Architectural Modeling and Analysis for Safety Engineering

A THESIS TOPIC PROPOSAL
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Danielle Stewart

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Dec, 2019

Acknowledgements

When I first began my journey into the great land of the PhD, my background was quite different than safety analysis and it took a number of months to feel even slightly comfortable in this new landscape. I could not have done that without the patient and kind help of Michael Whalen and Darren Cofer. A huge thank you to both of you for this support. You provided an environment in which it was fun to ask questions, easy to learn, and exciting to explore new places. I am forever grateful.

I also want to thank Mats Hiemdahl for your willingness to take me on as your student when the time came for Mike to move on. Despite your busy schedule, you always find time for my questions and ideas. Thank you.

And lastly, thank you to Antonia Zhai and John Sartori for reading these words, having insights and questions for me to ponder, and taking time to support my studies. Thanks for being on my committee!

Abstract

Model-based development tools are increasingly being used for system-level development of safety-critical systems. Architectural and behavioral models provide important information that can be leveraged to improve the system safety analysis process. Model-based design artifacts produced in early stage development activities can be used to perform system safety analysis, reducing costs and providing accurate results throughout the system life-cycle.

As critical systems become more dependent on software components, analysis regarding fault propagation through these software components becomes more important. The methods used to perform these analyses require understandability from the side of the analyst, scalability in terms of system size, and mathematical correctness in order to provide sufficient proof that a system is safe. Determination of the events that can cause failures to propagate through a system as well as the effects of these propagations can be a time consuming and error prone process. In this research, we introduce a safety analysis tool extension to the AADL modeling language, describe a technique for determining failure events with the use of Inductive Validity Cores (*IVC*), and show how an analyst can use these methods to produce compositionally derived artifacts that encode pertinant system safety information.

Contents

Acknowledgements **i**

Abstract **ii**

List of Figures **v**

1 Introduction **1**

2 Background **4**

2.1 The Safety Assessment Process and Required Artifacts 4

2.1.1 The Traditional Safety Assessment Process 6

2.1.2 Model-Based Safety Assessment Process Supported by Formal Methods 6

2.2 Related Work 7

2.3 Tools and Modeling Language 10

2.3.1 Architecture Analysis and Design Language 10

2.3.2 Assume Guarantee Reasoning Environment 10

2.3.3 Safety Annex for AADL 11

2.4 Definitions 11

3 Proposed Approach **14**

3.1 The Safety Annex and Fault Modeling 14

3.1.1 Implementation 14

3.1.2 The Sensor System 16

3.1.3 The Safety Annex for the Sensor System 16

3.1.4 Modeling Dependent Faults 18

3.2 Verification Capabilities in the Face of Component Failures 18

3.2.1 Maximum Fault Analysis 19

3.2.2 Probabilistic Threshold Analysis 20

3.3 Generation of Minimal Cut Sets from *MIVCs* 21

3.3.1 Illustrative Example 22

3.4 Compositional Probabilistic Computations 26

3.5 Evaluation of Results 28

4 Conclusion **30**

References **32**

List of Figures

2.1 A simple fault tree 5
2.2 Use of the Shared System/Safety Model in the ARP4754A Safety Assessment Process 7
3.1 Safety Annex Plug-in Architecture 15
3.2 Nominal `AGREE` Node and Extension with Faults 15
3.3 A Simple Sensor System 16
3.4 A Pressure Sensor and the `AGREE` and Safety Annexes 17
3.5 A Fault Node Definition 17
3.6 Max N Faults Analysis Statement 18
3.7 Probability Analysis Statement 19
3.8 `Lustre` Statement for Fault Count 19
3.9 Steps of the Transformation Process 22
3.10 Flat Fault Tree for the Sensor Example 24
3.11 Hierarchical Fault Tree for the Sensor Example 25
3.12 Sample System Contract Part I 27
3.13 Sample System Contract Part II 27

Chapter 1

Introduction

System safety analysis is crucial in the development life cycle of critical systems to ensure adequate safety as well as demonstrate compliance with applicable standards. A prerequisite for any safety analysis is a thorough understanding of the system architecture and the behavior of its components; safety engineers use this understanding to explore the system behavior to ensure safe operation, assess the effect of failures on the overall safety objectives, and construct the accompanying safety analysis artifacts [66, 67]. Developing adequate understanding, especially for software components, is a difficult and time consuming endeavor which has traditionally been performed manually. Model-based development is getting increased adoption in the critical systems domain [12, 37, 41, 45, 51]. In model-based development, the development efforts are centered on a model of the intended system behavior and various techniques, for example, formal verification, testing, test generation, execution and animation, etc. can be used to validate and verify the proposed system behavior. Given this increase in model-based development in critical systems, leveraging the resultant models in the safety analysis process and automating the generation of safety analysis artifacts, holds great promise in terms of accuracy and efficiency. Many of these artifacts are representation of the functionality of the system in the presence of faults and can be effectively used to find *minimal cut sets*, the minimal sets of faults that lead to a violation of a safety property. Some research groups have introduced automating the safety assessment process and have developed tools to support the automated generation of these artifacts [12, 16, 43]. Nevertheless, there are gaps in current capabilities we aim to address.

Many of the techniques developed require the development of models specific for the safety analysis task, that is, the techniques do not rely on extension of existing system models, but instead require new models that are separate entities [10, 15, 37, 42]. This requires extra manual labor and clear understanding of the system in order to create a separate fault model that accurately describes the system in question. As systems become more complex, it becomes difficult to ensure that the fault model developed for safety analysis conforms with the the model created for the development efforts.

The propagation of faults throughout the system can be handled in two main ways: one is to explicitly define the propagations and the other is to use a behavioral approach to the propagations. Given many possible faults in an industrial sized system, explicit propagation becomes complex and unwieldy. That being said, it is sometimes beneficial to also explicitly define fault propagations between components that are not logically connected, e.g., co-located components may fail together even though they are not logically connected, threads running on a processor will fail together if the processor fails, etc. Thus, it is beneficial to provide both options to an analyst.

Using a compositional approach—analyze the system components individually and combine the results—to fault modeling improves scalability compared to a monolithic approach—the components are all analyzed together. The compositional generation of the combinations of faults that can cause system failure is something that no other research has fully addressed and a scalable automatic generation of these artifacts is a major bottleneck in using automated approaches in industry. Given recent research in the area of formal verification, this has become theoretically possible.

Probabilistic evaluations of the faults and their overall impact on the system is a large part of the safety assessment process. Utilizing a compositional approach for the generation of minimal cut sets shows promise in compositional probabilistic computations as well.

In this proposal, we outline a novel way to utilize the verification information used during the system development process in order to automate the generation of both qualitative and quantitative safety analysis artifacts. Furthermore, we describe a Model Based Safety Analysis (MBSA) approach to critical system development in which the safety analysis is tied directly to the system model and the flexibility of the analysis provides various ways of capturing error propagation information, single points of failures, and minimal cut sets.

Our **long range goal** is to increase system safety through the support of a model-based safety assessment process backed by formal methods to help safety engineers with early detection of design issues and automation of the artifacts required for certification. The **objective of this proposal**, which is a logical step towards our goal, is to formally prove a relationship between formal verification results and safety analysis artifacts to allow for compositional generation of minimal cut sets and associated probabilities, define a modeling process that allows for flexible fault modeling in the MBSA approach, and finally to support these goals with relevant tools. We plan to accomplish the objectives of this proposal by pursuing the following aims:

Define and implement a Safety Annex to enable fault modeling in AADL. Research the modeling needs of a safety analyst and define a grammar that extends the Architecture Analysis and Design Language (AADL). Implement a tool that uses this grammar extension and a backend SMT model checker for compositional verification purposes.

