A Model-Based Framework for Analyzing the Safety of System Architectures

Panagiotis Manolios, PhD, Northeastern University

Kit Siu, GE Global Research

Michael Noorman, GE Aviation Systems

Hongwei Liao, PhD, Netflix

Key Words: fault trees, model-based design, safety analysis, safety-critical systems

SUMMARY & CONCLUSIONS

We introduce a compositional, model-based framework for modeling, visualizing and analyzing the safety of system architectures for safety-critical cyber-physical systems. Our work provides a unified, end-to-end, framework that encompasses high-level models, fault trees and qualitative and quantitative safety analyses in one semantically coherent framework. Our framework enables the rapid development, modification and evaluation of architectures for complex systems.

Our framework includes a modeling language for defining libraries of component models that include information on component reliability, connectivity and fault propagation. System architectures consist of a sequence of component instantiations, component connections, and the identification of top-level faults. Our framework includes algorithms for automatically synthesizing and reducing fault trees from architectures and library models. The generated fault trees are then automatically analyzed to determine cutsets and the probability of top-level faults. Finally, our framework includes visualization algorithms that depict fault trees and architectures at various levels of abstraction. We provide a case study of a model inspired by the Boeing 777 IMA architecture.

The framework is compositional because safety engineers only need to define reliability and fault propagation aspects at the component level. This is in contrast with current methods used in the field of avionics, where safety engineers directly construct system-level fault trees. Defining such fault trees requires significant expertise, time and care. Small changes to architectures can result in significant changes to fault trees. All of this makes analyzing a collection of architectures error-prone and prohibitive both in terms of time and money. We developed an open source tool that implements our framework, and provide an experimental evaluation consisting of the modeling and analysis of a collection of architectures. Our model-based framework provides a new paradigm, allowing significant automation in the area of safety analysis of architectures for complex avionics systems.

1 INTRODUCTION

One measure of the safety capability of a system is the level of risk that failures result in identified hazards, or undesired events, associated with the functions performed by the system. Examples include hazards that impact availability (loss of function) and integrity (erroneous, misleading or inadvertent function). Fault Tree Analysis (FTA) is the most common method for determining the combinations of failures that result in hazards, as well as the probability of such hazards occurring [1].

The current practice is that FTA is used at multiple stages within the development of safety critical systems. It is used during the early concept and initial architecture development phases to help drive architecture decisions. After the design details are finalized, FTA is used as a means of verifying compliance to qualitative and quantitative safety objectives. FTA is used as both a means to understanding the combinations of failures that lead to a given undesired event and a means of documenting the safety analysis.

Fault trees, though powerful, are hard to manually construct. According to [2], "The manual construction of fault trees relies on the ability of the safety engineer to understand and to foresee the system behavior … it is a time consuming and error-prone activity." Safety engineers also have to update fault trees and analyses as designs are refined and fleshed out at increasing layers of detail. This challenge is one of the driving forces behind our work.

We introduce a framework consisting of modeling, analysis and visualization capabilities that has been implemented and is available as an open-source tool, to be distributed by NASA, and is based on Inez, an extension of OCaml [3]. Our framework contributes to the area of Model Based Safety Analysis (MBSA).

1.1 Related Work

MBSA tools can be partitioned into two classes. The first class includes tools that require engineers to manually construct fault trees. These tools then perform various safety analyses, including the generation of minimal cutsets and quantitative analyses using the fault trees provided as input. Examples of such tools are OpenFTA and Windchill FTA. The second class includes tools that synthesize fault trees from models. Examples include HiP-HOPS, AltaRica, AADL with the Error Annex, as well as our framework.

HiP-HOPS [4] is an add-on tool that allows one to annotate Simulink or Sim-X models with reliability annotations, which are used to automatically generate fault trees and FMEAs (Failure Mode Effect Analyses). Minimal cutsets are generated from fault trees.

AltaRica [5-7] is a high-level modeling language for specifying the behavior of systems when faults occur. It includes a fault tree generator that automatically generates a static fault tree and uses a model checker to reason over dynamic properties of the system expressed in Linear Temporal Logic (LTL). AltaRica rejects models with loops even if there are no cycles in the failure propagation.

AADL is a modeling language for describing the structure of a system as an assembly of software components mapped onto execution platforms. The Error Annex EMV2 [8] provides support for specifying error models by adding to the AADL model error type, error propagations, composite error behavior, and component error behavior. EMV2 essentially requires users to provide a fault tree, as it requires a composite error behavior that specifies "the logic in a fault tree" [9].

