (0.08, 1.0); (0.9, 1.0)];
 output_flows = ["out"];
 formulas      = [(["out"; "ued"],
                   Or[F["gpmUedFlt"];F["uedNotCaughtByVote"]]);
                  (["uedNotCaughtByVote"],
                   And[F["2uedFaults"]; F["votePasses"]]);
                  (["2uedFaults"],
                   And[Or[F["gin1"; "ued"]; F["gin3"; "ued"]];
                       Or[F["gin2"; "ued"]; F["gin4"; "ued"]]]);
                  (["out"; "loa"],
                   Or[F["gpmLoaFlt"];
                      And[F["gin1"; "loa"]; F["gin3"; "loa"]];
                      And[F["gin2"; "loa"]; F["gin4"; "loa"]];
                      And[F["2uedFaults"]; F["voteFails"]]]])]};
```

The redundancy management selects a Sensor1 reading from either gin1 or gin3 and a Sensor2 reading from either gin2 or gin4. The voting logic checks the equality of the readings. UED is then when the GPM has a UED fault, or when data is equal but erroneous. LOA is then when the GPM has a LOA fault, or when both copies of at least one sensor propagate LOA or there is a UED fault.

# 10 Architecture Synthesis

The goal of the SOTERIA program is to advance the techniques and tools available for safety analysis. In addition to automating the generation of fault analysis from models, another powerful concept is to automate the generation of an architecture knowing what the safety requirements are. In this way, a computer is used to synthesize safety-informed architectures. This also provides the ability to do rapid design space exploration for trade-off analysis, especially important when there are competing safety objectives like integrity (UED) and availability (LOA).

## 10.1 Challenge Architectures

To ground our work, we developed two challenge problems that capture real word issues when trying to synthesize an architecture to satisfy different safety requirements. These examples are designed to exercise the tool in the following ways:

1. Support sufficient user input to define the function being performed (e.g. without any additional redundancy considerations, how are the inputs used to perform the specified function?)
2. Apply redundancy at multiple stages in the data flow
3. Where redundancy is applied, capture the functional operation of the redundancy management in a way that will support the development and provide the correct failure model
4. Support different types of data flows based on interface types
5. Support dataflows that revisit the same physical equipment
6. Handle aspects of the system that cannot be changed during synthesis

If the synthesis tool works for these simple examples and supports the capabilities listed above, the same tool should be able to support much more complex architecture situations.

The first architecture, shown in Figure 48, is an IMA based Navigation System architecture where two different sensors, both with different interface types, are used as inputs to determining the position of the aircraft. The base architecture is defined such that both sensor input types are required to perform the function (e.g. Loss of function if either sensor type is missing, Undetected Erroneous function if either sensor type is erroneous). In this architecture, redundancy can be added for any block, but the GPM fault formulas (along with the functional operation of the software on the GPM) need to be modified to handle the redundancy management. For example, if a second Air Data Computer is added, the GPM logic needs to either vote the redundant sources or source select between them depending on which safety measure is trying to be improved with the redundancy (integrity or availability).

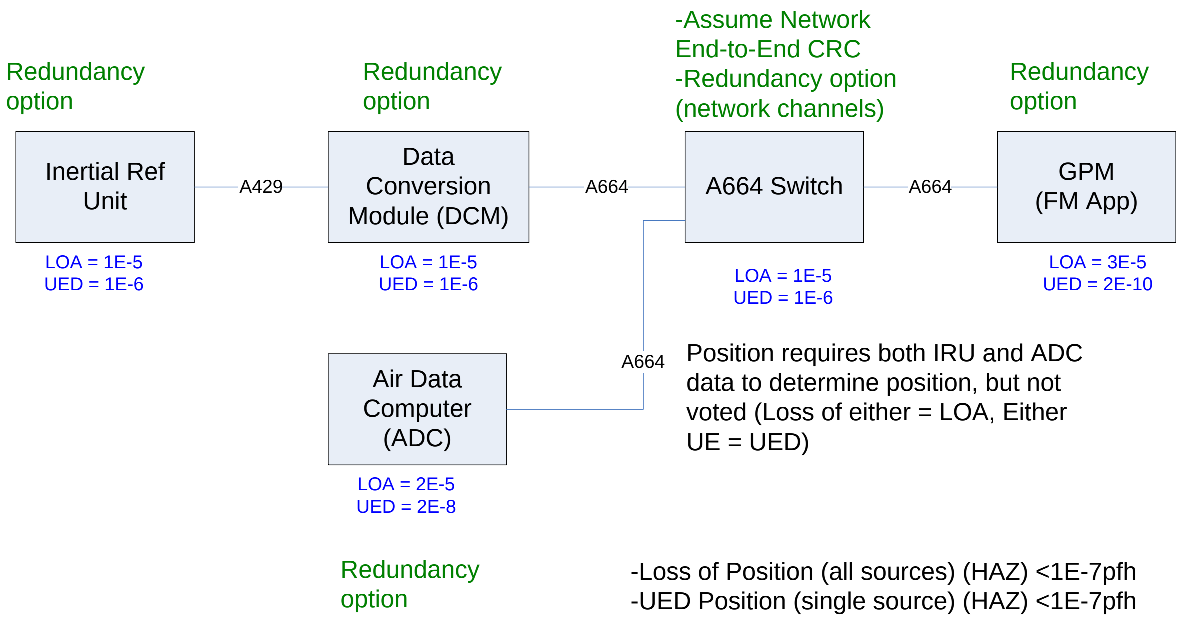


Figure 48. Challenge Problem #1 – Navigation Position

The second architecture, shown in Figure 49, is an IMA based Display System architecture where data flows from a sensor (Air Data Computer), into a GPM for processing, and then display commands are sent to the Displays. In this example, the A664 network switch is used twice in the data flow. Redundancy can be added for everything except the Displays. In this example, the addition of redundancy could affect the fault equations for the GPMs (handling redundant sensors) as well as the Displays (handling redundant Display commands).

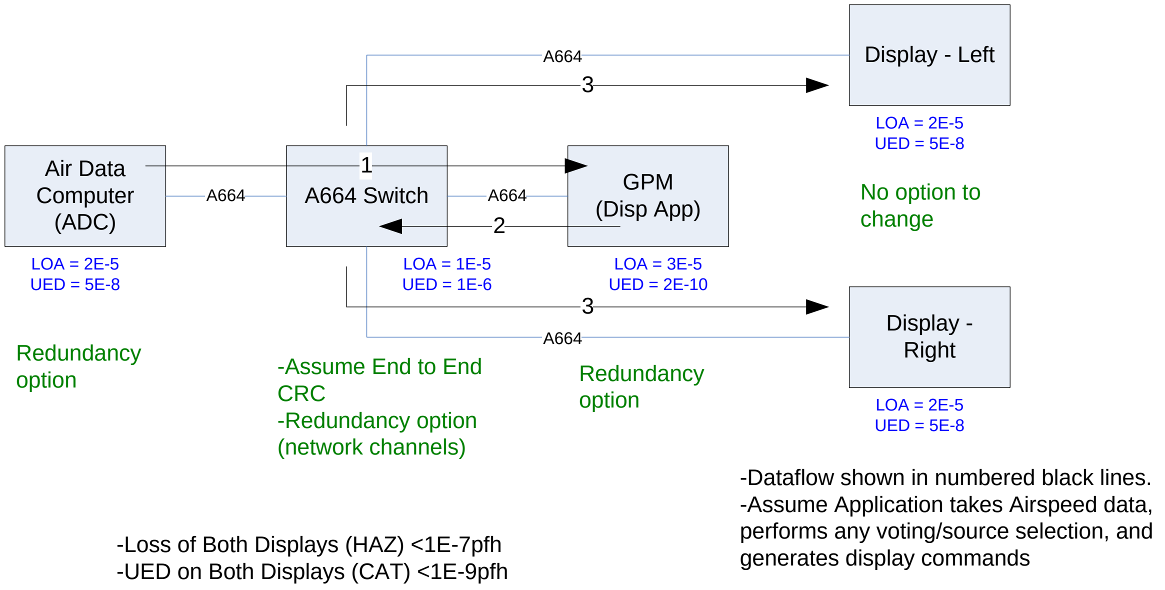


Figure 49. Challenge Problem #2 – Display of Airspeed

## 10.2 Inputs from User and Roles

To perform architecture synthesis, we defined several inputs to SOTERIA based on observations from the challenge problems. The challenge problems contain a wealth of information that are expressed explicitly and implicitly. We determined that the following inputs were required:

- Library of parametrized components:

  Each challenge problem is made up of a list of components. More accurately, they can be thought of as *classes* of components. Therefore, we expand our idea of a component library into a library of parameterized components. For each component class, we developed a function with a well-defined, well-documented collection of parameters that can be used to create a component in the component class. Practically, what this means is that for each parameterized component, there will be a function that, given the appropriate set of parameter values, generates a component. We call the parameterized components Big-C functions, and the generated component little-c.

- Component to component connection compatibility definition:

  To build an architecture, the tool needs to understand what components can and cannot be connected together. If two classes of components, A and B, are compatible, then they can be connected to one another. We model the constraints under which instances of A and B can be connected. This includes interface type compatibility (e.g. ARINC 664 vs ARINC 429). We also observed that there could be connection compatibility constraints that are either project agnostic, tied to the component library, or project specific that are independent from the library.

- Definition of available, “must use”, and replicate-able components

  The user must be able to specify a white list of available components to use in the architecture. As part of this definition it may require that a subset of a class of component be available based on the parameters discussed above. For example, the user may specify that an architecture is allowed to use component A with parameter x set to less than 3. There may also be specific components that must be used for different reasons. For example, the Navigation Position example requires both and IRU and an ADC to calculate position. In addition, the tool needs information about what components can and cannot be replicated when trying to improve the probabilities of the failure conditions.

- Failure conditions (Undesired events) and associated probability targets

  To determine if the safety objectives can be met by a given architecture, the failure conditions and target probabilities must be defined. We model the failure condition definition in the context of the components that directly drive the event (e.g. loss of display would need to tie to the output of a display component model). Also, there may need to be failure condition logic modeled in the event of a failure condition relying on the output of multiple components (e.g. loss of all primary displays would need to AND the outputs of all available primary display components).

We identified two distinguishing roles with separate responsibilities: the library owner and the end-user. The library owner defines a library of parameterized components. The library can be company-wide, project-specific, or product-specific. The library is reusable; the Big-C functions are reusable. Its definition is independent of the architecture synthesis task. The library owner is also responsible for defining compatibility constraints for the parameterized components that are project-agnostic. This person could be a safety engineer. The library owner needs to be a super-user of the tool, because writing Big-C functions requires familiarity with OCaml.

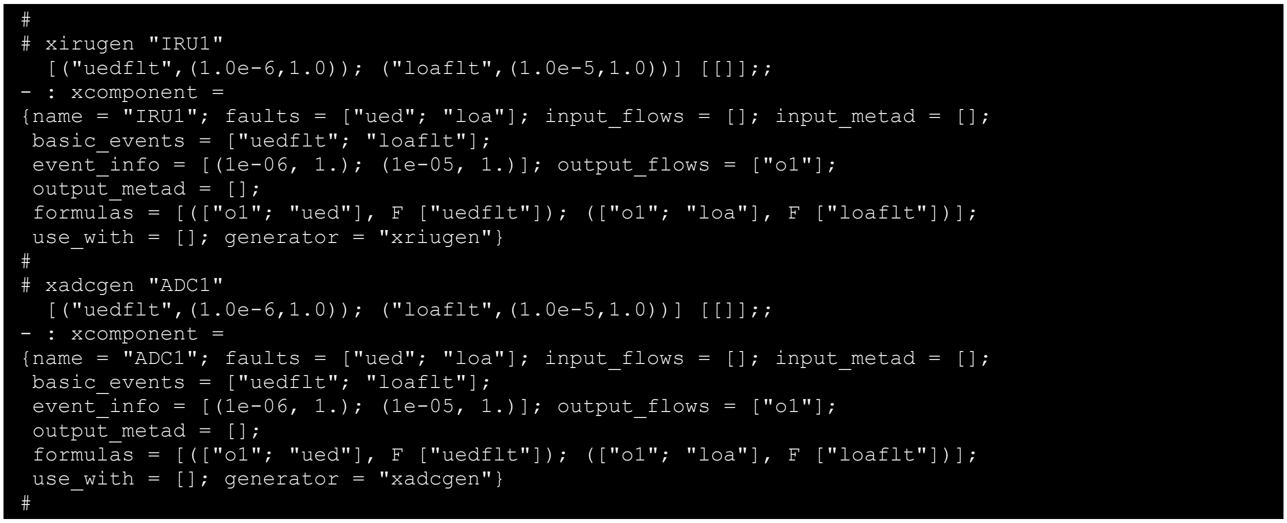
The end-user is the consumer of the library and the Big-C functions. This person provides the inputs to the tool such as the desired probability targets, the usable components including the list of parameters that defines the exploration space, and the list of connection compatibility. This person is a safety engineer or a systems engineer familiar with SOTERIA modeling construct.

### 10.2.1 Big-C functions

We describe the Big-C functions by way of the Navigation Position example in Figure 48. Here's a list of the Big-C functions needed for this problem.

**`xirugen`** and **`xadcgen:`** These two functions generate the class of inertial reference units (IRU) and air data computer (ADC) little-c components, respectively. They have no inputs and one output. Their parameters are the user assigned component names (cn), a list of basic events and event info. See the file navigation_position.ml in the SOTERIA code repository, under examples.

Here are example outputs from calling the Big-C functions xirugen and xadcgen.