Define compositional verification capabilities in the face of component failures. Define analysis procedures to allow an analyst to investigate system behaviour under a predetermined max number of faults or a max probability threshold for fault combinations, and develop and implement analysis procedures for these analyses.

Define compositional computation of minimal cut sets. Formally prove the relationship between verification results of compositional analysis and minimal cut sets, and implement the algorithms in the analysis tools supporting the Safety Annex.

Determine accurate compositional probabilistic computations for the top-level events. Define the computations, show that the computations are accurate, and implement the algorithms in the analysis tools supporting the Safety Annex.

Evaluate the results using representative models in AADL. Gather the safety analysis artifacts generated by these algorithms and discuss how they may be used within the safety assessment process.

The beginning phase of this dissertation research included research into the needs of safety analysts and resulted in the definition of the Safety Annex, the implementation of the Safety Annex parser in the OSATE tools, and the support for analysis of AADL models extended with the Safety Annex. The remaining steps of this research include completion of the theory behind the relationship between verification results and fault analysis artifacts and a full implementation of these algorithms in the tools supporting the Safety Annex.

This proposal is organized in three chapters. The background in Chapter 2 broadly discusses related work, the tools and modeling language used in this project, and some useful formal definitions. Chapter 3 describes the proposed approach and outlines the contributions of this dissertation. Lastly, the conclusion summarizes the approach of the project.

Chapter 2

Background

The background necessary for this research includes four main topics. A basic introduction to the safety assessment process is given along with definitions of important artifacts used in the process. Related work gives an overview of pertinent literature in this area of research. The specific modeling language and related tools are described, and lastly, relevant formal definitions are explained and provided for better understanding of the theoretical contributions of this dissertation.

2.1 The Safety Assessment Process and Required Artifacts

In order to approve an aircraft for flight through the FAA, one must show compliance with federal safety regulations. Industry has over the decades adopted standards published by the Society of Automotive Engineers (SAE); these include Aerospace Information Reports (AIR) and Aerospace Recommended Practices (ARP). These standards aim to provide an organized and acceptable approach for safety analysis that serves to meet the federal regulations and provides industry with a methodology for safety assessment [1,2]. Various artifacts are commonly used in the safety assessment process to show the behavior and functionality of the system in the presence of faults. Often, these artifacts are necessary for certification of the system. One artifact that is important to this research is a fault tree and its related minimal cut sets.

Fault Tree Analysis and Minimal Cut Sets

The use of fault trees are common in many safety assessment processes and the ability to generate the cut sets needed for the construction of the fault tree is a useful part of any safety analysis tool. The fault tree is a safety artifact commonly referenced in requirement protocol documents such as ARP4761, ARP4754, and AIR6110 [3,66,67].

A Fault Tree (FT) is a directed acyclic graph whose leaves model component failures and whose gates model failure propagation [65]. The system failure under examination is the root of the tree and is called the Top Level Event (TLE). The node types in a fault tree are *events* and *gates*. An event is an occurrence within the system, typically the failure of a subsystem down to an individual component. Events can be grouped into *basic events* which occur independently, and *intermediate events* which occur dependently and are caused by one or more other events [30]. These events model the failure of the system (or subsystem) under consideration. The gates represent how failures propagate through the system and how failures in subsystems can cause system wide failures. The two most common logic symbols used in an FT are the Boolean logic AND-gates and OR-gates. An AND-gate is used when the undesired top level event can only occur when all the lower conditions are true. The OR-gate is used when the undesired event can occur if any one or more of the next lower conditions is true. This is not a comprehensive list of gate types; others include voting, inhibit, or negation gates [65].

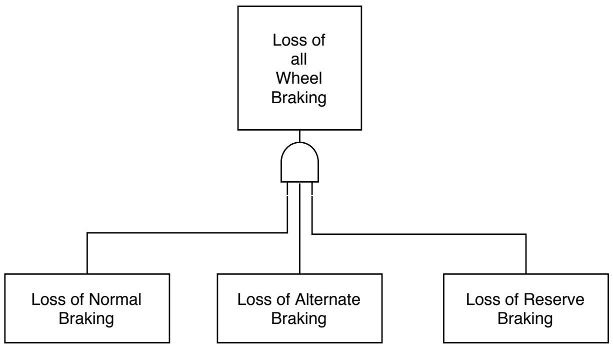


Figure 2.1: A simple fault tree

Figure 2.1 shows a simple example of a fault tree based on SAE ARP4761 [66]. In this example, the top level event corresponds to an aircraft losing all wheel braking. In order for this event to occur, all of the basic events must occur. This is seen through the use of the AND gate below the top level event. The gates in the fault tree describe how failures propagate through the system. Each gate has one output and one or more inputs. In Figure 2.1, the AND gate has three inputs and one output. The leaves of the tree represent the basic events of the system. In the case of this fault tree, these three events are also the Minimal Cut Sets (MinCutSets) for this top level event. A MinCutSet is the minimal set of basic events that must occur together in order to cause the TLE to occur. Generating and analyzing these MinCutSets is central to FTA and has been an active area of interest in the research community since fault trees were first described in Bell Labs in 1961 [30, 65].

There are two main types of fault tree analysis that we differentiate here as *qualitative* analysis and *quantitative* analysis. In qualitative analysis, the structure of the fault tree is considered and the MinCutSets are a way to indicate which combinations of component failures will cause the system to fail. On the other hand, in quantitative analysis, the probability of the TLE is calculated given the probability of occurrence of the basic events.

2.1.1 The Traditional Safety Assessment Process

The traditional safety assessment process at the system level is based on standards ARP4754A and ARP4761 [66, 67]. It starts with the System level Functional Hazard Assessment (SFHA) examining the functions of the system to identify potential functional failures and classifies the potential hazards associated with them.

The next step is the Preliminary System Safety Assessment (PSSA), updated throughout the system development process. A key element of the PSSA is a system level Fault Tree Analysis (FTA). The FTA is a deductive failure analysis to determine the causes of a specific undesired event in a top-down fashion. For an FTA, a safety analyst begins with a failure condition from the SFHA, and systematically examines the system design (e.g., signal flow diagrams provided by system engineers) to determine all credible faults and failure combinations that could cause the undesired event.

The lack of precise models of the system architecture and its failure modes often forces safety analysts to devote significant effort to gathering architectural details about the system behavior from multiple sources. Furthermore, this investigation typically stops at system level, leaving software function details largely unexplored.

Typically equipped with the domain knowledge about the system, but not detailed knowledge of how the software applications are designed, practicing safety engineers find it a time consuming and involved process to acquire the knowledge about the behavior of the software applications hosted in a system and its impact on the overall system behavior. Industry practitioners have come to realize the benefits of using models in the safety assessment process, and a revision of the ARP4761 to include Model Based Safety Analysis (MBSA) is under way.

2.1.2 Model-Based Safety Assessment Process Supported by Formal Methods

We propose a model-based safety assessment process backed by formal methods to help safety engineers with early detection of the design issues. This process uses a single unified model to support both system design and safety analysis. It is based on the following steps:

1. System engineers capture the critical information in a shared model: high-level hardware and software architecture, nominal behavior at the component level, and safety requirements at the system level.