The tools mentioned above are developed for very expressive modeling languages, e.g., Simulink. This makes it infeasible for such tools to fully support these modeling languages. For both AltaRica and AADL, the currently available tools have limitations and their own syntax and semantics, e.g., fault-tree compilers for AltaRica do not support delays. This leads to semantic inconsistencies that make it difficult for safety engineers to have confidence in the analyses produced by these tools. Instead, they tend to just directly define fault trees. This semantic gap was our motivation in defining our framework, since we started this project by first evaluating existing approaches.

1.2 Contributions

Our first contribution is the design of the framework. We introduce a unified framework that encompasses high-level models, fault trees and qualitative and quantitative safety analyses in one semantically coherent framework. There is no other framework we are aware of that provides this capability. For example, HiP-HOPS depends on Simulink and there is no documented claim that HiP-HOPS preserves the semantics of Simulink models. In fact, Mathworks does not publish the semantics and can change them as they see fit. This makes it unclear what the analyses mean with respect to the Simulink models.

Our second contribution is the design of the modeling language, introduced in Section 2. In order to have a semantically coherent framework, our modeling language is designed to provide a minimal set of core capabilities for modeling architectures at the level required to perform safety analyses. In contrast, current MBSA tools such as AltaRica require one to define a behavioral model, which is at a lower-level of abstraction than is needed for safety analysis. Our language is compositional, which allows one to define reusable libraries of components that provide consistency within and between the projects of an organization. It includes a fault-propagation language that supports defined equations, loops, defining multiple faults, defining faults in terms of other faults, etc. The language is an extension of OCaml, which allows us to support parameterized component models via component generators. The language is designed to only require information relevant to safety analysis, e.g., detailed behavioral models of components are not required. This allows for rapid development and prototyping of architectures in early stages of the design.

Our third contribution is the fault-tree synthesis algorithm for our modeling language, described in Section 3. While there exist fault-tree synthesis algorithms, the collection of language features and capabilities we allow requires a custom fault-tree synthesis algorithm. For example, dealing with loops, identifying when loops do not uniquely define fault propagation behavior, support for faults defined in terms of other faults and so on require a custom fault tree synthesis algorithm. Our algorithm is part of a unified framework, ensuring a semantic connection with the user-defined libraries used to model architectures. We have found that beyond a certain level of complexity, human-generated fault trees tend to be incomplete. While such incompleteness usually does not lead to significant differences in the probability of top-level faults, the possibility of a significant discrepancy exists. Our approach is guaranteed to not have such problems. The synthesis algorithm enables us to easily update all relevant fault trees after component definitions are updated and guarantees that components are handled in a complete, uniform way across an organization.

Our fourth contribution is the fault-tree reduction algorithm, described in Section 3. A practical problem with complete fault trees is that they tend to be more complex than fault trees humans generate, making them large and unwieldy. This makes it hard for safety engineers to understand and approve such trees, even if visualizations are used. Our fault tree reduction engine addresses this problem by using symbolic tree manipulation methods to generate equivalent, reduced fault trees. Experiments show that we often generate shallower fault trees with far fewer nodes, which are then easier for experts to understand and evaluate. In fact, these trees can be even simpler than trees generated by experts. Existing tools do not provide this capability. Many of them can generate BDDs and cutsets, but there is no guarantee that these transformations will simplify the tree (reduce the size of the formulas). In our case, we are guaranteed that the depth and size of the formula will not increase.

Our fifth contribution is the visualization algorithms of our framework, described in Section 2. We identify several useful views of architectures and automatically generate visualizations of these views.

Our sixth contribution is an open source tool, to be distributed by NASA. The tool provides evidence that the framework works and is usable by safety experts.

Our final contribution is our experimental evaluation of the framework, using a case study described in Section 4.

An important aspect of our framework is that it is compositional. That is, safety engineers only need to define reliability and fault propagation aspects at the component level. Instead of defining fault trees directly, one only defines how faults propagate through individual components. This only requires understanding one component at a time. Once this information is provided, it can be reused by any other safety engineer on any other project using that component, i.e.,

we wind up with a library of components that can be shared throughout an organization, thereby allowing for consistency, reuse and rapid architectural development. In contrast, most existing methods require safety engineers to define this information at the system-level, which requires significantly more effort and expertise. Our framework is also robust: making small changes to architectures is easy. Such changes can result in significant changes to fault trees, which makes the use of existing system-level methods costly and error-prone. Our model-based framework provides a new paradigm, allowing significant automation in the area of safety analysis of architectures for complex avionics systems.