```
#
# xirugen "IRU1"
  [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))] [[]];;
- : xcomponent =
{name = "IRU1"; faults = ["ued"; "loa"]; input_flows = []; input_metad = [];
 basic_events = ["uedflt"; "loaflt"];
 event_info = [(1e-06, 1.); (1e-05, 1.)]; output_flows = ["o1"];
 output_metad = [];
 formulas = [(["o1"; "ued"], F ["uedflt"]); (["o1"; "loa"], F ["loaflt"])];
 use_with = []; generator = "xriugen"}
#
# xadcgen "ADC1"
  [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))] [[]];;
- : xcomponent =
{name = "ADC1"; faults = ["ued"; "loa"]; input_flows = []; input_metad = [];
 basic_events = ["uedflt"; "loaflt"];
 event_info = [(1e-06, 1.); (1e-05, 1.)]; output_flows = ["o1"];
 output_metad = [];
 formulas = [(["o1"; "ued"], F ["uedflt"]); (["o1"; "loa"], F ["loaflt"])];
 use_with = []; generator = "xadcgen"}
#
```

**xdcmgen:** This function generates the class of data conversion module (DCM) component with n inputs and n outputs. The inputs propagate errors straight thru to the outputs. See the file navigation_position.ml in the SOTERIA code repository, under examples.

Here are examples of calling xdcmgen.

```
# xdcmgen "DCM1"
  [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))] [["1"]];;

- : xcomponent =
{name = "DCM1"; faults = ["ued"; "loa"]; input_flows = ["i1"];
 input_metad = []; basic_events = ["uedflt"; "loaflt"];
 event_info = [(1e-06, 1.); (1e-05, 1.)]; output_flows = ["o1"];
 output_metad = [];
 formulas =
  [(["o1"; "ued"], Or [F ["i1"; "ued"]; F ["uedflt"]]);
   (["o1"; "loa"], Or [F ["i1"; "loa"]; F ["loaflt"]])];
 use_with = []; generator = "xdcmgen"}
#
# xdcmgen "DCM2"
  [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))] [["2"]];;

- : xcomponent =
{name = "DCM2"; faults = ["ued"; "loa"]; input_flows = ["i1"; "i2"];
 input_metad = []; basic_events = ["uedflt"; "loaflt"];
 event_info = [(1e-06, 1.); (1e-05, 1.)]; output_flows = ["o1"; "o2"];
 output_metad = [];
 formulas =
  [(["o1"; "ued"], Or [F ["i1"; "ued"]; F ["uedflt"]]);
   (["o1"; "loa"], Or [F ["i1"; "loa"]; F ["loaflt"]]);
   (["o2"; "ued"], Or [F ["i2"; "ued"]; F ["uedflt"]]);
   (["o2"; "loa"], Or [F ["i2"; "loa"]; F ["loaflt"]])];
 use_with = []; generator = "xdcmgen"}
#
# xdcmgen "DCM3"
  [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))] [["3"]];;

- : xcomponent =
{name = "DCM3"; faults = ["ued"; "loa"]; input_flows = ["i1"; "i2"; "i3"];
 input_metad = []; basic_events = ["uedflt"; "loaflt"];
 event_info = [(1e-06, 1.); (1e-05, 1.)]; output_flows = ["o1"; "o2"; "o3"];
 output_metad = [];
 formulas =
  [(["o1"; "ued"], Or [F ["i1"; "ued"]; F ["uedflt"]]);
   (["o1"; "loa"], Or [F ["i1"; "loa"]; F ["loaflt"]]);
   (["o2"; "ued"], Or [F ["i2"; "ued"]; F ["uedflt"]]);
   (["o2"; "loa"], Or [F ["i2"; "loa"]; F ["loaflt"]]);
   (["o3"; "ued"], Or [F ["i3"; "ued"]; F ["uedflt"]]);
   (["o3"; "loa"], Or [F ["i3"; "loa"]; F ["loaflt"]])];
 use_with = []; generator = "xdcmgen"}
#
# xdcmgen "DCM12"
  [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))] [["12"]];;

- : xcomponent =
{name = "DCM12"; faults = ["ued"; "loa"];
 input_flows =
  ["i01"; "i02"; "i03"; "i04"; "i05"; "i06"; "i07"; "i08"; "i09"; "i10";
   "i11"; "i12"];
 input_metad = []; basic_events = ["uedflt"; "loaflt"];
 event_info = [(1e-06, 1.); (1e-05, 1.)];
 output_flows =
  ["o01"; "o02"; "o03"; "o04"; "o05"; "o06"; "o07"; "o08"; "o09"; "o10";
   "o11"; "o12"];
 output_metad = [];
 formulas =
  [(["o01"; "ued"], Or [F ["i01"; "ued"]; F ["uedflt"]]);
   (["o01"; "loa"], Or [F ["i01"; "loa"]; F ["loaflt"]]);
   (["o02"; "ued"], Or [F ["i02"; "ued"]; F ["uedflt"]]);
   (["o02"; "loa"], Or [F ["i02"; "loa"]; F ["loaflt"]]);
   (["o03"; "ued"], Or [F ["i03"; "ued"]; F ["uedflt"]]);
   (["o03"; "loa"], Or [F ["i03"; "loa"]; F ["loaflt"]]);
   (["o04"; "ued"], Or [F ["i04"; "ued"]; F ["uedflt"]]);
   (["o04"; "loa"], Or [F ["i04"; "loa"]; F ["loaflt"]]);
   (["o05"; "ued"], Or [F ["i05"; "ued"]; F ["uedflt"]]);
   (["o05"; "loa"], Or [F ["i05"; "loa"]; F ["loaflt"]]);
   (["o06"; "ued"], Or [F ["i06"; "ued"]; F ["uedflt"]]);
   (["o06"; "loa"], Or [F ["i06"; "loa"]; F ["loaflt"]]);
   (["o07"; "ued"], Or [F ["i07"; "ued"]; F ["uedflt"]]);
   (["o07"; "loa"], Or [F ["i07"; "loa"]; F ["loaflt"]]);
   (["o08"; "ued"], Or [F ["i08"; "ued"]; F ["uedflt"]]);
   (["o08"; "loa"], Or [F ["i08"; "loa"]; F ["loaflt"]]);
   (["o09"; "ued"], Or [F ["i09"; "ued"]; F ["uedflt"]]);
   (["o09"; "loa"], Or [F ["i09"; "loa"]; F ["loaflt"]]);
   (["o10"; ...], ...); ...];
 use_with = ...; generator = ...}
#
```

**xswitchgen:** This function generates the class of switch (SW) components. The parameters are [$k_1$, …, $k_n$], where there are n sources and for source i the replication factor is $k_i$. This component has $k_1 + k_2 + k_3 + … + k_n$ inputs and outputs. The inputs propagate errors straight thru to the outputs. See the file navigation_position.ml in the SOTERIA code repository, under examples.

Here are examples of calling xswitchgen_crc and xswitchgen.