2. System engineers use a model checker to verify that the safety requirements are satisfied by the nominal design model.

3. Safety engineers augment the nominal model with the component failure modes. In addition, safety engineers specify the fault hypothesis for the analysis which corresponds to how many simultaneous faults the system must be able to tolerate.

4. Safety engineers use a model checker to analyze if the safety requirements and fault tolerance objectives are satisfied by the design in the presence of faults. If the design does not tolerate the specified number of faults (or probability threshold of fault occurrence),

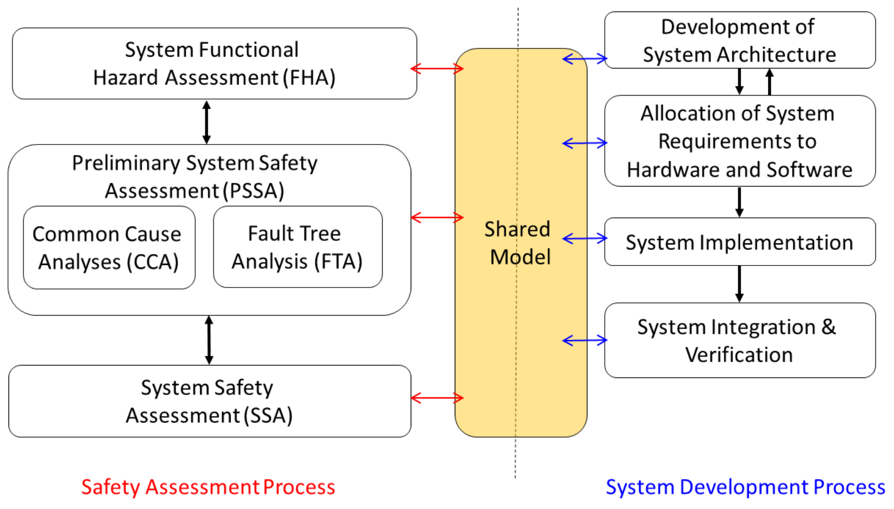

System Functional Hazard Assessment (FHA)
Preliminary System Safety Assessment (PSSA)
Common Cause Analyses (CCA)
Fault Tree Analysis (FTA)
System Safety Assessment (SSA)
Shared Model
Development of System Architecture
Allocation of System Requirements to Hardware and Software
System Implementation
System Integration & Verification
Safety Assessment Process
System Development Process

Figure 2.2: Use of the Shared System/Safety Model in the ARP4754A Safety Assessment Process

then the tool produces counterexamples leading to safety requirement violation in the presence of faults, as well as all minimal set of fault combinations that can cause the safety requirement to be violated.

5. The safety engineers examine the results to assess the validity of the fault combinations and the fault tolerance level of the system design. If a design change is warranted, the model will be updated with the latest design change and the above process is repeated.

Figure 2.2 presents our proposed use of a single unified model to support both system design and safety analysis. It describes both system design and safety-relevant information that are kept distinguishable and yet are able to interact with each other. The shared model is a living model that captures the current state of the system design as it moves through the development lifecycle, allowing all participants of the ARP4754A process to be able to communicate and review the system design. Safety analysis artifacts can be generated directly from the model, providing the capability to more accurately analyze complex systems.

2.2 Related Work

The related work has two main foci. The first is in regard to safety analysis tools and research and how the Safety Annex differs from related approaches. The second outlines related work in minimal cut set generation, probabilistic computations over fault trees, and tools that implement this research.

Safety Analysis Tools and Research: A model-based approach for safety analysis was proposed by Joshi et al. [43–45]. In this approach, a safety analysis system model (SASM) is the central artifact in the safety analysis process, and traditional safety analysis artifacts, such as fault trees, are automatically generated by tools that analyze the SASM.

The contents and structure of the SASM differ significantly across different conceptions of MBSA. We can draw distinctions between approaches along several different axes. The first is whether they propagate faults explicitly through user-defined propagations, which we call *failure logic modeling* (FLM) or through existing behavioral modeling, which we call *failure effect modeling* (FEM). The next is whether models and notations are *purpose-built* for safety analysis vs. those that extend *existing system models* (ESM).

For FEM approaches, there are several additional dimensions. One dimension involves whether *causal* or *non-causal* models are allowed. Non-causal models allow simultaneous (in time) bi-directional error propagations, which allow more natural expression of some failure types (e.g., reverse flow within segments of a pipe), but are more difficult to analyze. A final dimension involves whether analysis is *compositional* across layers of hierarchically-composed systems or *monolithic*. Our approach is an extension of AADL (ESM), causal, compositional, mixed FLM/FEM approach.

Tools such as the AADL Error Model Annex, Version 2 (EMV2) [32] and HiP-HOPS for EAST-ADL [24] are *FLM*-based *ESM* approaches. Given many possible faults, these explicitly defined propagation relationships require substantial user effort and become very complex. In addition, it becomes the analyst's responsibility to determine whether faults can propagate; missing propagations lead to unsound analyses. In the approach of this dissertation, propagations occur through system behaviors (defined by the nominal contracts) with no additional user effort. Furthermore, the interactions that cannot occur through behavioral propagation alone can be manually added as dependencies; this provides the *mixed FEM/FLM* approach.

Closely related to our work is the model-based safety assessment toolset called COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [13]. COMPASS is a mixed *FLM/FEM*-based, *causal compositional* tool suite that uses the SLIM language, which is based on a subset of AADL, for its input models [14, 18]. In SLIM, a nominal system model and the error model are developed separately and then transformed into an extended system model. This extended model is automatically translated into input models for the NuSMV model checker [25, 57], MRMC (Markov Reward Model Checker) [47, 55], and RAT (Requirements Analysis Tool) [60]. The safety analysis tool xSAP [10] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [11]. COMPASS is an impressive tool suite, but some of the features that make AADL suitable for SW/HW architecture specification: event and event-data ports, threads, and processes, appear to be missing, which means that the SLIM language may not be suitable as a general system design notation (ESM).

SmartIFlow [42] is a *FEM*-based, *purpose-built*, *monolithic non-causal* safety analysis tool that describes components and their interactions using finite state machines and events. Verification is done through an explicit state model checker which returns sets of counterexamples for safety requirements in the presence of failures. SmartIFlow allows *non-causal* models containing simultaneous (in time) bi-directional error propagations. On the other hand, the tools do not yet appear to scale to industrial-sized problems, as mentioned by the authors: *"As current experience is based on models with limited size, there is still a long way to go to make this approach ready for application in an industrial context"* [42].

The Safety Analysis and Modeling Language (SAML) [37] is a *FEM*-based, *purpose-built*, *monolithic causal* safety analysis language. System models constructed in SAML can be used used for both qualitative and quantitative analyses. It allows for the combination of discrete probability distributions and non-determinism. The SAML model can be automatically imported into several analysis tools like NuSMV [25], PRISM (Probabilistic Symbolic Model Checker) [49], or the MRMC probabilistic model checker [47].