2 MODELING

Our modeling language is designed to provide a minimal set of capabilities for modeling architectures at the level required to perform safety analyses. The modeling language has evolved as we used it to model a variety of architectures, some of which were quite complex. For example, the language allows one to model feedback loops; while such loops seem to be rare in avionics applications, they sometimes exist and they can be modeled and analyzed in our current framework.

Our modeling language allows us to define components, libraries and architectures. A *component* consists of a name, a list of fault types supported, a list of inputs, a list of basic events and their characteristics, a list of outputs and formulas characterizing the propagation of faults through the component. A *library* is a collection of components. An *architecture* or a *model* is a set of component instantiations, which allow us to override basic event information, along with connection information and the identification of a top-level fault.

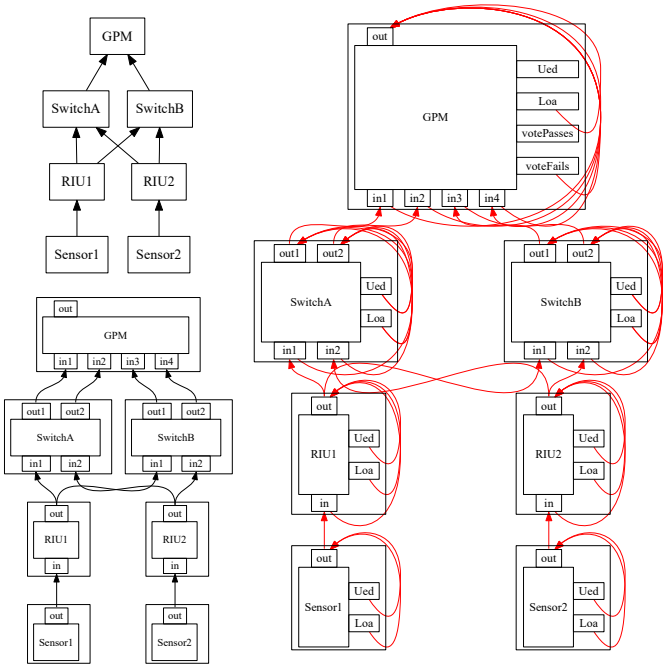

Figure 1 Physical (upper left), functional (lower left), and fault propagation (right) architectures

We will use the running example architecture shown in *Figure 1* to highlight the modeling language. The architectural views were generated by our framework's visualization capability. The architecture consists of two sensors that generate analog readings that are sent to the RIUs (Remote Interface Units) that digitize the data and send it on the network along channels A and B, using Switches A and B. RIU*i* only accepts input from Sensor*i*, but both switches accept input from both RIUs. The switches send the data to the GPM (General Processing Module). A visualization of the architecture at this high-level of abstraction is provided by the *physical architecture*, a graph whose nodes are component instances and whose edges correspond to one or more connections between components. To describe the workings of the GPM, it helps to consider the more detailed *functional architecture*, a graph whose nodes are ports of components and whose edges correspond to connections between ports. The GPM includes a redundancy management unit, which selects a sensor reading from Sensor1 and Sensor2. The Sensor1 data comes in through ports in1 and in3, whereas the Sensor2 data comes in through ports in2 and in4. For our purposes, we will assume that this selection is nondeterministic. Next, the GPM has a voting unit, which checks that the data sources selected for sensor readings are equivalent.

Our modeling language allows us to formalize how various faults propagate through components. In the GPM, a UED (Undetected Erroneous Data) fault occurs if the data are equal, but erroneous. For that to happen, at least one of the inputs from Sensor1 to the GPM propagated a UED fault AND at least one of the inputs from Sensor2 to the GPM propagated a UED fault AND the vote passes, i.e., it does not report a problem when there is one because the errors cancel each other out. An LOA (Loss of Availability) error occurs when both copies of at least one sensor propagate an LOA fault: then one of the sensor readings is unavailable. But, what if both are available, but erroneous? Then, due to the voting, we have an LOA fault (we caught a problem). But, that's similar to what UED is checking, so this is an example of where we want the propagation logic for one fault (LOA) to refer to the propagation logic for another (UED).