```
# xswitchgen_crc "A664sw1"
  [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))]
  [["2";"2"]];;
  - : xcomponent =
{name = "A664sw1"; faults = ["ued"; "loa"];
 input_flows = ["i1"; "i2"; "i3"; "i4"]; input_metad = [];
 basic_events = ["uedflt"; "loaflt"; "crc32_flt"];
 event_info = [(1e-06, 1.); (1e-05, 1.); (2.32830643654e-10, 1.)];
 output_flows = ["o1"; "o2"; "o3"; "o4"]; output_metad = [];
 formulas =
  [(["o1"; "ued"], Or [F ["i1"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
   (["o1"; "loa"], Or [F ["i1"; "loa"]; F ["loaflt"]]);
   (["o2"; "ued"], Or [F ["i2"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
   (["o2"; "loa"], Or [F ["i2"; "loa"]; F ["loaflt"]]);
   (["o3"; "ued"], Or [F ["i3"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
   (["o3"; "loa"], Or [F ["i3"; "loa"]; F ["loaflt"]]);
   (["o4"; "ued"], Or [F ["i4"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
   (["o4"; "loa"], Or [F ["i4"; "loa"]; F ["loaflt"]])];
 use_with = []; generator = "xswitchgen"}
#
# xswitchgen "A664sw2"
  [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))]
  [["2"]];;
  - : xcomponent =
{name = "A664sw2"; faults = ["ued"; "loa"]; input_flows = ["i1"; "i2"];
 input_metad = []; basic_events = ["uedflt"; "loaflt"];
 event_info = [(1e-06, 1.); (1e-05, 1.)]; output_flows = ["o1"; "o2"];
 output_metad = [];
 formulas =
  [(["o1"; "ued"], Or [F ["i1"; "ued"]; F ["uedflt"; "ued"]]);
   (["o1"; "loa"], Or [F ["i1"; "loa"]; F ["loaflt"; "loa"]]);
   (["o2"; "ued"], Or [F ["i2"; "ued"]; F ["uedflt"; "ued"]]);
   (["o2"; "loa"], Or [F ["i2"; "loa"]; F ["loaflt"; "loa"]])];
 use_with = []; generator = "xswitchgen"}
#
```

**xfmagen:** This Big-C function generates the class of flight management application (FMA) components and is the most complex of the Big-C generators. The parameters are:

- [$s_1$, …, $s_n$]: source replication factor per input type
- [$k_1$, …, $k_n$]: channel replication factor per input type
- [$sv_1$, …, $sv_n$]: indicates the voting per source ($sv_i$ out of $s_n$)
- [$cv_1$, …, $sv_n$]: (boolean) indicates whether channel voting happens per input type

This component has $s_1 * k_1 + … + s_n * k_n$ inputs.

This function must be smart enough to come up with the right formulas given the parameters. The number of possible formulas can be quite large depending on the values of the parameters. Here are some small examples of combinations of source replicas, s, and channel replicas, k.

| s=[2;2], k=[2,2] | | |
|---|---|---|
| Source | Replica | Channel |
| 1 | 1 | 1 |
| 1 | 1 | 2 |
| 1 | 2 | 1 |
| 1 | 2 | 2 |
| 2 | 1 | 1 |
| 2 | 1 | 2 |
| 2 | 2 | 1 |
| 2 | 2 | 2 |

| s=[1;2], k=2,2] | | |
|---|---|---|
| Source | Replica | Channel |
| 1 | 1 | 1 |
| 1 | 1 | 2 |
| 2 | 1 | 1 |
| 2 | 1 | 2 |
| 2 | 2 | 1 |
| 2 | 2 | 2 |

Here's an example of how the parameters affect the formulas. For s=[1;2], k=[2;2], sv=[1;1], and cv=[true; false], the UED formula is determined through the following steps. Figure 50 shows the source and channel replication relative to an architecture.

- Step 1 - Channel Replication:
  - Source 1 Ch Vote = True, Source 2 Ch Vote = False
  - Source 1 UED = 111 AND 112
  - Source 2x UED = 211 OR 212
  - Source 2y UED = 221 OR 222
- Step 2 - Source Voting:
  - Source 1 Vote 1 of 1, Source 2 Vote 1 of 2
  - Source 1 UED = Source 1 UED
  - Source 2 UED = Source 2x UED OR Source 2y UED
- Step 3 - Multiple Sources – All used
  - UED = Source 1 UED OR Source 2 UED
- Final Formula:
  - UED = Or[ And[ 111; 112]; Or [ Or[211; 212]; Or[221; 222] ] ]

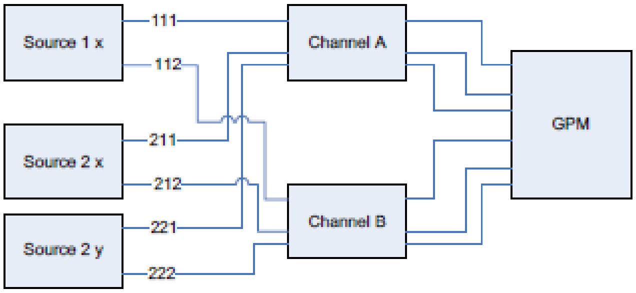


Figure 50 Parameters that define source and channel replications

In the Big-C implementation, we add to the UED formula the N_of construct to make it more generally applicable for voting. The final formula is then:

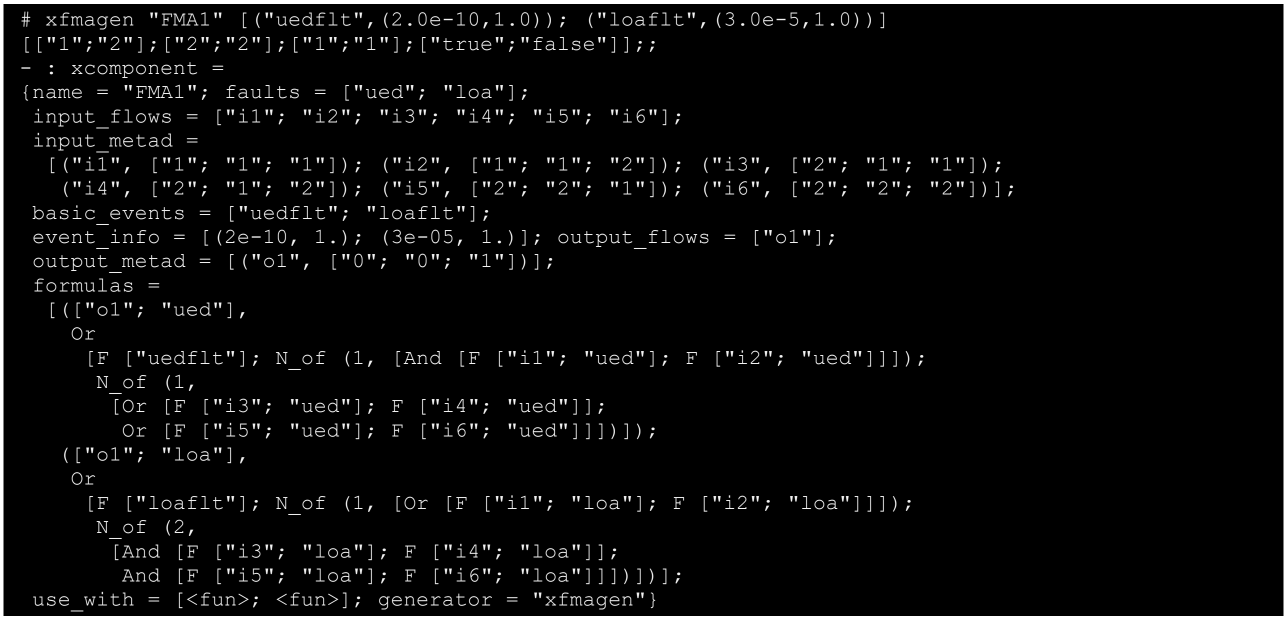
UED = Or[ [***N_of( 1***, And[ 111; 112] )]; [***N_of( 1***, [Or [ Or[211; 212]; Or[221; 222] ] ]])]

The LOA formula is then:

LOA = Or[ [***N_of( 1***, Or[ 111; 112] )]; [***N_of( 1***, [And [ Or[211; 212]; And[221; 222] ] ]])]

We've explained how this function comes up with the right formulas given the parameters. See the file navigation_position.ml in the SOTERIA code repository, under examples.

Here is an example of calling xfmagen.



```
# xfmagen "FMA1" [("uedflt",(2.0e-10,1.0)); ("loaflt",(3.0e-5,1.0))]
[["1";"2"];["2";"2"];["1";"1"];["true";"false"]];;
- : xcomponent =
{name = "FMA1"; faults = ["ued"; "loa"];
 input_flows = ["i1"; "i2"; "i3"; "i4"; "i5"; "i6"];
 input_metad =
  [("i1", ["1"; "1"; "1"]); ("i2", ["1"; "1"; "2"]); ("i3", ["2"; "1"; "1"]);
   ("i4", ["2"; "1"; "2"]); ("i5", ["2"; "2"; "1"]); ("i6", ["2"; "2"; "2"])];
 basic_events = ["uedflt"; "loaflt"];
 event_info = [(2e-10, 1.); (3e-05, 1.)]; output_flows = ["o1"];
 output_metad = [("o1", ["0"; "0"; "1"])];
 formulas =
  [(["o1"; "ued"],
    Or
     [F ["uedflt"]; N_of (1, [And [F ["i1"; "ued"]; F ["i2"; "ued"]]]);
      N_of (1,
       [Or [F ["i3"; "ued"]; F ["i4"; "ued"]];
        Or [F ["i5"; "ued"]; F ["i6"; "ued"]]])]);
   (["o1"; "loa"],
    Or
     [F ["loaflt"]; N_of (1, [Or [F ["i1"; "loa"]; F ["i2"; "loa"]]]);
      N_of (2,
       [And [F ["i3"; "loa"]; F ["i4"; "loa"]];
        And [F ["i5"; "loa"]; F ["i6"; "loa"]]])])];
 use_with = [<fun>; <fun>]; generator = "xfmagen"}
```

There are two more things to notice with the xfmagen function. There is an input meta-data field, `input_metad`, which maps the input flows to source-replica-channel. This function needs to generate this information because the formula depends on the source being connected to the intended input.

Meta-data will also be used by the architecture synthesis to generate parameters, as will be described in a later section. Notice also the `use_with` field (in the case of the FMA, `use_with = [xirugen; xadcgen]`. This field is for the library owner to specify what type of source is meant to be used with this FMA. As the composer of Big-C functions, the library owner has specific ideas about what classes of sources are intended for a class of application.

### 10.2.2 End-User Inputs

This section describes in detail the format of the end-user inputs to SOTERIA for architecture synthesis.

**Target Probabilities:** The end-user lists the targets using 2 fields: target and prob. The target is a string with the end-user defined name and prob is a tuple with the fault and the probability target. The end-user can list as many probability targets in this list. The targets do not have to all be from the same top-level component. Here's what the end-user specifies for the Navigation Position example.

```
let user_probTargets =
[
    {target = "FMA"; prob = ("ued", 1e-7)};
    {target = "FMA"; prob = ("loa", 1e-7)};
];;
```

**Components to Use:** The end-user specifies what components to use. In the same data structure, the end-user specifies the Big-C function that generates the component, the min and max number of the component to include in the architecture, a Boolean specifying whether the component must be used, and the list of basic events and their event info. Below is an example of the usable list for the Navigation Position Example.

```
let usables =
[
("FMA", xfmagen, xdummygen_al, (1,2), true, [("uedflt",(2.0e-10,1.0)); ("loaflt",(3.0e-5,1.0))]);
("A664sw", xswitchgen_crc, xdummygen_al, (1,2), true, [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))]);
("DCM", xdcmgen, xdcmgen_al, (1,3), true, [("uedflt",(1.0e-7,1.0)); ("loaflt",(1.0e-5,1.0))]);
("ADC", xadcgen, xdummygen_al, (1,4), true, [("uedflt",(2.0e-8,1.0)); ("loaflt",(2.0e-5,1.0))]);
("IRU", xirugen, xdummygen_al, (1,3), true, [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))]);
];;
```

(Note that there is another function after the Big-C function, which we call a helper function to generate the argument list. This function is only needed to assist the DCM generator, an issue in the architecture synthesis algorithm which we will describe in section 10.3 on the algorithm approach. Because OCaml is strongly typed, we needed to add dummy functions to the other components even though they are just empty placeholders.)

**Connection Compatibility:** The end-user provides connection compatibility information to express what components can connect to what. This is a list of 3 fields: connectables which is the end-user defined name of the component, connectable_inputs which is a list of end-user defined names in the form of a string, and connectable_outputs which is also a list of end-user defined name in the form of a string. In the Navigation Position example, the processing unit that hosts the FMA is compatible with an A664sw.

An ADC must go through a network switch, A664sw, to reach the FMA. Similarly, an IRU must go through a DCM before it connects to A664sw to reach the FMA.

```
let user_connectables =
[
   {connectable = "FMA";
    connectable_inputs = ["A664sw"];
    connectable_outputs = [] };
   {connectable = "A664sw";
    connectable_inputs = ["ADC"; "DCM"];
    connectable_outputs = ["FMA"] };
   {connectable = "ADC";
    connectable_inputs = [];
    connectable_outputs = ["A664sw"] };
   {connectable = "DCM";
    connectable_inputs = ["IRU"];
    connectable_outputs = ["A664sw"] };
   {connectable = "IRU";
    connectable_inputs = [];
    connectable_outputs = ["DCM"] };
];;
```

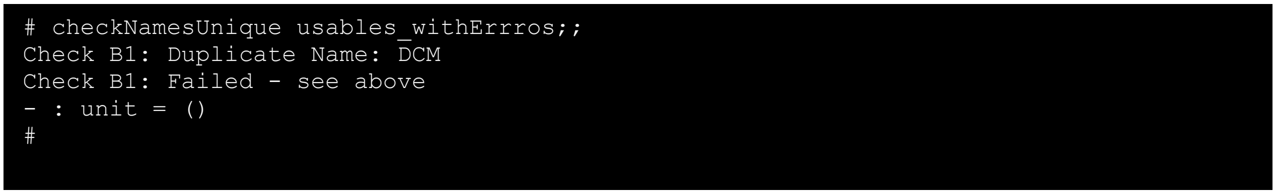
#### *10.2.2.1 Validation Checks*

We developed some validation checks to assist the end-user in providing valid inputs to SOTERIA for architecture synthesis. Below are the usables and user_connectable inputs with errors injected to demonstrate the validation checks.