AltaRica [9, 59] is a *FEM*-based, *purpose-built*, *monolithic* safety analysis language with several dialects. There is one dialect of AltaRica which use dataflow (*causal*) semantics, while the most recent language update (AltaRica 3.0) uses non-causal semantics. The dataflow dialect has substantial tool support, including the commercial Cecilia OCAS tool from Dassault [7]. For this dialect the Safety assessment, fault tree generation, and functional verification can be performed with the aid of NuSMV model checking [15]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [6]. AltaRica 3.0 has support for simulation and Markov model generation through the OpenAltaRica (www.openaltarica.fr) tool suite.

Minimal Cut Set Generation and Probabilistic Evaluations: Formal verification tools based on model checking have been used to automate the generation of safety analysis artifacts [8, 12, 20, 21, 43, 68, 74]; unfortunately, these methods are limited in the generation of MinCutSets. This limitation impacts scalability of the tools produced and readability of fault trees; often the results do not represent the hierarchical structure of the system and result in a two-level fault tree that is a 'mile wide and an inch deep.' Some research has been performed address these limitations and to generate hierarchical fault trees [12, 16], but this approach is not performed in a compositional way. In reality, fault trees are often manually generated due to the restrictions of automated approaches.

The classical approach to *fault tree analysis* (FTA) is two-fold: first the minimal cut sets are determined by either a bottom-up or a top-down approach and then probabilistic computations proceed using the MinCutSets [40, 61, 73]. Due to scalability reasons in large systems, usually not all cut sets are considered; only the ones with lower cardinality (i.e., higher overall probability) are assessed [22, 73]. The representation of Boolean formulae as Binary Decision Diagrams (BDDs) was first formalized in the mid 1980s [23] and were extended to the representation of fault trees not many years later [62]. After this formalization, the BDD approach to FTA provided a new approach to safety analysis. The model is constructed using a BDD, then a second BDD (usually slightly restructured) is used to encode MinCutSets [63]. Probabilistic quantities are then assessed using either of these encodings. The hope was that this approach would provide "exact" probabilistic quantities of the top level event and not require the full MinCutSet generation to proceed. Unfortunately, due to the structure of BDDs, the worst case is exponential in size in terms of the number of variables [23,62,63]. In industrial sized systems, this is not realistically useful.

As the years continued, research was performed to encode the BDDs in more efficient ways and thus shrinking the size in order to address scalability; these are referred to as *zero-suppressed* BDDs, or ZBDDs [54]. While this helped to produce more scalable results than a pure BDD encoding, ZBDDs still suffered from scalability problems [46, 52].

Some research has been introduced that attempted to address this problem by use of under and over approximation of the top-level probability during the minimal cut set computations [12]. This allows for a converging limits on either side of the true probability. This provides valuable information even when all MinCutSets cannot be computed. Other research and tools have given the over-approximation as simply a few orders of magnitude higher than the under-approximation to account for the error term dropped during MinCutSet computations where only certain cardinalities are considered [15, 21, 22, 24].

2.3 Tools and Modeling Language

2.3.1 Architecture Analysis and Design Language

We are using the Architectural Analysis and Design Language (AADL) to construct system architecture models. AADL is an SAE International standard language that provides a unifying framework for describing the system architecture for performance-critical, embedded, real-time systems [4, 31]. From its conception, AADL has been designed for the design and construction of avionics systems. Rather than being merely descriptive, AADL models can be made specific enough to support system-level code generation. Thus, results from analyses conducted, including the new safety analysis proposed here, correspond to the system that will be built from the model.

An AADL model describes a system in terms of a hierarchy of components and their interconnections, where each component can either represent a logical entity (e.g., application software functions, data) or a physical entity (e.g., buses, processors). An AADL model can be extended with language annexes to provide a richer set of modeling elements for various system design and analysis needs (e.g., performance-related characteristics, configuration settings, dynamic behaviors). The language definition is sufficiently rigorous to support formal analysis tools that allow for early phase error/fault detection.

2.3.2 Assume Guarantee Reasoning Environment

The Assume Guarantee Reasoning Environment (`AGREE`) is a tool for formal analysis of behaviors in AADL models [28]. It is implemented as an AADL annex and annotates AADL components with formal behavioral contracts. Each component's contracts can include assumptions and guarantees about the component's inputs and outputs respectively, as well as predicates describing how the state of the component evolves over time.

`AGREE` translates an AADL model and the behavioral contracts into Lustre [38] and then queries a user-selected model checker to conduct the back-end analysis. The analysis can be performed compositionally or monolithically.

Monolithic vs. Compositional Analysis: Compositional analysis of systems was introduced in order to address the scalability of model checking large software systems [28, 39, 58]. Monolithic verification and compositional verification are two ways that mathematical verification of component properties can be performed. In monolithic analysis, the model is flattened and the top level properties are proved using the contracts of all components. The analysis can alternatively be performed compositionally following the architecture hierarchy such that analysis at a higher level is based on the components at the next lower level and conducted layer by layer; the components of a system are organized hierarchically and each layer of the architecture is viewed a system. The idea is to partition the formal analysis of a system architecture into verification tasks that correspond into the decomposition of the architecture.

A component contract in `AGREE` is an assume-guarantee pair. Intuitively, the meaning of a pair is: if the assumption is true, then the component will ensure that the guarantee is true. For any given layer, the proof consists of demonstrating that the system guarantee is provable given the guarantees of its direct subcomponents and the system assumptions. This proof is performed one layer at a time starting from the top level of the system. When compared to monolithic analysis (i.e., analysis of the flattened model composed of all components), the compositional approach allows the analysis to scale to much larger systems [28].

2.3.3 Safety Annex for AADL

The Safety Annex for AADL and its supporting extensions to the AADL tools provide the ability to reason about faults and faulty component behaviors in AADL models and has been developed throughout the course of this project [69–72]. In the Safety Annex approach, formal assume-guarantee contracts are used to define the nominal behavior of system component and the nominal model is verified using AGREE. The Safety Annex implementation weaves faults into the nominal model and analyzes the behavior of the system in the presence of faults. The tool supports behavioral specification of faults and their implicit propagation through behavioral relationships in the model and provides support to capture binding relationships between hardware and software components of the system.

2.4 Definitions

The Boolean Satisfiability (`SAT`) problem attempts to determine if there exists a total truth assignment to a given propositional formula, that evaluates to TRUE. Generally, a propositional formula is any combination of the disjunction and conjunction of literals (as an example, a and $\neg a$ are literals). `SAT` solvers in model checking work over a constraint system to determine satisfiability.

A constraint system C is an ordered set of n abstract constraints $\{C_1, C_2, ..., C_n\}$ over a set of variables. The constraint C_i restricts the allowed assignments of these variables in some way [50]. Given a constraint system, we require some method of determining, for any subset $S \subseteq C$, whether S is *satisfiable* (`SAT`) or *unsatisfiable* (`UNSAT`). When a subset S is SAT, this means that there exists an assignment allowed by all $C_i \in S$; when no such assignment exists, S is considered `UNSAT`. Given a constraint system C, there are certain subsets of C that are of interest in terms of satisfiability. Definitions 2-4 are taken from research by Liffiton et. al. [50].

Definition 1. *: A Minimal Unsatisfiable Subset (MUS) M of a constraint system C is a subset $M \subseteq C$ such that M is unsatisfiable and $\forall c \in M : M \setminus \{c\}$ is satisfiable.*

An MUS is the minimal explanation of the constraint systems infeasability.

Definition 2. *: A Minimal Correction Set (MCS) M of a constraint system C is a subset $M \subseteq C$ such that $C \setminus M$ is satisfiable and $\forall S \subset M : C \setminus S$ is unsatisfiable.*

A *MCS* can be seen to "correct" the infeasability of the constraint system by the removal from C the constraints in *MCS*.