An example of how we specify a component is shown on the top half of *Figure 2*. The basic events include internal faults that lead to UED and GPM faults, as well as events corresponding to the voting logic. The basic event information required corresponds to the lambda and tau values of the basic event, i.e., the failure rate and exposure time. Boolean formulas are used to define how UED and LOA faults propagate through the component. These formulas can include references to faults that propagate through the inputs, internal faults and defined equations. Defined equations make fault propagation logic easier to understand (just like functions make programs easier to understand) and give us the ability to define the propagation of a fault in terms of formulas defining the propagation of another fault. The utility of defined equations became apparent when we used the framework to model large, realistic examples. There should be a formula for

```
{name         = "GPM";
 faults       = ["ued"; "loa"];
 input_flows  = ["in1"; "in2"; "in3"; "in4"];
 basic_events = ["Ued"; "Loa"; "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["Ued"];F["uedNotCaughtByVote"]]);
                 (["uedNotCaughtByVote"],
                  And[F["2uedFaults"]; F["votePasses"]]);
                 (["2uedFaults"],
                  And[Or[F["in1"; "ued"]; F["in3"; "ued"]];
                      Or[F["in2"; "ued"]; F["in4"; "ued"]]]);
                 (["out"; "loa"],
                  Or[F["Loa"];
                     And[F["in1"; "loa"]; F["in3"; "loa"]];
                     And[F["in2"; "loa"]; F["in4"; "loa"]];
                     And[F["2uedFaults"]; F["voteFails"]]])];};
```

```
instances =
  [makeInstance "Sensor1" "Sensor" ();
   makeInstance "Sensor2" "Sensor" ();
   makeInstance "RIU1" "RIU" ();
   makeInstance "RIU2" "RIU" ();
   makeInstance "SwitchA" "Switch" ();
   makeInstance "SwitchB" "Switch" ();
   makeInstance "GPM" "GPM" ();
  ];
connections =
  [ (("RIU1",  "in"),  ("Sensor1", "out"));
    (("RIU2",  "in"),  ("Sensor2", "out"));
    (("SwitchA", "in1"), ("RIU1", "out"));
    (("SwitchA", "in2"), ("RIU2", "out"));
    (("SwitchB", "in1"), ("RIU1", "out"));
    (("SwitchB", "in2"), ("RIU2", "out"));
    (("GPM",  "in1"), ("SwitchA", "out1"));
    (("GPM",  "in2"), ("SwitchA", "out2"));
    (("GPM",  "in3"), ("SwitchB", "out1"));
    (("GPM",  "in4"), ("SwitchB", "out2"));];
top_fault = ("GPM", F["out"; "ued"]);
```

Figure 2 Component model (top) and library (bottom)

each fault-type/output-flow combination and fault formulas can refer to the formulas for other faults. The types of formulas allowed are positive Boolean formulas because for safety analysis, a fault not occurring never leads to another fault occurring, i.e., the propagation functions are positive. The basic operators allowed include references to *atomic* variables (such as faults propagated through inputs, basic events and references to defined equations), arbitrary-arity conjunctions, arbitrary-arity disjunctions and an N-of construct that corresponds to *n* or more of a list of formulas being true.

A collection of components corresponds to a library, as shown in the bottom half of *Figure 2*. An architecture definition is a collection of component instantiations, along with connection information and the identification of a top-level fault, which can be any formula (but is just an atomic variable in our example). Our modeling language allows one to override the default lambda and tau values of component instances (not shown). This is often required to account for the operating environment of a component, e.g., the failure rate of a component may vary depending on whether it is in a pressurized space or not.

Furthermore, our modeling language allows us to define parameterized component models. Parameters can be used to specify the number of inputs and outputs of a component, the source selection mechanisms used by a component, the voting mechanisms used, and so on.

The fault propagation view of the architecture, as shown in *Figure 1* extends the functional architecture with nodes corresponding to basic events and shows what parts of the architecture can affect the probability of the top-level fault. If an edge (corresponding to the propagation of a fault from one component to another) can affect the top-level fault, it is colored red. If a basic event can affect the top-level fault, then a (set of) red edge(s) is added from that event to the output(s) along which the basic event propagates.

3 FAULT TREE SYNTHESIS, REDUCTION AND ANALYSIS

Once architectures are modeled in our framework, it is possible to algorithmically synthesize fault trees, the topic of this section. From the synthesized fault trees, we can use standard algorithms to perform qualitative and quantitative analyses to determine the safety of the modeled architecture. Therefore, our framework allows us to go from models to analyses with a push of a button. In contrast, current practice in avionics requires safety engineers to construct fault-trees by hand. Notice that a fault tree is required per fault, so if there are 30 faults of interest and 5 architectures to consider, then 150 fault trees must be generated. This makes it difficult to explore multiple architectures and it makes it easy to introduce errors when fault trees must be updated to reflect architectural changes, which often undergo numerous changes in the course of an industrial project.