```
let usables_withErrors =
[
("FMA", xfmagen, xfmagen_al, (1,1), true,
          [("uedflt",(2.0e-10,1.0)); ("loaflt",(3.0e-5,1.0))]);
("A664sw", xswitchgen, xswitchgen_al, (1,2), true,
          [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))]);
("DCM", xdcmgen, xdcmgen_al, (0,3), true,
          [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))]); (* B2 Failure 1 *)
("ADC", xadcgen, xadcgen_al, (2,1), true,
          [("uedflt",(2.0e-8,1.0)); ("loaflt",(2.0e-5,1.0))]); (* B2 Failure 2 *)
("IRU", xirugen, xirugen_al, (1,1), true,
          [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))]);
("DCM", xdcmgen, xdcmgen_al, (1,4), true,
          [("uedflt",(2.0e-6,1.0)); ("loaflt",(3.0e-5,1.0))]); (* B1 Failue 1*)
("MissingMod", xdcmgen, xdcmgen_al, (1,3), true,
          [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))]); (* B3a Failure *)
];;

let user_connectables_withErrors =
[
    {connectable = "FMA";
     connectable_inputs = ["A664sw"];
     connectable_outputs = []};
    {connectable = "A664sw";
     connectable_inputs = ["A664sw2"; "ADC"; "DCM";]; (* B3c Failure *)
     connectable_outputs = ["A664sw"; "FMA"]};
    {connectable = "ADC";
     connectable_inputs = [];
     connectable_outputs = ["A664sw"]};
    {connectable = "ADC2"; (* B3b Failure *)
     connectable_inputs = [];
     connectable_outputs = ["A664sw"]};
    {connectable = "DCM";
     connectable_inputs = ["IRU"];
     connectable_outputs = ["A664sw"]};
    {connectable = "IRU";
     connectable_inputs = [];
    connectable_outputs = ["DCM2"]}; (* B3d Failure *)
];;
```

### 10.2.2.1.1 Names in Usable are Unique

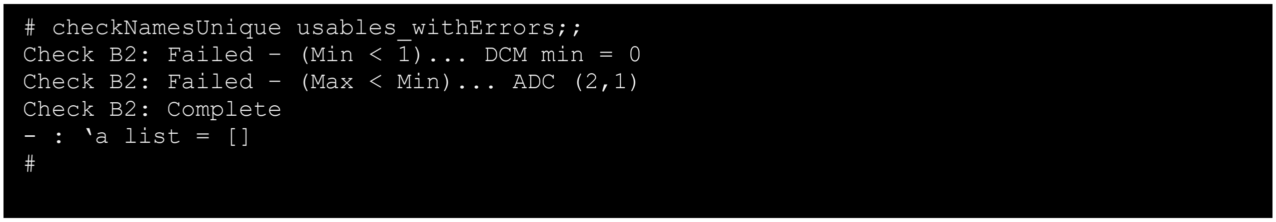
The validation check, `checkNamesUnique,` checks that all the user defined names of the usable components are unique. Here, the check correctly detects that the name “DCM” was used twice.



```
# checkNamesUnique usables_withErrros;;
Check B1: Duplicate Name: DCM
Check B1: Failed - see above
- : unit = ()
#
```

### 10.2.2.1.2 Min >= 1 and Max > Min

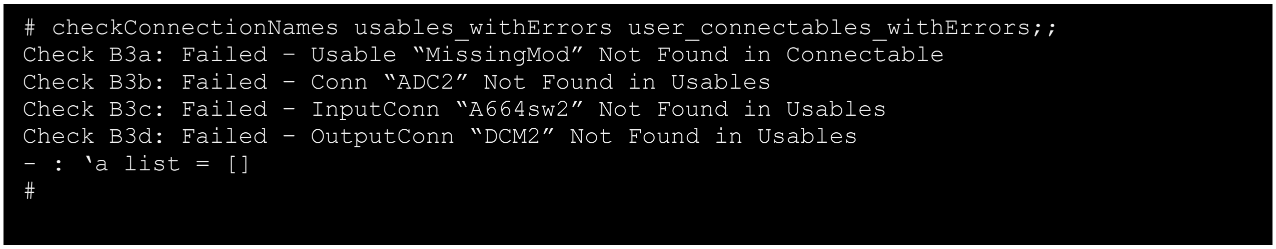
The validation check, `checkNamesUnique`, checks that in the usables the user specified number of minimal components is greater than 0 and less than the specified number of maximal. Here, the check correctly finds that “DCM” minimum is less than 1, and finds that the “ADC” max was less than the min.



```
# checkNamesUnique usables_withErrors;;
Check B2: Failed – (Min < 1)... DCM min = 0
Check B2: Failed – (Max < Min)... ADC (2,1)
Check B2: Complete
- : ‘a list = []
#
```

### 10.2.2.1.3 Connectable Names Match Usable Names

The validation check, `checkConnectionNames`, ensures that the information in the connectables list and usable list are congruent with each other. Here, the validation check correctly finds several things: “MissingMod” usable is not specified in the connectables, “ADC2” is listed in connectables, but is not part of usable list, “A664sw2” is listed as a possible input to a component, but is not in part of the usable list, “DCM2” is listed as a possible output to a component, but is not part of the usable list.



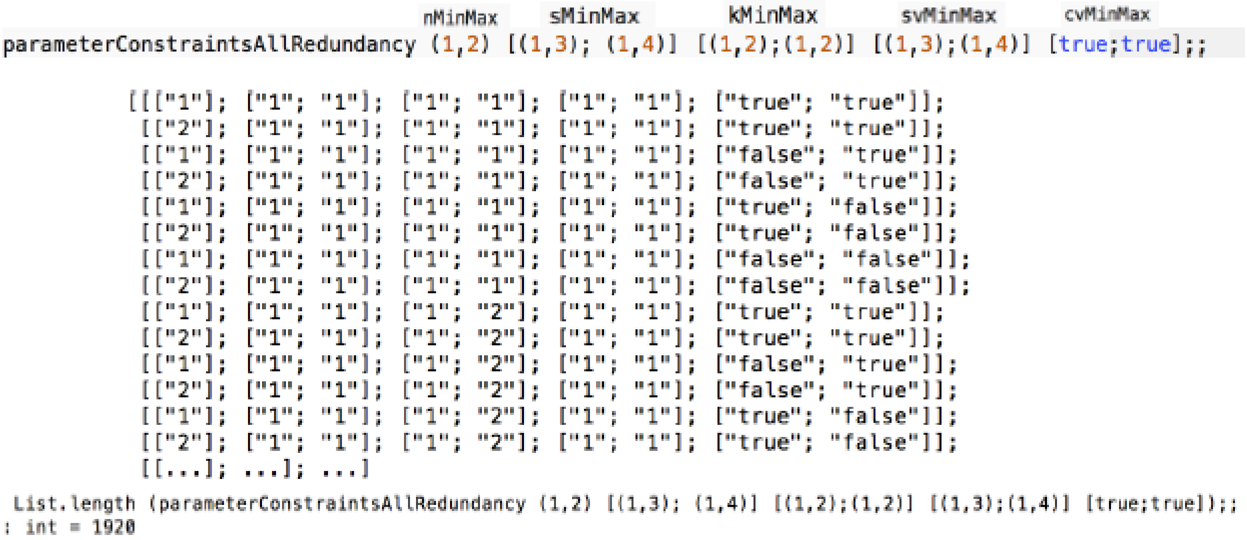
```
# checkConnectionNames usables_withErrors user_connectables_withErrors;;
Check B3a: Failed – Usable “MissingMod” Not Found in Connectable
Check B3b: Failed – Conn “ADC2” Not Found in Usables
Check B3c: Failed – InputConn “A664sw2” Not Found in Usables
Check B3d: Failed – OutputConn “DCM2” Not Found in Usables
- : ‘a list = []
#
```

## 10.3 Architecture Synthesis Goals and Approach

The goal of our architecture synthesis is to automatically generate architectures that meet the end-user specified target probabilities. When there is more than one satisfying solution, our tool will report them all. If all the target probabilities are not met, then our tool will return those that met at least one of the probabilities. Our tool also saves the results in a format that the end-user can further analyze, alter, and explore. Our tool will only offer solutions that “fit” and are usable. In other words, if the end-user specifies a maximum of 3 IRUs, then we don’t want to synthesize architectures that violate this constraint even if it satisfies the probability targets. Our tool will also generate well-formed architectures with legal connections, as specified by the end-user.

To achieve these synthesis goals, we developed an algorithm that takes the end-user inputs (the list of usables, connections, and probability targets), determines the list of possible architectures given the minimum and maximum number of usable components, iterates through the list of possible architectures, and performs a top-down approach to synthesize well-formed architectures with legal connections. The algorithm then performs the safety analysis and calculates the top-level probability.

The algorithm explores the space given the parameter constraints defined by the end-user expressed in usables. The space can be quite large! We gave an example of usables in the previous section, where the FMA can be replicated 1-2 times, the A664sw’s can be replicated 1-2 times, DCMs 1-3 times, ADCs 1-4 times, and IRUs 1-3 times. Figure 51 shows the function that generates the list of all possible architectures under these constraints for which there are as many as 1,920. This includes all the component redundancy options combined with voting options.



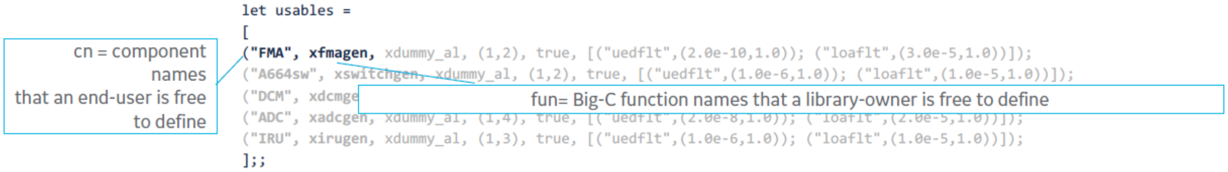
```
                                nMinMax   sMinMax        kMinMax       svMinMax      cvMinMax
parameterConstraintsAllRedundancy (1,2) [(1,3); (1,4)] [(1,2);(1,2)] [(1,3);(1,4)] [true;true];;

        [[["1"]; ["1"; "1"]; ["1"; "1"]; ["1"; "1"]; ["true"; "true"]];
         [["2"]; ["1"; "1"]; ["1"; "1"]; ["1"; "1"]; ["true"; "true"]];
         [["1"]; ["1"; "1"]; ["1"; "1"]; ["1"; "1"]; ["false"; "true"]];
         [["2"]; ["1"; "1"]; ["1"; "1"]; ["1"; "1"]; ["false"; "true"]];
         [["1"]; ["1"; "1"]; ["1"; "1"]; ["1"; "1"]; ["true"; "false"]];
         [["2"]; ["1"; "1"]; ["1"; "1"]; ["1"; "1"]; ["true"; "false"]];
         [["1"]; ["1"; "1"]; ["1"; "1"]; ["1"; "1"]; ["false"; "false"]];
         [["2"]; ["1"; "1"]; ["1"; "1"]; ["1"; "1"]; ["false"; "false"]];
         [["1"]; ["1"; "1"]; ["1"; "2"]; ["1"; "1"]; ["true"; "true"]];
         [["2"]; ["1"; "1"]; ["1"; "2"]; ["1"; "1"]; ["true"; "true"]];
         [["1"]; ["1"; "1"]; ["1"; "2"]; ["1"; "1"]; ["false"; "true"]];
         [["2"]; ["1"; "1"]; ["1"; "2"]; ["1"; "1"]; ["false"; "true"]];
         [["1"]; ["1"; "1"]; ["1"; "2"]; ["1"; "1"]; ["true"; "false"]];
         [["2"]; ["1"; "1"]; ["1"; "2"]; ["1"; "1"]; ["true"; "false"]];
         [[...]; ...]; ...]
 List.length (parameterConstraintsAllRedundancy (1,2) [(1,3); (1,4)] [(1,2);(1,2)] [(1,3);(1,4)] [true;true]);;
: int = 1920
```

Figure 51 All possible architectures given the end-user defined exploration space

Now let us describe our approach to architecture synthesis and our implementation by way of the Navigation Position example from Figure 48. Let’s also take one set of parameters from the list of possible architectures: app redundancy n=`["1"]`, source redundancy s=`["2"; "1"]`, channel redundancy k=`["1"; "2"]`, source voting options sv=`["1"; "1"]`, and channel voting options cv=`["true"; "false"]`. The algorithm starts from the top and works backward from the failure condition. So, in this example, we start with the FMA since the target probabilities specified by the end-user requires an FMA. There is a lot of information from the top: types of sources, since the library-owner specified this information when he composed the Big-C function with the “use-with” field, and the number of sources and switches from the set of architecture parameters. The algorithm will first automatically generate the library, then automatically generate the model with all the right connections, then iterate through the target probability list.

To automatically generate the library, the algorithm recursively generates the list of library components (little-c’s) by calling the Big-C functions. The philosophy is that the auto generated library will be inclusive of every instance that will be used in the model. This is a slightly different use of the modeling language in SOTERIA in that when a model is constructed manually the end-user can make multiple instantiations of the same library component. For the synthesis algorithm, however, the ability to reuse little-c is not that important. In fact, creating all the component instantiations in the automated library simplifies the automatic model creation process, which will be apparent when we describe how the model is automatically generated. The most important thing about auto generating the library is to make sure that all instances of little-c’s specified for a set of parameters are generated. We break the process of auto generating the library into 3 smaller steps: 1) generate the apps, 2) generate the sources, 3) generate the switches. We decompose the problem into these 3 steps because we know that an architecture is always made up of these 3 classes of components.

We use lookup tables to translate the end-user inputs into something that the algorithm can reference. We create a hash table to translate the user defined component name, cn, to the name of the Big-C function that generates it (Figure 52). (We use hash tables, but other data structures can also be used.) Creating this map is necessary because cn is a name that the end-user is free to choose, the Big-C function name is something that the library owner is also free to choose. Similarly, we create a hash table to create a map between cn to connections (Figure 53). Since we already have a validation check to make sure that everything in user_connectables is defined in usables, the algorithm doesn’t have to catch the case when the hash table lookup returns an empty list.

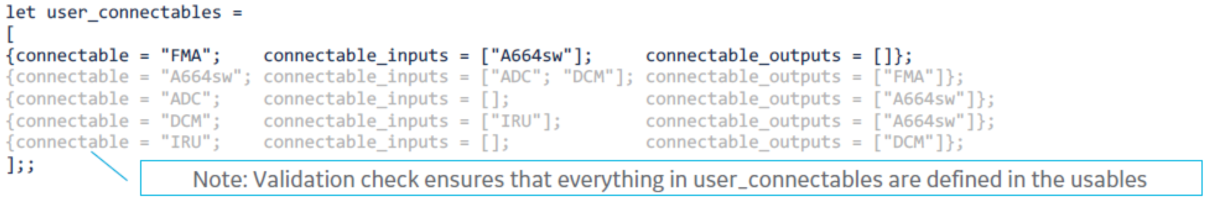