A duality exists between the *MUS*s of a constraint system and the *MCS*s as established by Reiter [64]. This duality is defined in terms of *Minimal Hitting Sets* (*MHS*). A hitting set of a collection of sets A is a set H such that every set in A is "hit" be H; H contains at least one element from every set in A. Every *MUS* of a constraint system is a minimal hitting set of the system's *MCS*s, and likewise every *MCS* is a minimal hitting set of the system's *MUS*s [29, 50, 64].

Definition 3. *: Given a collection of sets K, a hitting set for K is a set $H \subseteq \cup_{S \in K} S$ such that $H \cap S \neq \emptyset$ for each $S \in K$. A hitting set for K is minimal if and only if no proper subset of it is a hitting set for K.*

Since we are interested in sets of active faults that cause violation of the safety property, we turn our attention to Minimal Cut Sets.

Definition 4. *A Minimal Cut Set (MinCutSet) is a minimal collection of faults that lead to the violation of the safety property. Furthermore, any subset of a MinCutSet will not cause this property violation.*

We define a minimal cut set consistently with much of the research in this field [30, 65].

Inductive Validity Cores: Given a complex model, it is useful to extract traceability information related to the proof; in other words, which elements of the model were necessary to construct the proof. An algorithm was introduced by Ghassabani et al. to provide Inductive Validity Cores (*IVC*) as a way to determine which model elements are necessary for the inductive proofs of the safety properties for sequential systems [35]. Given a safety property of the system, a model checker is invoked to construct a proof of the property. The *IVC* generation algorithm extracts traceability information from the proof process and returns a minimal set of the model elements required in order to prove the property. Later research extended this algorithm in order to produce all minimal *IVC* elements (`All_IVCs`) [5, 36].

The `All_IVCs` algorithm considers a constraint system consisting of the assumptions and contracts of system components and the negation of the safety property of interest (i.e. the top level event). It then collects all Minimal Unsatisfiable Subsets (*MUS*s) of this constraint system; these are the minimal explanations of the constraint systems infeasibility in terms of the *negation* of the safety property. Equivalently, these are the minimal model elements necessary to prove the safety property.

Chapter 3

Proposed Approach

The contributions of this project can be seen as two main categories of research work. The first set was accomplished in the beginning phase of this project: behavioral and explicit error propagation through the implementation of the Safety Annex for AADL [70,72]. The remaining pieces of this research provide the bulk of the contribution and consist of the compositional generation of minimal cut sets through the transformation of inductive validity cores and using the fault tree generated by this transformation to compute the probability of a safety property violation.

The usage of the terms error, failure, and fault are defined in ARP4754A and are described here for ease of understanding [67]. An *error* is a mistake made in implementation, design, or requirements. A *fault* is the manifestation of an error and a *failure* is an event that occurs when the delivered service of a system deviates from correct behavior. If a fault is activated under the right circumstances, that fault can lead to a failure. The term *error propagation* is used to refer to the propagation of the corrupted state caused by an active fault.

3.1 The Safety Annex and Fault Modeling

3.1.1 Implementation

The Safety Annex is written in Java as a plug-in for the OSATE AADL toolset, which is built on Eclipse. It is not designed as a stand-alone extension of the language, but works with behavioral contracts specified using the `AGREE` AADL annex [28]. The architecture of the Safety Annex and related tools is shown in Figure 3.1.

A system model is defined and implemented in AADL and `AGREE` contracts are used to define the nominal behaviors of system components as guarantees that hold when assumptions about the values the component's environment are met. The `AGREE` *abstract syntax tree* (AST) reflects the system architecture and the contracts defined on components; this is translated into `Lustre` [38], a synchronous dataflow language used by `JKind`, an infinite-state model checker for safety properties [33].

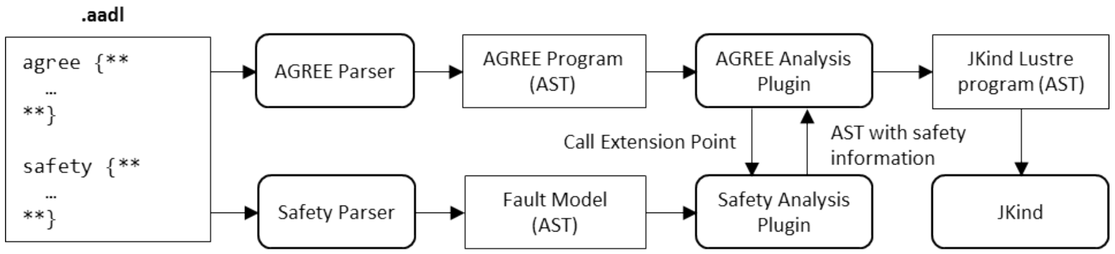
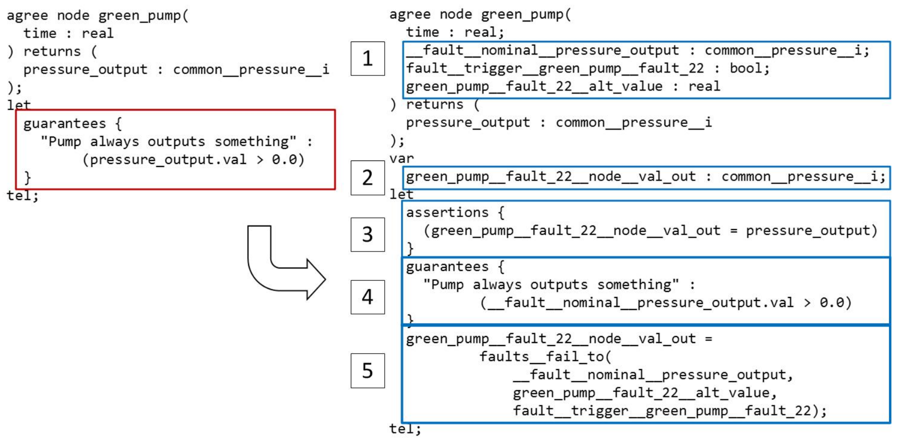


Figure 3.1: Safety Annex Plug-in Architecture

When the fault model is created using the Safety Annex, an extension point in AGREE allows the fault information to be added to the AGREE AST. This extended model AST is then translated into Lustre and verified by JKind..