We describe the fault tree synthesis algorithm informally, due to space limitations. The idea is simply to expand out the equations defining the top-level fault under consideration starting from the component instance that has the fault. If the fault involves more than one component, then we define a dummy component that is connected to all the required components. The algorithm expands out the fault propagation logic associated with component outputs. Recall that there may be defined equations, in which case these are expanded and simplified until we arrive at a formula in terms of the basic events of the component and the inputs of the component (really input/fault pairs). The architecture is consulted to determine what component instance outputs are connected to the identified inputs and the process repeats, i.e., we expand out the appropriate fault propagation logic associated with the newly discovered component instances as before. If there are no cycles, we eventually have a formula in terms of just basic events, potentially with many events appearing multiple times. For our running example in *Figure 1*, the direct fault tree generated for the top-level fault LOA of GPM, is shown in *Figure 3*. What we have found is that such trees can be significantly simplified, which makes it easier to use them to understand system behavior. Therefore, our framework includes a collection of formula transformations that when used (as is the default case) generate the reduced fault tree shown in *Figure 4*. Notice that this fault tree is much simpler than the direct fault tree, e.g., the number of nodes and operators has been reduced significantly. The two trees are semantically equivalent, but the reduced fault tree is much easier to understand and it does not have any repeated events!

From these fault trees, we generate cutsets, which correspond to the minimal DNF formula that is equivalent to the fault trees. Since the formulas are positive, there is a unique minimal DNF formula that can be generated using existing algorithms [1]. Our framework uses symbolic

manipulation algorithms for both reducing fault trees and generating cutsets. An example of the simplifications used includes the *lift* transformation, which is essentially distributivity in reverse.

$$\Lambda_{i=1}^{k}[g, (\vee f, f_1^i \dots f_{ni}^i)] = \Lambda[g, \vee f(\Lambda_{i=1}^{k}(\vee f_1^i \dots f_{ni}^i))]$$

Now, this transformation can increase the depth of the tree and even the number of Boolean operators, but the number of variable occurrences decreases. Another example of a transformation we apply is that we replace the second occurrence of g in the formula below with *false*.

$$\vee[\dots, g, \dots, \wedge(\dots, \vee(\dots, g, \dots), \dots), \dots]$$

We also apply the duals of the above transformations. These transformations are used in conjunction with constant propagation and transformations for flattening, simplifying and sorting formulas. In addition, where appropriate, transformations are applied until a fixpoint is reached. As the lift transformation highlights, the algorithms have to be careful to avoid loops, so depending on the context in which they are used some transformation are not applied unless they meet context-sensitive criteria. The details can be found in a planned journal version of this paper.

The cutsets generated are shown in *Figure 5*. Notice that while the cutsets are minimal DNF formulas, they are not minimal Boolean formulas, as a comparison *Figure 4* between and *Figure 5* shows. Notice also that the cutsets include repeated events, whereas the reduced fault tree does not.

The cutsets are very useful because each term of the top-level disjunction corresponds to a minimal set of basic events that together lead to the top-level fault under consideration. For example, we can determine that for LOA faults, there are 5 basic events that correspond to single points of failure. That this is the case is not apparent from the direct fault tree in Figure 2, but it is from the reduced fault tree in Figure 3. However only the cutsets are guaranteed to provide this information.

From the cutsets, we can use standard techniques to determine the probability that a top-level fault occurs. The probability of a failure for a basic event (from reliability theory) is $1 - e^{-\lambda T}$. To compute the probability using cutsets