```
let usables =
[
("FMA", xfmagen, xdummy_al, (1,2), true, [("uedflt",(2.0e-10,1.0)); ("loaflt",(3.0e-5,1.0))]);
("A664sw", xswitchgen, xdummy_al, (1,2), true, [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))]);
("DCM", xdcmge
("ADC", xadcgen, xdummy_al, (1,4), true, [("uedflt",(2.0e-8,1.0)); ("loaflt",(2.0e-5,1.0))]);
("IRU", xirugen, xdummy_al, (1,3), true, [("uedflt",(1.0e-6,1.0)); ("loaflt",(1.0e-5,1.0))]);
];;
```

cn = component names that an end-user is free to define

fun= Big-C function names that a library-owner is free to define

Figure 52 Inputs to the cn to Big-C function lookup table



```
let user_connectables =
[
{connectable = "FMA";    connectable_inputs = ["A664sw"];     connectable_outputs = []};
{connectable = "A664sw"; connectable_inputs = ["ADC"; "DCM"]; connectable_outputs = ["FMA"]};
{connectable = "ADC";    connectable_inputs = [];             connectable_outputs = ["A664sw"]};
{connectable = "DCM";    connectable_inputs = ["IRU"];        connectable_outputs = ["A664sw"]};
{connectable = "IRU";    connectable_inputs = [];             connectable_outputs = ["DCM"]};
];;
```

Note: Validation check ensures that everything in user_connectables are defined in the usables

Figure 53 Inputs to the cn connections lookup table

The algorithm needs to keep track of source and channel replication information when allocating parameters for the downstream components. For example, FMA has 2 sources (ADC and IRU) whose parameters are pushed down to the Big-C functions that generate them. The encoding for these sources needs to be consistent throughout the entire model. We use meta-data to specify this information, which was introduced in the FMA Big-C function description. The library owner generates the input meta-data information that maps the input flows to a tuple of (source, replica, channel). The algorithm replaces the generic source numbers with end-user defined component names. The FMA input_metad gets translated in the following way. The “use_with” information in the FMA component specifies which sources are used and in which order.

```
{name         = "FMA1";
 faults       = ["ued"; "loa"];
 input_flows  = ["i1"; "i2"; "i3"; "i4"];
 input_metad  = [("i1",["1";"1";"1"]); ("i2",["1";"2";"1"]); ("i3",["2";"1";"1"]); ("i4",["2";"1";"2"])];
 basic_events = ["uedflt"; "loaflt"];
 event_info   = [(2e-10, 1.); (3e-05, 1.)]; output_flows = ["o1"];
 output_metad = [("o1", ["0"; "0"; "1"])];
 formulas     = [(["o1"; "ued"], Or [F ["uedflt"]; N_of (1, [And [F ["i1"; "ued"]]; And [F ["i2"; "ued"]]]);
                                                    N_of (1, [Or [F ["i3"; "ued"]]; [F ["i4"; "ued"]]])]);
                 (["o1"; "loa"], Or[F ["loaflt"]; N_of (2, [Or [F ["i1"; "loa"]]; Or [F ["i2"; "loa"]]]);
                                                  N_of (1, [And [F ["i3"; "loa"]]; [F ["i4"; "loa"]]])])];
 use_with     = [<fun>; <fun>];
 generator    = "xfmagen"}
```

[xirugen; xadcgen]

```
{name         = "FMA1";
 faults       = ["ued"; "loa"];
 input_flows  = ["i1"; "i2"; "i3"; "i4"];
 input_metad  = [("i1",["IRU";"1";"1"]); ("i2",["IRU";"2";"1"]);
                 ("i3",["ADC";"1";"1"]); ("i4",["ADC";"1";"2"])];
 basic_events = ["uedflt"; "loaflt"];
 event_info   = [(2e-10, 1.); (3e-05, 1.)]; output_flows = ["o1"];
 output_metad = [("o1", ["0"; "0"; "1"])];
 formulas     = [(["o1"; "ued"], Or [F ["uedflt"]; N_of (1, [And [F ["i1"; "ued"]]; And [F ["i2"; "ued"]]]);
                                                    N_of (1, [Or [F ["i3"; "ued"]]; [F ["i4"; "ued"]]])]);
                 (["o1"; "loa"], Or[F ["loaflt"]; N_of (2, [Or [F ["i1"; "loa"]]; Or [F ["i2"; "loa"]]]);
                                                  N_of (1, [And [F ["i3"; "loa"]]; [F ["i4"; "loa"]]])])];
 use_with     = [<fun>; <fun>];
 generator    = "xfmagen"}
```

The algorithm then uses the meta-data list from the app as parameters and recursively goes through the list to generate the sources. Notice that the meta-data list is of length 4, but there are only 3 sources: IRU1, IRU2, and ADC1. The algorithm is smart enough to figure out that there are only 3 sources, and so only generates 2 instances of IRU and 1 instance of ADC. The Big-C function for the sources generates a blank output meta-data list. The synthesis algorithm fills in the list with relevant information. Notice that the source meta-data information does not have channel information.

```
{name = "IRU1"; faults = ["ued"; "loa"]; input_flows = []; input_metad = [];
 basic_events = ["uedflt"; "loaflt"]; event_info = [(1e-06, 1.); (1e-05, 1.)];
 output_flows = ["o1"]; output_metad = [(“o1", ["IRU"; "1"])];
 formulas = [(["o1"; "ued"], F ["uedflt"]); (["o1"; "loa"], F ["loaflt"])];
 use_with = []; generator = "xriugen"};
{name = "IRU2"; faults = ["ued"; "loa"]; input_flows = []; input_metad = [];
 basic_events = ["uedflt"; "loaflt"]; event_info = [(1e-06, 1.); (1e-05, 1.)];
 output_flows = ["o1"]; output_metad = [(“o1", ["IRU"; "2"])];
 formulas = [(["o1"; "ued"], F ["uedflt"]); (["o1"; "loa"], F ["loaflt"])];
 use_with = []; generator = "xriugen"};
{name = "ADC1"; faults = ["ued"; "loa"]; input_flows = []; input_metad = [];
 basic_events = ["uedflt"; "loaflt"]; event_info = [(2e-08, 1.); (2e-05, 1.)];
 output_flows = ["o1"]; output_metad = [(“o1", ["ADC"; "1"])];
 formulas = [(["o1"; "ued"], F ["uedflt"]); (["o1"; "loa"], F ["loaflt"])];
 use_with = []; generator = "xadcgen"};
```

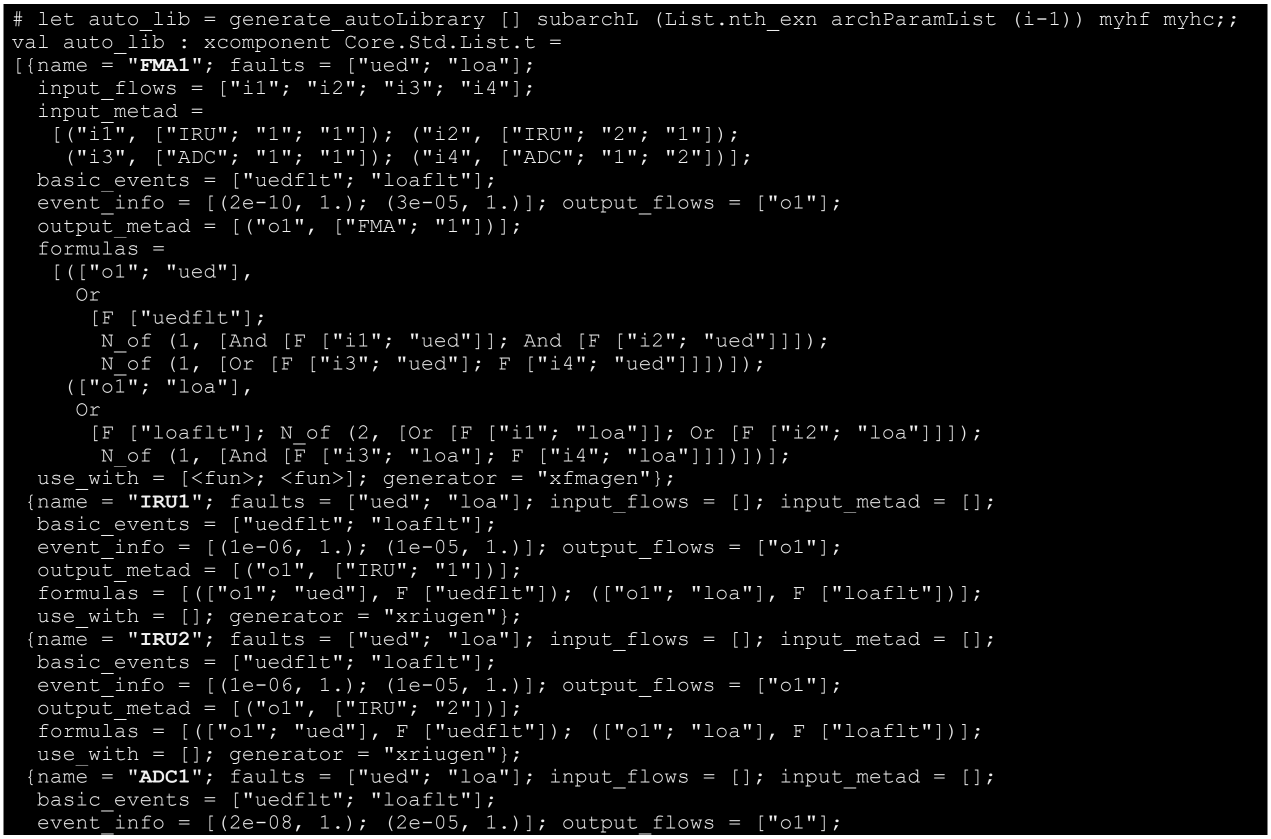
The algorithm uses the meta-data list from the app as parameters to generate the switches as well. It parses the meta-data to figure out the network replication required for each source.

```
Switch 1 List = [("i1", ["IRU"; "1"; "1"]); ("i2", ["IRU"; "2"; "1"]); ("i3", ["ADC"; "1"; "1"])];
Switch 2 List = [("i4", ["ADC"; "1"; “2"])];
```

It uses the connectables to find what connects to the app and gets the end-user defined cn names for the switches. The Big-C function for the switches generates a blank input and output meta-data lists. The synthesis algorithm fills in the lists with relevant information about the sources.