This extension allows faults to modify the behavior of component outputs. An example of a portion of a nominal AGREE node written in Lustre and its extended contract is shown in Figure 3.2. The left column of the figure shows the nominal Lustre pump definition with an AGREE contract on the output; and the right column shows the additional local variables for the fault (boxes 1 and 2), the assertion binding the fault value to the nominal value (boxes 3 and 4), and the fault node definition (box 5). Once augmented with fault information, the AGREE model follows the standard translation path to the model checker JKind.



```
agree node green_pump(
  time : real
) returns (
  pressure_output : common__pressure__i
);
let
 guarantees {
   "Pump always outputs something" :
        (pressure_output.val > 0.0)
 }
tel;
```

```
agree node green_pump(
  time : real;
  __fault__nominal__pressure_output : common__pressure__i;
  fault__trigger__green_pump__fault_22 : bool;
  green_pump__fault_22__alt_value : real
) returns (
  pressure_output : common__pressure__i
);
var
  green_pump__fault_22__node__val_out : common__pressure__i;
let
  assertions {
    (green_pump__fault_22__node__val_out = pressure_output)
  }
  guarantees {
    "Pump always outputs something" :
           (__fault__nominal__pressure_output.val > 0.0)
  }
  green_pump__fault_22__node__val_out =
          faults__fail_to(
              __fault__nominal__pressure_output,
              green_pump__fault_22__alt_value,
              fault__trigger__green_pump__fault_22);
tel;
```

Figure 3.2: Nominal AGREE Node and Extension with Faults

3.1.2 The Sensor System

An example is a helpful guide to illustrate the Safety Annex. A simple sensor system is shown in Figure 3.3. The top level of the system takes two inputs; environmental pressure and temperature. Each subsystem monitors its corresponding input and will send a shutdown command if the input surpasses some threshold. Each subsystem consists of three sensors. Each subsystem's output is regulated by a majority voter; thus, if the majority of sensors in a subsystem report high temperature (or pressure), the shutdown command is sent.

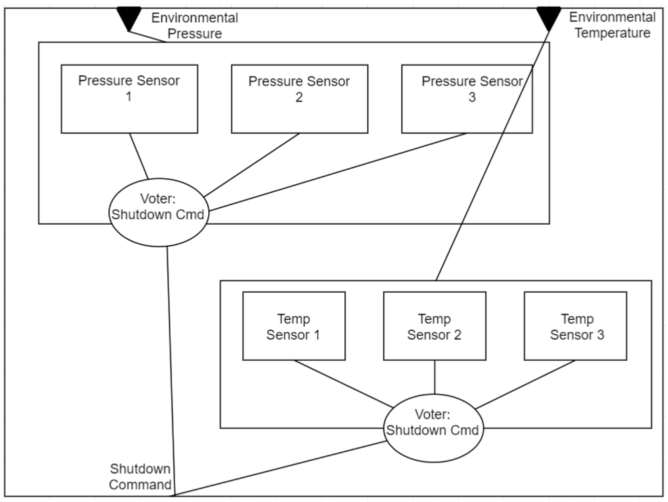


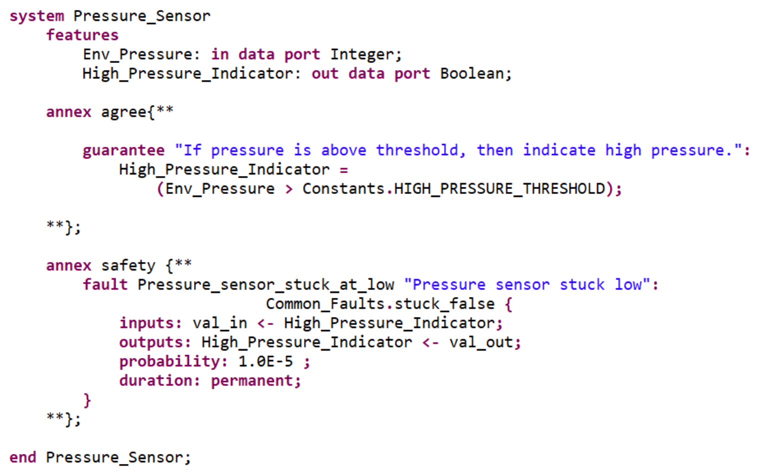
Figure 3.3: A Simple Sensor System

This example will be referenced intermittently throughout this document.

3.1.3 The Safety Annex for the Sensor System

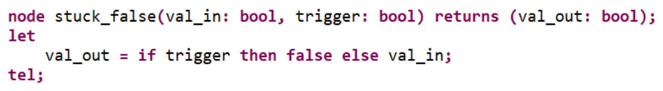
In short, the Safety Annex allows users to describe faults and connect them to AADL component outputs. Within the AADL component instance model, an annex is added which contain the fault definitions for the given component. The flexibility of the fault definitions allows the user to define numerous types of fault *nodes* by utilizing the `AGREE` node syntax. An example of the `AGREE` and Safety Annexes defined on a pressure sensor in the sensor system is shown in Figure 3.4. The fault definition connects to a *fault node* that is defined in Figure 3.5 (the reference to this fault node is seen in the syntax of the fault definition in Figure 3.4: `Common_Faults.stuck_false`). The fault node, `stuck_false`, restricts the output to false, even when the pressure is higher than the threshold.

When a fault is activated by a trigger condition, the fault node outputs a value which overrides the component's nominal output value. The faulty behavior may violate the contracts of other components in the system, including assumptions of downstream components. The impact of a fault is shown by counterexamples returned by the model checker when fault analysis



```
system Pressure_Sensor
    features
        Env_Pressure: in data port Integer;
        High_Pressure_Indicator: out data port Boolean;

    annex agree{**

        guarantee "If pressure is above threshold, then indicate high pressure.":
            High_Pressure_Indicator =
                (Env_Pressure > Constants.HIGH_PRESSURE_THRESHOLD);

    **};

    annex safety {**
        fault Pressure_sensor_stuck_at_low "Pressure sensor stuck low":
                            Common_Faults.stuck_false {
            inputs: val_in <- High_Pressure_Indicator;
            outputs: High_Pressure_Indicator <- val_out;
            probability: 1.0E-5 ;
            duration: permanent;
        }
    **};

end Pressure_Sensor;
```

Figure 3.4: A Pressure Sensor and the AGREE and Safety Annexes



```
node stuck_false(val_in: bool, trigger: bool) returns (val_out: bool);
let
    val_out = if trigger then false else val_in;
tel;
```

Figure 3.5: A Fault Node Definition

is run on the model.

The fields of a fault definition are shown in Figure 3.4 and are described for clarity.

- **Fault node statement**: This is seen as `Common_Faults.stuck_false` in the figure and refers to a file of fault node definitions (`Common_Faults`) containing commonly used faults such as stuck false, stuck true, fail to, or in this case `stuck_false`. This provides the node definition in the `Lustre` code.

- **Input**: The inputs state which values from the component are passed into the node definition. In this case, the nominal component (pressure sensor) output `High_Pressure_Indicator` is passed into the node parameter `val_in`.

- **Output**: The output from the fault node is designated to an AADL component feature, essentially overwriting it if the fault is active. In this case, the node's output will be the pressure sensor output: *High_Pressure_Indicator*.

- **Probability**: This field specifies the probability of occurrence for this fault. This is required for probabilistic analysis.

- **Duration**: When a fault becomes active, it is designated as a permanent fault.

3.1.4 Modeling Dependent Faults

The impact of a fault is computed by the AGREE model checker when the safety analysis is run on the fault model and thus determined implicitly; the error propagation is not explicitly defined as in other closely related tools [11, 32]. However, failures in hardware (HW) components can trigger behavioral faults in the system components that depend on them. This makes it beneficial to allow for explicit error propagation and the notion of dependencies in the fault model. For example, a CPU failure may trigger faulty behavior in the threads bound to that CPU or a failure in one HW component may trigger failure in other HW components located nearby, such as overheating, fire, or an explosion in the containment location. The Safety Annex provides the capability to explicitly model the impact of hardware failures on other faults, behavioral or non behavioral.

Users specify dependencies between the HW component faults and faults that are defined in other components, either HW or SW. The hardware fault then acts as a trigger for its dependencies. This allows for propagation from the faulty HW component to the components that rely on it, affecting the behavior on the outputs of the affected SW components. Within the Lustre implementation, this corresponds to a statement linking the active fault to all dependent faults. Thus, if the triggering fault is active, so are all dependencies.