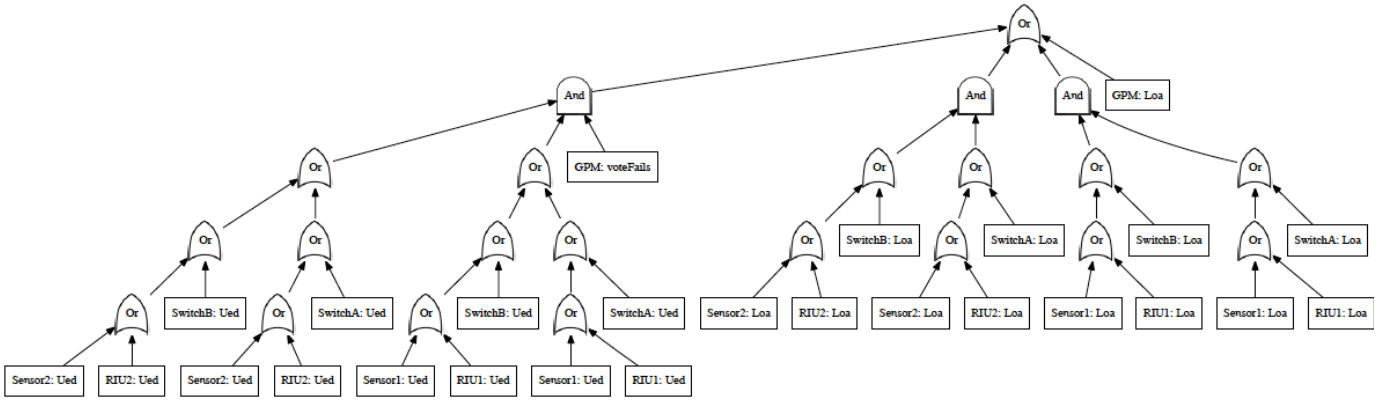

Figure 3 Direct Fault Tree example

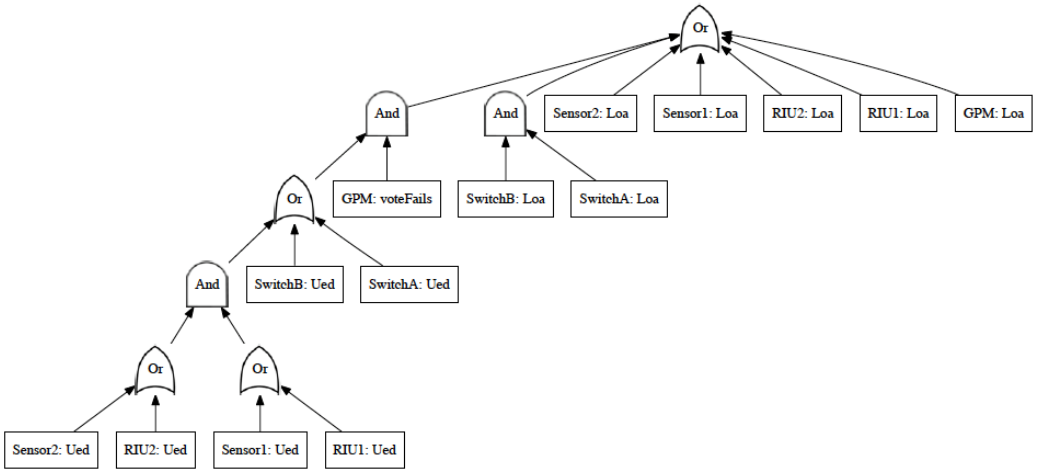

Figure 4 Reduced Fault Tree example

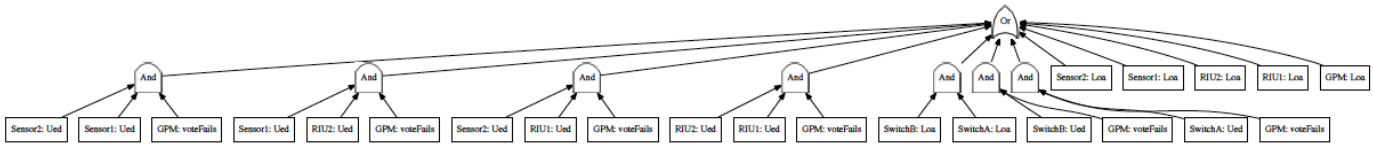

Figure 5 Cutsets

requires the use of the inclusion-exclusion principle. Note that we assume that basic events are independent (as is standard in safety analysis and this is something that is validated via other means). The reason we need inclusion/exclusion is that basic events can be repeated. However, if we have a formula with no repeated events (as is the case with our reduced fault tree above), then the probability computations are straightforward. There are various specialized techniques for determining the probability of a top-level failure efficiently, and these techniques are outside the scope of our work. Our framework also generates importance metrics, which order cutsets by their contribution to the top-level fault. This is very useful in understanding what parts of an architecture are primarily responsible for high probabilities, which helps safety engineers design new architectures that address these issues. The next section provides a detailed example.