```
{name = "A664sw2"; faults = ["ued"; "loa"]; input_flows = ["i1"];
 input_metad = [("i1", ["ADC"; "1"; "2"])];
 basic_events = ["uedflt"; "loaflt"; "crc32_flt"];
 event_info = [(1e-06, 1.); (1e-05, 1.); (2.32830643654e-10, 1.)];
 output_flows = ["o1"]; output_metad = [("o1", ["ADC"; "1"; "2"])];
 formulas = [(["o1"; "ued"], Or [F ["i1"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
             (["o1"; "loa"], Or [F ["i1"; "loa"]; F ["loaflt"]])];
 use_with = []; generator = "xswitchgen"}

{name = "A664sw1"; faults = ["ued"; "loa"]; input_flows = ["i1"; "i2"; "i3"];
 input_metad =
  [("i1", ["IRU"; "1"; "1"]); ("i2", ["IRU"; "2"; "1"]); ("i3", ["ADC"; "1"; "1"])];
 basic_events = ["uedflt"; "loaflt"; "crc32_flt"];
 event_info = [(1e-06, 1.); (1e-05, 1.); (2.32830643654e-10, 1.)];
 output_flows = ["o1"; "o2"; "o3"];
 output_metad =  [("o1", ["IRU"; "1"; "1"]); ("o2", ["IRU"; "2"; "1"]);
                  ("o3", ["ADC"; "1"; "1"])];
 formulas = [(["o1"; "ued"], Or [F ["i1"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
             (["o1"; "loa"], Or [F ["i1"; "loa"]; F ["loaflt"]]);
             (["o2"; "ued"], Or [F ["i2"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
             (["o2"; "loa"], Or [F ["i2"; "loa"]; F ["loaflt"]]);
             (["o3"; "ued"], Or [F ["i3"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
             (["o3"; "loa"], Or [F ["i3"; "loa"]; F ["loaflt"]])];
 use_with = []; generator = "xswitchgen"}
```

The auto generated library up to this point is shown below.



```
# let auto_lib = generate_autoLibrary [] subarchL (List.nth_exn archParamList (i-1)) myhf myhc;;
val auto_lib : xcomponent Core.Std.List.t =
[{name = "FMA1"; faults = ["ued"; "loa"];
  input_flows = ["i1"; "i2"; "i3"; "i4"];
  input_metad =
   [("i1", ["IRU"; "1"; "1"]); ("i2", ["IRU"; "2"; "1"]);
    ("i3", ["ADC"; "1"; "1"]); ("i4", ["ADC"; "1"; "2"])];
  basic_events = ["uedflt"; "loaflt"];
  event_info = [(2e-10, 1.); (3e-05, 1.)]; output_flows = ["o1"];
  output_metad = [("o1", ["FMA"; "1"])];
  formulas =
   [(["o1"; "ued"],
     Or
      [F ["uedflt"];
       N_of (1, [And [F ["i1"; "ued"]]; And [F ["i2"; "ued"]]]);
       N_of (1, [Or [F ["i3"; "ued"]; F ["i4"; "ued"]]])]);
    (["o1"; "loa"],
     Or
      [F ["loaflt"]; N_of (2, [Or [F ["i1"; "loa"]]; Or [F ["i2"; "loa"]]]);
       N_of (1, [And [F ["i3"; "loa"]; F ["i4"; "loa"]]])])];
  use_with = [<fun>; <fun>]; generator = "xfmagen"};
 {name = "IRU1"; faults = ["ued"; "loa"]; input_flows = []; input_metad = [];
  basic_events = ["uedflt"; "loaflt"];
  event_info = [(1e-06, 1.); (1e-05, 1.)]; output_flows = ["o1"];
  output_metad = [("o1", ["IRU"; "1"])];
  formulas = [(["o1"; "ued"], F ["uedflt"]); (["o1"; "loa"], F ["loaflt"])];
  use_with = []; generator = "xriugen"};
 {name = "IRU2"; faults = ["ued"; "loa"]; input_flows = []; input_metad = [];
  basic_events = ["uedflt"; "loaflt"];
  event_info = [(1e-06, 1.); (1e-05, 1.)]; output_flows = ["o1"];
  output_metad = [("o1", ["IRU"; "2"])];
  formulas = [(["o1"; "ued"], F ["uedflt"]); (["o1"; "loa"], F ["loaflt"])];
  use_with = []; generator = "xriugen"};
 {name = "ADC1"; faults = ["ued"; "loa"]; input_flows = []; input_metad = [];
  basic_events = ["uedflt"; "loaflt"];
  event_info = [(2e-08, 1.); (2e-05, 1.)]; output_flows = ["o1"];
  output_metad = [("o1", ["ADC"; "1"])];
  formulas = [(["o1"; "ued"], F ["uedflt"]); (["o1"; "loa"], F ["loaflt"])];
  use_with = []; generator = "xadcgen"};
 {name = "A664sw2"; faults = ["ued"; "loa"]; input_flows = ["i1"];
  input_metad = [("i1", ["ADC"; "1"; "2"])];
  basic_events = ["uedflt"; "loaflt"; "crc32_flt"];
  event_info = [(1e-06, 1.); (1e-05, 1.); (2.32830643654e-10, 1.)];
  output_flows = ["o1"]; output_metad = [("o1", ["ADC"; "1"; "2"])];
  formulas =
   [(["o1"; "ued"], Or [F ["i1"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
    (["o1"; "loa"], Or [F ["i1"; "loa"]; F ["loaflt"]])];
  use_with = []; generator = "xswitchgen"}
 {name = "A664sw1"; faults = ["ued"; "loa"]; input_flows = ["i1"; "i2"; "i3"];
  input_metad =
  [("i1", ["IRU"; "1"; "1"]); ("i2", ["IRU"; "2"; "1"]);
   ("i3", ["ADC"; "1"; "1"])];
  basic_events = ["uedflt"; "loaflt"; "crc32_flt"];
  event_info = [(1e-06, 1.); (1e-05, 1.); (2.32830643654e-10, 1.)];
  output_flows = ["o1"; "o2"; "o3"];
  output_metad =
   [("o1", ["IRU"; "1"; "1"]); ("o2", ["IRU"; "2"; "1"]);
    ("o3", ["ADC"; "1"; "1"])];
  formulas =
  [(["o1"; "ued"], Or [F ["i1"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
   (["o1"; "loa"], Or [F ["i1"; "loa"]; F ["loaflt"]]);
   (["o2"; "ued"], Or [F ["i2"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
   (["o2"; "loa"], Or [F ["i2"; "loa"]; F ["loaflt"]]);
   (["o3"; "ued"], Or [F ["i3"; "ued"]; And [F ["uedflt"]; F ["crc32_flt"]]]);
   (["o3"; "loa"], Or [F ["i3"; "loa"]; F ["loaflt"]])]; use_with = [];
  generator = "xswitchgen"}]
```

There is an in-between layer missing, which is the DCM. The issue is that the set of parameters from the list of possible architectures (n=["1"], s=["2"; "1"], and k=["1"; "2"]) does not infer the parameters for the DCM. The end-user has described what components can connect to the DCM and the number of times the DCM can be replicated. With that information, the algorithm knows it has several choices for the DCM. In this example, because there are 2 IRUs that must connect to DCMs, it can generate 1 DCM with 2 inputs, or 2 DCMs with 1 input each. The way the algorithm adds these options to the auto generated library is to first go through the list of usables and determines what hasn't been generated yet. Then it uses the lookup tables to find the Big-C generator for the missing components. Then, the parameters are generated with the use of a helper function.

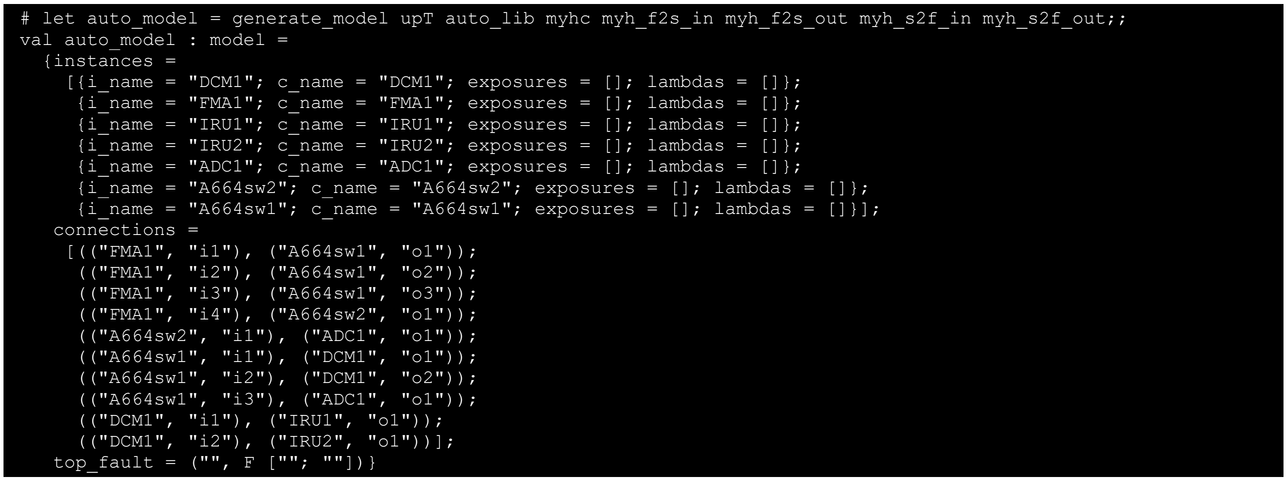
```
# xdcmgen_al 2;;
- : string list list Core.Std.List.t Core.Std.List.t =[[[["1"]]; [["1"]]]; [[["2"]]]]
```

Finally, the algorithm automatically generates the model. Recall that the model is made up of the following fields:

```
model =
{ instances
  connections
  top_fault
}
```

For the instances, all the algorithm does is take the auto generated library and makes an instance of each component. This is because of our philosophy that the auto generated library is inclusive of every instance that will be used in the model. For the connections, all the algorithm does is match the meta-data. This is why there was so much effort in updating the meta-data when the components were being generated in the library. For the top_fault, since the library and connections are already generated, the algorithm just iterates through the list of end-user target probabilities and performs the FTA for each.

The auto generate model is shown below. Notice that the top_fault is intentionally left empty to be filled when iterating through the targets.