3.2 Verification Capabilities in the Face of Component Failures

When performing safety analysis, it is useful to be able to restrict the number of failed components in terms of a maximum number or a probabilistic computation. Instead of allowing the Lustre fault activation assignments to remain unrestricted, statements can be added to the Safety Annex that specify the type of analysis to be run. This is either *maximum fault analysis* or *probabilistic threshold analysis*.

The fault analysis statement (also referred to as the fault hypothesis) resides in the AADL system implementation that is selected for verification. This may specify either a maximum number of faults that can be active at any point in execution (Figure 3.6) or that the only faults to be considered are those whose probability of simultaneous occurrence is above some probability threshold (Figure 3.7).

```
annex safety {**
        analyze : max 1 fault
**};
```

Figure 3.6: Max N Faults Analysis Statement

Tying back to the fault tree analysis in traditional safety analysis, the former is analogous to restricting the minimal cut sets to a specified maximum number of terms, and the latter is analogous to restricting the minimal cut sets to only those whose probability is above some set value. In the former case, we assert that the sum of the active faults is at or below some

```
annex safety {**
        analyze : probability 1.0E-7
**};
```

Figure 3.7: Probability Analysis Statement

integer threshold. In the latter, we determine all combinations of faults whose probabilities are above the specified probability threshold, and describe this as a proposition over fault activation literals. Active faults are divided into two categories: independently active (activated by its own triggering event) and dependently active (activated when the faults they depend on become active). The top level fault hypothesis applies to independently active faults. Faulty behaviors augment nominal behaviors whenever their corresponding faults are active (either independently active or dependently active).

3.2.1 Maximum Fault Analysis

The max fault hypothesis specifies a maximum number of faults that can be active at any point in execution. This is analogous to restricting the minimal cut sets to a specified maximum number of terms in the fault tree analysis in traditional safety analysis. In implementation (i.e., the translated `Lustre` model feeding into the model checker), we assert that the sum of the fault activation literals assigned to *true* is below some integer threshold. This can be seen in Figure 3.8 with maximum fault count set at 3. While solving the satisfiability problem, JKind

```
assert (__fault__global_count =
  ((if __fault__independently__active__TempSensor1 then 1 else 0) +
  ((if __fault__independently__active__TempSensor2 then 1 else 0) +
  ((if __fault__independently__active__TempSensor3 then 1 else 0) +
  ((if __fault__independently__active__PressureSensor1 then 1 else 0) +
  ((if __fault__independently__active__PressureSensor2 then 1 else 0) +
  ((if __fault__independently__active__PressureSensor3 then 1 else 0)))))));

assert (__fault__global_count <= 3);
```

Figure 3.8: `Lustre` Statement for Fault Count

will iterate through the possible combinations given the restriction of the hypothesis statement. If the system with respect to a certain safety property is found to be UNSAT, a counterexample showing the trace of the system (including active faults) is given to the user.

When running the analysis compositionally, JKind employs a top-down approach; it attempts to prove the top-level requirements using the contracts of the direct subcomponents, then it moves to the next layer down, and so on. If using maximum N fault analysis compositionally, the results pertain only to the current level of analysis.

In the traditional safety assessment process, safety analysts require information regarding single points of failure on specific components or even system wide. The Safety Annex supports this need and the detailed counterexamples can provide important insight into the system development process.

3.2.2 Probabilistic Threshold Analysis

The grammar of the Safety Annex allows for a probabilistic assignment to a fault definition as shown in Figure 3.4. The *probabilistic fault hypothesis* specifies that only faults whose probability of simultaneous occurrence is above some probability threshold should be considered. In the implementation, we determine all combinations of faults whose probabilities are above the specified probability threshold and describe this as a proposition over the fault activation literals. If the probability of a combination of faults is less than the designated top level threshold, these faults may be activated and the behavioral effects can be seen through a counterexample.

To perform this analysis, it is assumed that the faults occur independently. Algorithm 1 describes this process. First, all faults are removed from consideration that are too unlikely given the probability threshold. The remaining faults are arranged in a priority queue $\mathcal{Q}$ from high to low. Take the fault with highest probability from the queue (step 5) and attempt to combine the remainder of the faults in $\mathcal{R}$ (step 7). If this combination is lower than the threshold (step 8), then we do not take into consideration this set of faults and instead remove the tail of the remaining faults in $\mathcal{R}$.

In this calculation, we assume independence among the faults, but in the Safety Annex it is possible to define dependence between faults using a fault propagation statement. After the possible fault combinations are computed using Algorithm 1, the triggered dependent faults are added to the combination as appropriate while ignoring their respective probabilities.

Algorithm 1: Monolithic Probability Analysis

```
1  $\mathcal{F} = \{\}$ : fault combinations above threshold ;
2  $\mathcal{Q}$ : faults, $q_i$, arranged with probability high to low ;
3  $\mathcal{R} = \mathcal{Q}$ , with $r \in \mathcal{R}$;
4  while $\mathcal{Q} \neq \{\} \wedge \mathcal{R} \neq \{\}$ do
5      $q =$ removePriorityElement($\mathcal{Q}$) ;
6      for $i = 0 : |\mathcal{R}|$ do
7          $prob = q \times r_i$ ;
8          if $prob < threshold$ then
9              removeTail($\mathcal{R}, i : |\mathcal{R}|$);
10         else
11             add($\{q, r_i\}, \mathcal{Q}$);
12             add($\{q, r_i\}, \mathcal{F}$);
```

Often it is the case that safety properties contain probabilistic thresholds that must be analyzed during the safety assessment process. This type of fault hypothesis statement and the associated fault analysis supports these requirements.

3.3 Generation of Minimal Cut Sets from *MIVCs*

As was previously explained, the MIVCs (Minimal Inductive Validity Cores) are *MUS*s (Minimal Unsatisfiable Subsets) of a constraint system. The *MCS*s (Minimal Correction Sets) can be obtained from all *MUS*s by generating the hitting sets of all *MUS*s. Thus, the `All_IVCs` algorithm produces all *MUS*s and the hitting set algorithm can transform these into *MCS*s. If the constraint system is defined to take into account faults, these *MCS*s can be transformed into *MinCutSets*.

Recall that a constraint system is an ordered set of abstract constraints over a set of variables. In the case of a nominal model augmented with faults, a constraint system can be defined as follows. Let F be the set of fault activation literals and G be the set of component contracts (guarantees).

Definition 5. *A constraint system* $C = \{C_1, C_2, ..., C_n\}$ *where for* $i \in \{1, ..., n\}$, C_i *has the following constraints for any* $f_j \in F$ *and* $g_k \in G$ *with regard to the top level property* P*:*

$$C_i \in \begin{cases} f_j : & false \\ g_k : & true \\ P : & false \end{cases}$$

The `All_IVCs` algorithm collects all minimal unsatisfiable subsets of a given transition system in terms of the *negation* of the top level property [5, 36]. Assuming that the nominal model proves (no faults are active), it is not surprising that the guarantees (constrained to *true*) and the negation of the safety property is UNSAT. The *MUS*s are the minimal explanation of the infeasibility of this constraint system; equivalently, these are the minimal sets of model elements necessary for proof of the safety property.