3.1 Handling Loops

The fault tree generation algorithm includes one more complication that we now describe. In certain architectures, there are cyclic dependencies between faults. The NASA Fault Tree Handbook [1], version 1.1, includes a new section describing such feedback loops. The handbook describes how in the space shuttle, the orbiter sends a control signal to the main engine, which provides the feedback signal, so the failure of the orbiter depends on the failure of the main engine, which depends on the failure of the orbiter, …. Our fault tree synthesis algorithm supports such loops. First, loops are detected when we notice a cycle in the expanded fault propagation definition. We note that one can legitimately expand out the definition of some fault more than once without there being a loop, so we have to take context into account, by only looking for loops along the path of ancestors of the fault tree node being generated. Once loops are found, they have to be broken in a way that makes sense, e.g., $a \vee a \vee a \vee \cdots$ should be set to a, which is equivalent to replacing all but the first occurrence of a with *false*. However, in the formula $a \wedge a \wedge a \wedge \cdots$ we should replace all but the first occurrence of a with *true*. Our algorithm breaks loops only in the context of conjunctions and disjunctions that include at least one basic event and replaces repeated events with the appropriate identity for the operator. If there are no basic events, then there are a number of potential solutions to the equations and that indicates an error, which we report.

4 CASE STUDY

We present a case study that is inspired by the Boeing 777 IMA architecture. *Figure* 6 shows the functional architecture. The system contains three types of input components: inertial reference units (IRU), multi-purpose control display units (MCDU), and distance measuring equipment (DME). The data generated from the input components are first transmitted to input/output modules (IOM) and then transmitted to flight management computers (FMC) via network buses. Consider LOA (Loss of Availability) faults. Since the probability that network buses are responsible for LOA faults is much smaller than that of other components in the system, network buses are omitted for the sake of simplicity. After FMCs perform their computations, the results are first transmitted to IOMs via network buses and then transmitted to symbol generators (SG) for display. The system contains two types of display components: primary flight displays (PFD) and navigation displays (ND). The top-level fault of interest is the LOA of PFD1. In this architecture, input components contain two instances for dual redundancy, and the communication over network buses is supported by dual channel redundancy (channels A and B).

Our tool automatically synthesizes the fault tree of the system and generates the reduced fault tree and cutsets, along with visualizations. The top-level probability of failure is 3.000E-6. The fault probability and the importance metric indicate that the cutset IOM1, IOM2, and IOM3 account for more than 99.99% of the top-level probability of failure, which can be explained by the following observations. It can be seen that IOM1 introduces a single point of failure to the two IRU components. Loss of IOM1 will lead to loss of *both* IRU components, which essentially removes the dual redundancy of IRUs. Similarly, IOM2 and IOM3 introduce single points of failures for MCDUs and DMEs, respectively. To improve the system's safety performance, we remove these single points of failure in an updated system architecture where the input components are shuffled so that IOM1, IOM2, and IOM3 take different types of input components. The updated architecture's top-level probability of failure is 2.160E-10, which is a significant improvement over the original system. In the new system, the only dominant cutset is the basic event of PFD1, which accounts for 92.59% of the top-level probability which is unavoidable because PFD1 is the sink node of the fault data flow under consideration.

The above case study shows that our framework, with automated fault tree synthesis and analysis capabilities, can help a safety engineer to quickly identify, correct and fix

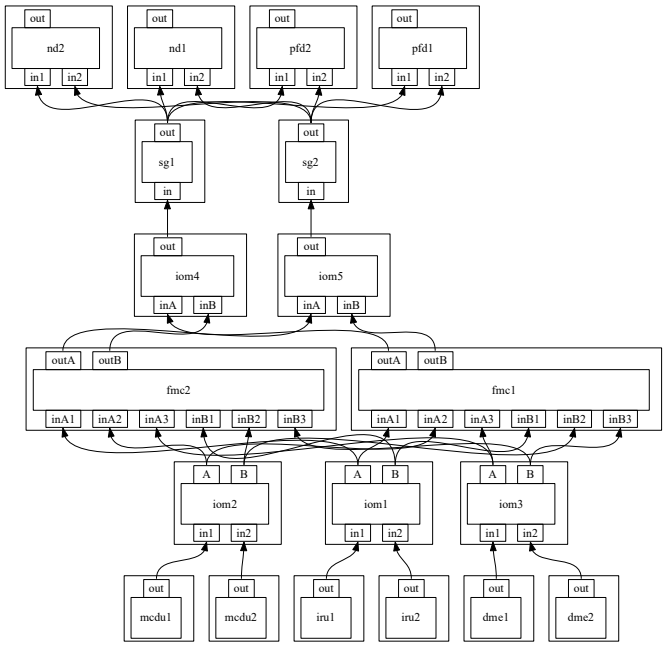

Figure 6 Boeing 777 IMA architecture

architectural problems. To go from the original architecture to the new architecture, we only need to modify the connections associated with the inputs of the IOMs. The rest of the system model and component library remains the same, which demonstrates the advantage of our component-based approach. Our framework synthesizes the fault tree, generates the cutsets and computes the top-level probability of failure for the new architecture automatically, removing the fault tree construction burden from the safety engineer, allowing her to focus on architecture-level designs.