```
# let auto_model = generate_model upT auto_lib myhc myh_f2s_in myh_f2s_out myh_s2f_in myh_s2f_out;;
val auto_model : model =
  {instances =
    [{i_name = "DCM1"; c_name = "DCM1"; exposures = []; lambdas = []};
     {i_name = "FMA1"; c_name = "FMA1"; exposures = []; lambdas = []};
     {i_name = "IRU1"; c_name = "IRU1"; exposures = []; lambdas = []};
     {i_name = "IRU2"; c_name = "IRU2"; exposures = []; lambdas = []};
     {i_name = "ADC1"; c_name = "ADC1"; exposures = []; lambdas = []};
     {i_name = "A664sw2"; c_name = "A664sw2"; exposures = []; lambdas = []};
     {i_name = "A664sw1"; c_name = "A664sw1"; exposures = []; lambdas = []}];
   connections =
    [(("FMA1", "i1"), ("A664sw1", "o1"));
     (("FMA1", "i2"), ("A664sw1", "o2"));
     (("FMA1", "i3"), ("A664sw1", "o3"));
     (("FMA1", "i4"), ("A664sw2", "o1"));
     (("A664sw2", "i1"), ("ADC1", "o1"));
     (("A664sw1", "i1"), ("DCM1", "o1"));
     (("A664sw1", "i2"), ("DCM1", "o2"));
     (("A664sw1", "i3"), ("ADC1", "o1"));
     (("DCM1", "i1"), ("IRU1", "o1"));
     (("DCM1", "i2"), ("IRU2", "o1"))];
   top_fault = ("", F [""; ""])}
```

## 10.4 Results

In this section, we show the results of using our special purpose architecture synthesis algorithm.

We successfully synthesized a satisfying solution for the Navigation Position Example. Figure 54 shows a sampling of 3 out of the 1,920 possible architectures, including a visualization of the example from the set of parameters we demonstrated in the previous section. Figure 55 shows the architecture that satisfies both UED and LOA target probabilities.

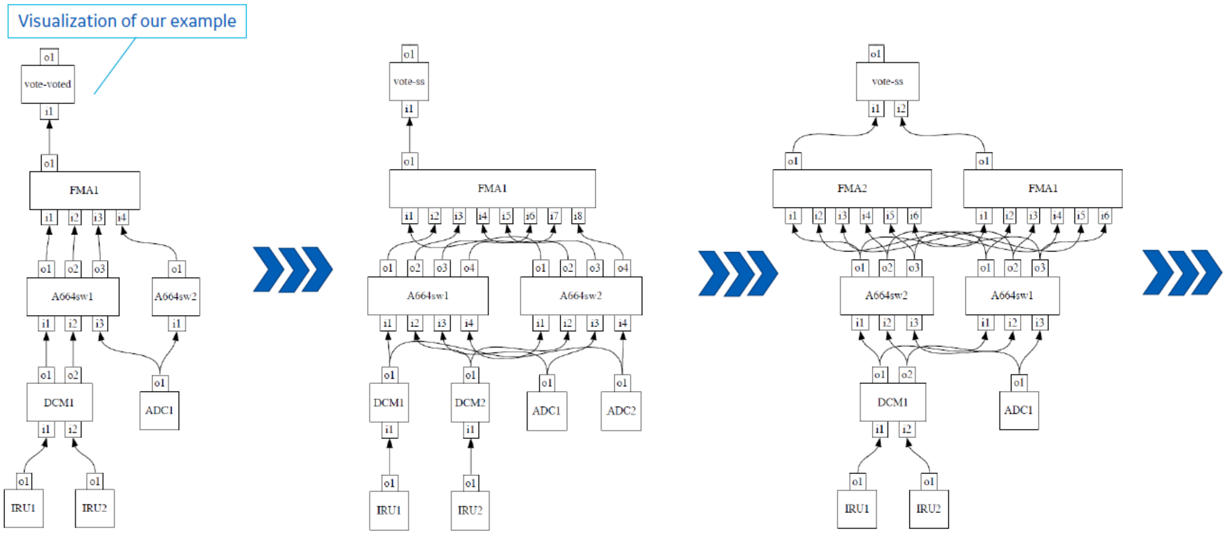


Figure 54 Sampling of the Navigation Position exploration

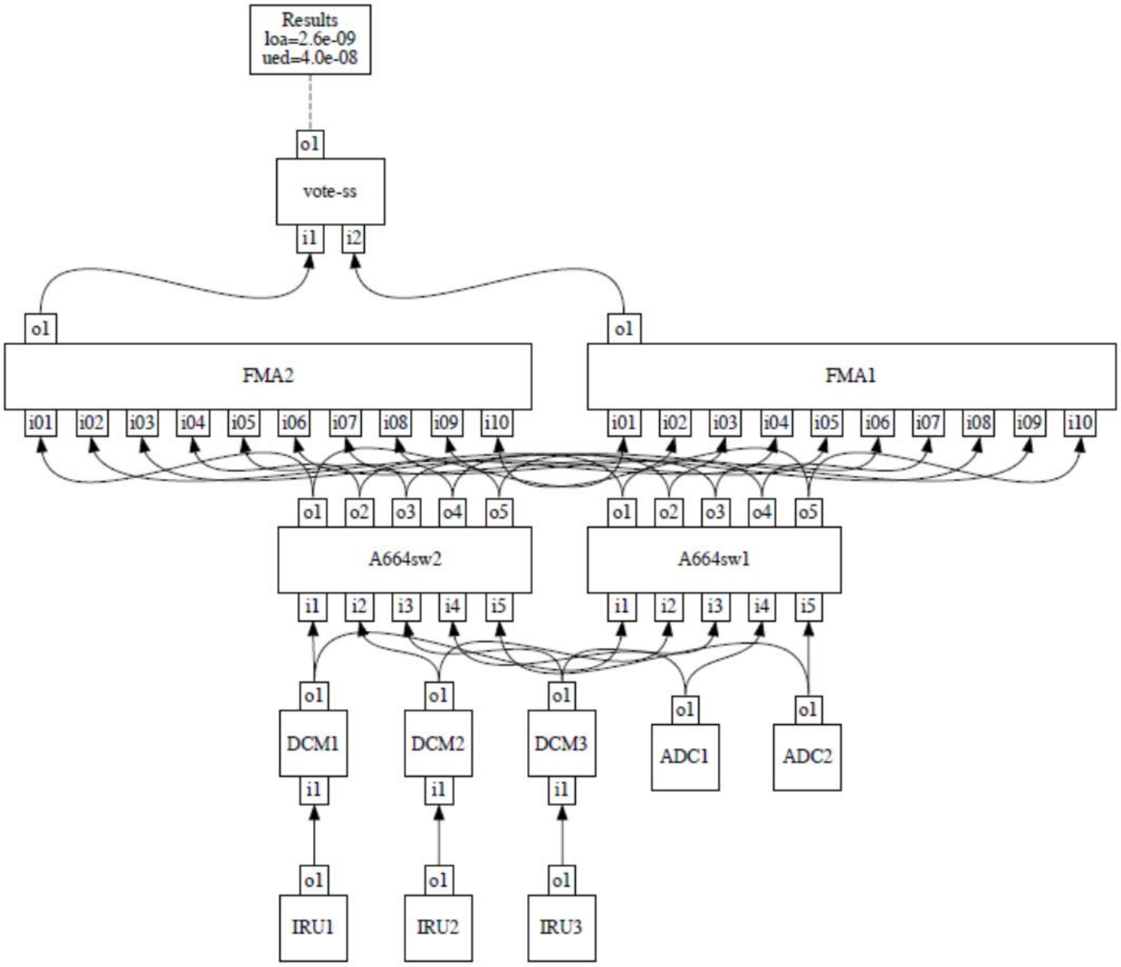
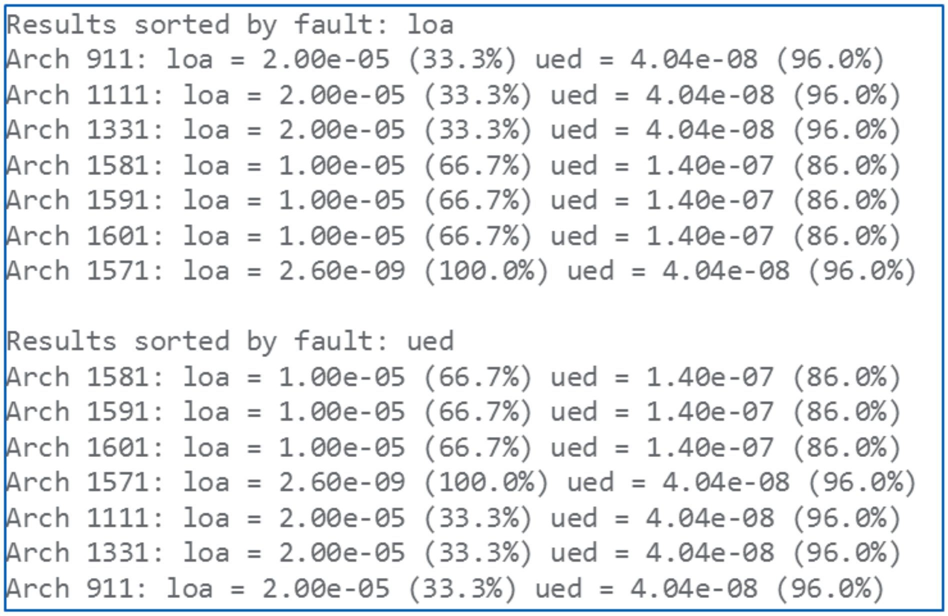


Figure 55 Satisfying architecture for the Navigation Position example

Since multiple architectures can satisfy the targets, the algorithm processes the results and sorts them based on a “goodness” factor, which we define as (result – objective) / objective = % better than target. The lower the goodness factor, the least viable the architecture. Figure 56 shows the results sorted for the Navigation Position example. Note that we artificially modified the targets to LOA = 3e-5 and UED = 1e-6 to show more satisfying architectures to demonstrate the sorting.



```
Results sorted by fault: loa
Arch 911: loa = 2.00e-05 (33.3%) ued = 4.04e-08 (96.0%)
Arch 1111: loa = 2.00e-05 (33.3%) ued = 4.04e-08 (96.0%)
Arch 1331: loa = 2.00e-05 (33.3%) ued = 4.04e-08 (96.0%)
Arch 1581: loa = 1.00e-05 (66.7%) ued = 1.40e-07 (86.0%)
Arch 1591: loa = 1.00e-05 (66.7%) ued = 1.40e-07 (86.0%)
Arch 1601: loa = 1.00e-05 (66.7%) ued = 1.40e-07 (86.0%)
Arch 1571: loa = 2.60e-09 (100.0%) ued = 4.04e-08 (96.0%)

Results sorted by fault: ued
Arch 1581: loa = 1.00e-05 (66.7%) ued = 1.40e-07 (86.0%)
Arch 1591: loa = 1.00e-05 (66.7%) ued = 1.40e-07 (86.0%)
Arch 1601: loa = 1.00e-05 (66.7%) ued = 1.40e-07 (86.0%)
Arch 1571: loa = 2.60e-09 (100.0%) ued = 4.04e-08 (96.0%)
Arch 1111: loa = 2.00e-05 (33.3%) ued = 4.04e-08 (96.0%)
Arch 1331: loa = 2.00e-05 (33.3%) ued = 4.04e-08 (96.0%)
Arch 911: loa = 2.00e-05 (33.3%) ued = 4.04e-08 (96.0%)
```

Figure 56 Solutions sorted based on a "goodness" factor

The Display of Airspeed example (Figure 49) is a little different in that the meta-data changes mid-way in the architecture. The inputs to the Disp App are relevant to ADC outputs, while the inputs to the Cockpit Display are not relevant to ADC, but are relevant to the Disp App outputs. To handle this, we altered our algorithm to handle sub-architectures, where sub-architectures are defined as applications, switches, and sources. The Display of Airspeed example can be broken into two sub-architectures: Display App – A664 switches – ADCs; Cockpit Displays – A664 switches – Display App. Notice that the Display App is the top of the first sub-architecture, but becomes the source in the second sub-architecture. The Display App and the Cockpit Display components are similar in that they both take multiple inputs to formulate a single output. The Navigation Position example (Figure 48) can be thought of as just one sub-architecture: FMA – A664 switches – ADC/DCM+IRUs.

Despite the differences in the two challenge problems, we were able to synthesize architectures for both using the same algorithm. Figure 57 shows the synthesized architecture for the Display of Airspeed example. We were able to synthesize an architecture with all the legal connections, but we ran into limitations that we did not have time to resolve on this program. The synthesized architecture has two levels of switches, A664swA and A664swB. The challenge problem specified only one layer of switches. What’s missing in the synthesis algorithm is the ability to aggregate the layer of switches so that dataflow revisits a component. This is doable, but we simply ran out of time on this program.

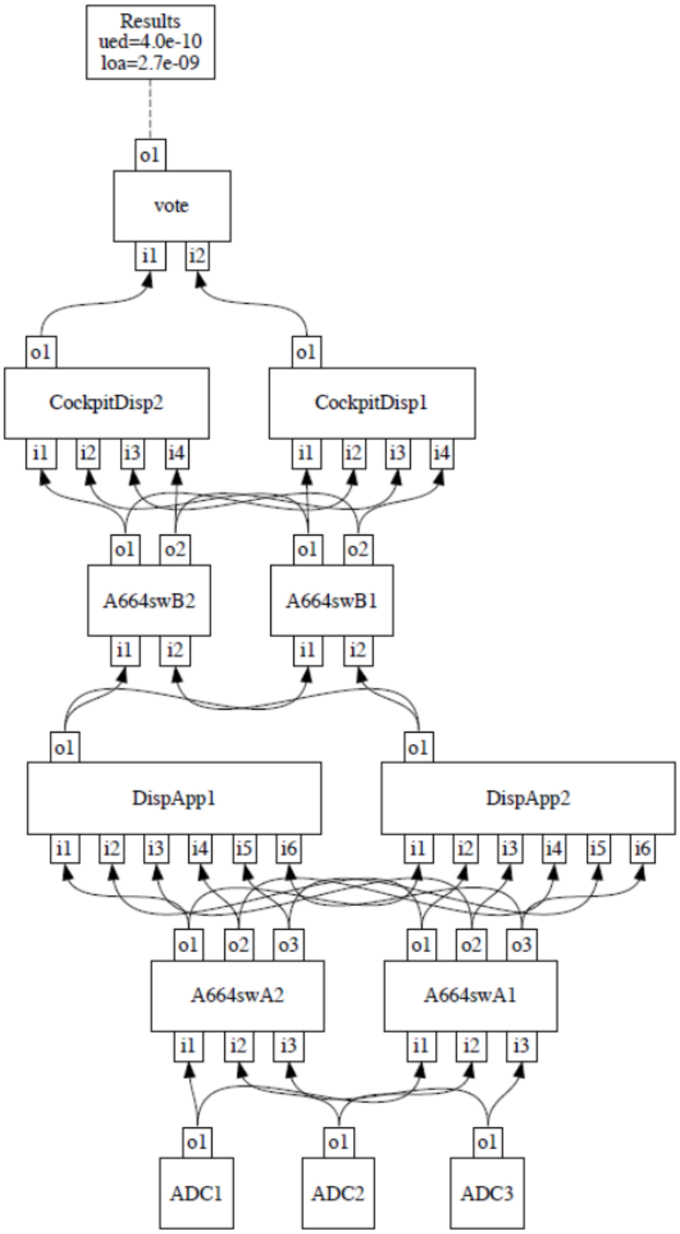


Figure 57 Synthesized architecture for Display of Airspeed example

## 10.5 Possible Extensions

There are several possible extensions to the work that we did on synthesizing architectures. We could alter the algorithm to optimize based on an objective function. Objectives could include minimizing the number of x component or minimizing based on the cost of a component. We could direct the exploration based on feedback from the cut-set, something that we did not have time to incorporate into our algorithm. The exploration could have been distinguished between architecture component changes versus logical computation/voting changes. Some of the architecture observations we did not get to exercise with our challenge problems include CRC compatibility, project-agnostic vs. project-specific connection constraints, and elimination of compatible components whose fault does not contribute to the top-level target. Also, our algorithm only considered functional connections and not physical connection limitations. For example, we did not consider physical limitations on the number of ports in a switch.

# 11 Conclusion

In this report, we detailed the work we did to advance safety analysis techniques through model-based fault-tree synthesis and architecture synthesis. The modeling language we developed on SOTERIA is compositional and enables rapid development, modification, and evaluation of system architectures. On this program, we analyzed IMA systems, but we believe that our modeling language is generally applicable. Current safety analysis techniques require the manual construction of fault trees, which is an error prone process. The tool we developed on this program provides a new paradigm that automates much of the safety analysis process.

Our architecture synthesis approach allows users to automatically explore possible architectures given constraints to a design space. The combination of component connections and voting options can lead to a large space, so automating the exploration process makes a lot of sense. We demonstrated that our tool can synthesize an architecture that meets the safety probability targets for one of our challenge problems, and can synthesize an architecture to our second challenge problem, but with some limitations. We proposed several possible extensions to further our architecture synthesis approach.

# 12 Acknowledgements

The team gratefully acknowledges the following people for their contributions to the project and for their unwavering support throughout the three years of this program: Hongwei Liao for his work on evaluating prior-art and coming up with the first incarnation of our architecture synthesis algorithm, Camila Rodriguez for her work in validating the B777 and B787 examples, Rich Haadsma for his work on the Wheel Brake/Landing Gear example, Michael Durling for his guidance, and Gary Quackenbush for his support.

# 13 Works Cited

AAIB (UK). (2003). *AAIB Bulletin 2/2005 EW/C2003/08/11, Airbus A3320-200, C-FTDF, Cardiff UK, 3 Aug 2003.* Retrieved from Skybrary: http://www.skybrary.aero/bookshelf/books/376.pdf

Aeronautical Radio Inc. (ARINC). (2014). *ARINC Characteristic 702A Advance Flight Management Computer System.*

Andrews, J. (1998). *Tutorial - Fault Tree Analysis*. Retrieved July 2015, from Selected Papers on Reliability: http://www.ewp.rpi.edu/hartford/~ernesto/S2008/SMRE/Papers/Andrews-FTA-tutor.pdf

Banach, R., & Bozzano, M. (2006). Retrenchment, and the generation of fault trees for static, dynamic. *SAFECOMP*, 127-141.

Bozzano, M., Cimatti, A., Fernandes Pires, A., Jones, D., Kimberly, G., Petri, T., . . . Tonetta, S. (2015). Formal Design and Safety Analysis of AIR6110 Wheel Brake System. In *Computer Aided Verification* (pp. 518-535). Springer International Publishing. Retrieved from https://es-static.fbk.eu/people/bozzano/publications/cav2015a.pdf

Contini, S., & Squellati, G. (1984). *Automated Fault Tree Construction.* Luxembourg: Commission of the European Communities.

DO-297. (2005). *DO-297 IMA Development Guidance and Certification Issues Document.* RTCA.

Du-pont, A. (2013, August 21). 787-8 Dreamliner, AeroMexico [digital image]. Retrieved from http://cdn-www.airliners.net/aviation-photos/photos/1/5/4/2314451.jpg

Ericson II, C. (1999). *Fault Tree Analysis: A tutorial, pdf.* Retrieved July 2015, from TheCourse(TM) for Project Management - Course Materials: http://www.thecourse-pm.com/Library/FaultTreeAnalysis2.pdf

FAA. (2011). *FAA Advisory Circular: Installation of Electronic Display in Part 23 Airplanes.* FAA.

FAA. (2014). *FAA Advisory Circular: Electronic Flight Displays.*

*github.com/vasilisp/inez*. (n.d.). Retrieved from Inez: https://github.com/vasilisp/inez

Hang, C., Manolios, P., & Papavasileiou, V. (2011). Synthesizing Cyber-Physical Architectural Models with Real-Time Constraints. *International Conference on Computer Aided Verification (CAV)* (pp. 441-456). Snowbird, UT: Springer.

*janestreet.github.io*. (n.d.). Retrieved from Open Source @ Jane Street: http://janestreet.github.io/

Jensen, D. (2005, November 1). B787 Cockpit: Boeing's Bold Move. *Avionics*. Retrieved from Access Intelligence.

Majdara, A., & Wakabayashi, T. (2009). Component-based modeling of systems for automated fault tree generation. *Reliability Engineering and System Safety*, 1076-1086.

Manolios, P., & Papavasileiou, V. (2013). ILP Modulo Theories. *25th International Conference on Computer Aided Verification (CAV)* (pp. 662-677). Saint Petersburg, Russia: Springer.

Manolios, P., Siu, K., Noorman, M., & Liao, H. (2017). *A Model-Based Framework for Modeling, Visualizing, and Analyzing the Safety of System Architectures.*

Meister, J. (2014, July 15). *Esper Tech Blog*. Retrieved from Why We Use OCaml: https://tech.esper.com/2014/07/15/why-we-use-ocaml/

Minsky, Y. (2016, January 25). *Why OCaml?* Retrieved from Jane Street Tech Blog: https://blog.janestreet.com/why-ocaml/

Moir, I., Seabridge, A., & Jukes, M. (2013). *Civil Avionics Systems, Second Edition.* John Wiley & Sons, Ltd.

(n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Spoiler_(aeronautics)

(n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Lufthansa_Flight_2904

NASA. (2002). *Fault Tree Handbook with Aerospace Applications.* NASA.

S18. (1996). *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.* SAE International.

SAE. (2011). *SAE: AIR 6110, Contiguous Aircraft/System Development Process Example.*

Salem, S., Apostolakis, G., & Okent, D. (1976). *A computer-oriented approach to fault tree construction.* Palo Alto, TX: EPRI NP-288.

U.S. Nuclear Regulatory Commission. (1981). *Fault Tree Handbook.* Washington, D.C.: NUREG.

Waclena, K. (2006, June 17). *OCaml for the Skeptical*. Retrieved from Why OCaml: http://www2.lib.uchicago.edu/keith/ocaml-class/why.html

Wang, Y. (2004). *Develoment of a Computer-Aided Fault Tree Synthesis Methodology for Quantitative Risk Analysis in the Chemical Process Industry.* Texas A&M University.

Wolfig, R., & Jakovlijevic, M. (2008). Distributed IMA and DO-297: Archtictural, Communication and Certification Attributes. *Digital Avionics Systems Conference, 2008* (pp. 1.E.4-1-1.E.4-10). St.Paul, MN: IEEE.

*www.ocaml.org*. (n.d.). Retrieved from OCaml: www.ocaml.org

| | |
|---|---|
| **REPORT DOCUMENTATION PAGE** | *Form Approved*<br>*OMB No. 0704–0188* |

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YYYY)*<br>01-06-2019 | **2. REPORT TYPE**<br>Contractor Report | **3. DATES COVERED** *(From - To)* |

| | |
|---|---|
| **4. TITLE AND SUBTITLE**<br>Safe and Optimal Techniques Enabling Recovery, Integrity, and Assurance | **5a. CONTRACT NUMBER** |
| | **5b. GRANT NUMBER** |
| | **5c. PROGRAM ELEMENT NUMBER** |
| **6. AUTHOR(S)**<br>Siu, Kit Y.; Herencia-Zapana, Heber; Manolios, Panagiots;<br>Noorman, Michael | **5d. PROJECT NUMBER** |
| | **5e. TASK NUMBER**<br>NNL15AA02C |
| | **5f. WORK UNIT NUMBER**<br>340428.02.10.07.01 |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**<br>NASA Langley Research Center<br>Hampton, Virginia 23681-2199 | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**<br>National Aeronautics and Space Administration<br>Washington, DC 20546-0001 | **10. SPONSOR/MONITOR'S ACRONYM(S)**<br>NASA |
| | **11. SPONSOR/MONITOR'S REPORT NUMBER(S)**<br>NASA/CR-2019-220283 |

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Unclassified-
Subject Category 62
Availability: NASA STI Program (757) 864-9658

**13. SUPPLEMENTARY NOTES**
Langley Technical Monitor: Wilfredo Torres-Pomales

**14. ABSTRACT** There is a trend in the aviation industry to go from federated to integrated computing systems. Combining a number of traditional stand-alone federated systems into an integrated common platform (called Integrated Modular Avionics, IMA) has the benefit of increased power efficiency, reduced support hardware, and reduced cabling. However, changing from federated to integrated has a significant impact on the system architecture and hence the process of how avionic systems are to be analyzed. Traditional approaches to safety analysis become inefficient when functional boundaries can no longer be assumed for failure independence and fault isolation. In this report, we describe a tool that we developed to accelerate the safety engineer's ability to perform safety analysis of IMA systems through modeling, as well as optimize the system engineer's ability to develop a system through architecture synthesis. This work was the result of a three-year research effort called SOTERIA (Safe and Optimal Techniques Enabling Recovery, Integrity, and Assurance). We developed a compositional modeling language that supports rapid development, modification, and evaluation of architectures. The modeling language is structured such that the end-user defines a library of components with information on component reliability, connectivity, and fault propagation logic. The system model is built by instantiating the components from the library, connecting the components, and identifying the top-level faults of interest. Our tool is compositional in that the end-user only needs to define safety aspects at the component level. The tool takes the model and automatically synthesizes both the qualitative and quantitative safety analyses. We go further by allowing users to describe system information such as components to use in an architecture and their connection compatibility and automatically synthesize an architecture that meets the top-level probability target adhering to end-user specified constraints. This capability allows users to rapidly explore a design space.

**15. SUBJECT TERMS**
Analysis; Architecture; Failure; Fault; Modeling; Safety; Synthesis

| | | | | | |
|---|---|---|---|---|---|
| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON**<br>STI Help Desk (email: help@sti.nasa.gov) |
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | | | **19b. TELEPHONE NUMBER** ***(Include area code)*** |
| U | U | U | UU | 136 | (757) 864-9658 |