5 ACKNOWLEDGEMENTS

This research was the result of NNL15AA02C, a cost-sharing contract between NASA and GE. Tool distribution is through NASA Langley Research Center. The authors would like to thank Camila Rodriguez and Heber Herencia-zapana for their support on the examples used in this paper.

REFERENCES

1. NASA, *Fault Tree Handbook with Aerospace Applications*, 2002.
2. R. Banach and M. Bozzano, "Retrenchment, and the generation of fault trees for static, dynamic and cyclic systems," *International Conference on Computer Safety, Reliability, and Security SAFECOMP*, pp 127-141, 2006.
3. P. Manolios, J. Pais and V. Papavasileiou, "The Inez mathematical programming modulo theories framework," *Computer Aided Verification*, vol. 9207, no. LNCS, pp. 53-69, 2015.
4. Y. Papadopoulos, *HiP-HOPS Automated Fault Tree, FMEA and Optimisation tool - User Manual*, University of Hull.
5. A. Batteux and T. Prosvirnova, *AltaRica 3.0 Language Specification*.
6. R. Bernard, J. Aubert, P. Bieber and S. M. C. Merlini, "Experiments in model based safety analysis: flight controls," *IFAC Proceedings Volumes*, 40(6): 43-48, 2007.
7. P. Bieber, C. Bougnol, C. Castel, J. Heckmann, C. Kehren, S. Metge and C. Sequin, "Safety assessment with AltaRica - lessons learnt based on two aircraft system studies," *18th IFIP World Computer Congress*, pp 26, 2004.
8. P. Feller, J. Hudak, J. Delange and D. Gluch, *Architecture Fault Modeling and Analysis with the Error Model Annex,* version 2, Software Engineering Institute, Carnegie Mellon University, 2016.
9. J. Delange and P. Feiler, "Architecture fault modeling with the AADL error-model annex," *EUROMICRO Conference on Software Engineering and Advanced Applications*, pp 361-368, 2014.

BIOGRAPHIES

Panagiotis Manolios, PhD
Northeastern University
360 Huntington Avenue
Boston, MA, 02115, USA

e-mail: pete@ccs.neu.edu

Panagiotis Manolios is a Professor in the College of Computer and Information Science at Northeastern University. He earned his doctoral degree in Computer Science at the University of Texas, Austin in 2001. He is a widely published expert in formal verification and validation. He has served as an Associate Editor of ACM Transactions on Design Automation of Electronic Systems (TODAES), as a member of the FMCAD, ACL2 and ITP Steering Committees, is a member of the IFIP working group 1.9/2.15 on Verified Software. He is a member of ACM, IEEE and Sigma Xi.

Kit Siu
GE Global Research
One Research Circle
Niskayuna, NY 12309, USA

e-mail: siu@ge.com

Kit Siu is a Senior Engineer with 18 years of experience. She received her masters from Rensselaer Polytechnic Institute and bachelors from Columbia University. She works at GE Global Research developing validation and verification technology for software. She is a co-PI on one of DARPA's Cyber Assured Systems Engineering (CASE) programs and was the PI on SOTERIA (Safe & Optimal Techniques Enabling Recovery, Integrity, and Assurance), a NASA funded research for analyzing safety of IMA architectures.

Michael Noorman
GE Aviation Systems
3290 Patterson Avenue SE
Grand Rapids, MI, 49512, USA

e-mail: michael.noorman@ge.com

Mike Noorman is a Principal Engineer and has been with GE Aviation since 2005. He received his bachelors from Michigan State University. He has been the lead Safety Engineer on various GE Aviation projects. He provides technical oversight for safety activities across many GE Aviation products, develops internal safety processes, mentors other safety engineers, and supports multiple development programs. He is an active member of the SAE S-18 committee working on updates to ARP4761, Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment.

Hongwei Liao, PhD
Netflix, Inc.
5808 Sunset Boulevard
Los Angeles, CA, 90028, USA

e-mail: hliao@netflix.com

Hongwei Liao is currently a Senior Data Scientist at Netflix. He received his Ph.D. and Masters from University of Michigan. Prior to his current role, Dr. Liao worked at Aviall - A Boeing Company and GE Global Research.