We utilize this algorithm by providing not only component contracts constrained to *true* as model elements, but also fault activation literals constrained to *false*, i.e. the faults are inactive. Thus the resulting MIVCs (*MUS*s) will contain the required contracts and constrained fault activation literals necessary to prove the safety property.

Because of the duality between *MUS*s and *MCS*s, all *MCS*s can be obtained by finding the hitting sets. The *MCS* can be seen to correct the infeasibility of the constraint system and provides the minimal such correction. By removing the constraints from C that are found in any *MCS*, C becomes satisfiable. In terms of our constraint system with fault activation literals, by *activating* the faults in the *MCS* and *violating* the contracts in the *MCS*, we can prove the *negation* of the property P. If the contracts in the *MCS* are replaced with the faults that cause its violation, the *MCS* is transformed into a *MinCutSet*. A high level summary of the steps of this transformation are shown in Figure 3.9.

The Steps of Transformation from *MUS* to *MinCutSet*

1. Redefine constraint system by adding fault activation literals to the constraint system used by the `All_IVCs` algorithm.

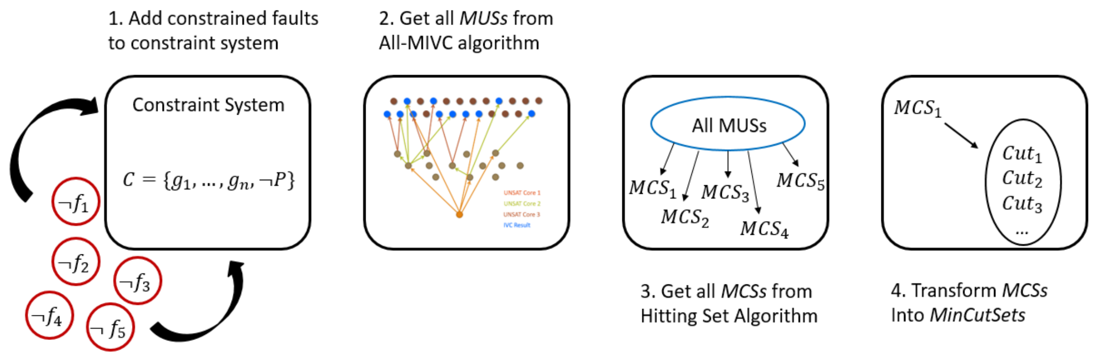

1. Add constrained faults to constraint system
2. Get all *MUSs* from All-MIVC algorithm
Constraint System
$C = \{g_1, \dots, g_n, \neg P\}$
$\neg f_1$
$\neg f_2$
$\neg f_3$
$\neg f_4$
$\neg f_5$
UNSAT Core 1
UNSAT Core 2
UNSAT Core 3
IVC Result
All MUSs
MCS_1
MCS_2
MCS_3
MCS_4
MCS_5
Cut_1
Cut_2
Cut_3
...
3. Get all *MCSs* from Hitting Set Algorithm
4. Transform *MCSs* Into *MinCutSets*

Figure 3.9: Steps of the Transformation Process

2. Collect all *MUSs* through the `All_IVCs` algorithm.
3. Transform the *MUS*s (*MIVCs*) into *MCS*s by use of a hitting set algorithm [34, 56].
4. Replace all contracts in the *MCS*s by the faults that cause their violation (*MinCutSets* for that contract).

3.3.1 Illustrative Example

Using the sensor system described in Section 3.1.2, this transformation can be more easily understood. The top-level safety property of the system states that a shutdown occurs when and only when it should:

$$(temp\_input > threshold) \lor (pressure\_input > threshold) \\ \iff Shutdown$$

The property of each sensor subsystem states that when the majority of sensors report high, denoted as out_{si} for sensor si, a shutdown command is sent:

$$majority\_vote(out_{s1}, out_{s2}, out_{s3}) \iff Shutdown$$

Each sensor has a behavioral property stating that if the environmental input is high, the shutdown command is sent:

$$(environment > threshold) \iff Shutdown$$

A fault is defined on each of the sensors which when active, causes the sensors to fail low (the environmental input is high, but they do not send a shutdown command). It is easy to see that this system is resilient to a single fault anywhere due to the majority voting mechanism.

Leaf Level: To illustrate the *MIVC* to *MinCutSet* transformation, we start with a leaf level of the system: a pressure sensor, $p1$. (Note: The `All_IVCs` algorithm proceeds in a top-down fashion, but for clarification in the example, we look at the results in a bottom-up approach.) In this layer, the `All_IVCs` algorithm treats the sensor guarantee, g_{p1}, as the property of interest:

$$g_{p1} : (environment > threshold) \iff Shutdown$$

and the model elements provided to the `All_IVCs` algorithm consist of only fault activation literals for this leaf component constrained to *false*; we call the fault on $p1$, f_{p1}. Thus, the constraint system for this layer is:

$$C_{leaf} = \{\neg f_{p1} \neg g_{p1}\}$$

The `All_IVCs` algorithm returns all *MIVC*s for this constraint system, $MIVC = \{\{\neg f_{p1}\}\}$, of which there is only one: in order for this leaf level guarantee to hold, the fault must be constrained to false.

The hitting set algorithm finds that the set $\{\neg f_{p1}\}$ sufficiently 'hits' all *MIVC*s and this is our *MCS*. This means that if the constraint is removed from f_{p1} in C, our constraint system is satisfiable. Thus, the *MinCutSet* for $\neg g_{p1}$ is $\{f_{p1}\}$, and in the same manner, we find the *MinCutSets* for g_{p2}, g_{p3} and the guarantees for the temperature sensors.

Mid Level: For the mid-level pressure system, P, the guarantee of interest is:

$$g_P : majority\_vote(out_{p1}, out_{p2}, out_{p3}) \iff Shutdown$$

and can be also written as:

$$g_P : ((out_{p1} \wedge out_{p2}) \vee (out_{p1} \wedge out_{p3}) \vee (out_{p2} \wedge out_{p3})) \iff Shutdown$$

The constraint system looks only at the supporting guarantees (one level below) and any fault literals in the current level. Thus, the constraint system is:

$$C = \{g_{p1}, g_{p2}, g_{p3}, \neg g_P\}$$

The resulting *MIVC*s are: $\{\{g_{p1}, g_{p2}\}, \{g_{p1}, g_{p3}\}, \{g_{p2}, g_{p3}\}\}$. If any pairwise combination of guarantees hold, then the property g_P holds, i.e. the shutdown command is sent. The hitting set algorithm determines all sets whose intersection with all *MIVC*s are nonempty. In this case, they are equivalent to the *MIVC*s.

Since *MinCutSets* only contain faults, a replacement must be made between the guarantees found in the *MCS*s and the faults that cause the violation of these guarantees. We know those faults due to the processing done at the leaf level; after replacement, we obtain the *MinCutSets* for $\neg g_P$: $\{\{f_{p1}, f_{p2}\}, \{f_{p1}, f_{p3}\}, \{f_{p2}, f_{p3}\}\}$.

Top Level: The top level safety property is:
$SP : (temp\_input > threshold) \vee (pressure\_input > threshold) \iff Shutdown$, and

requires guarantees from both the temperature and pressure subsystems, g_P and g_T respectively. The resulting *MIVC*s are: $\{\{g_P\}, \{g_T\}\}$. This is also equivalent (in this case) to the *MCS*s generated through the hitting set algorithm. Replacement of these contracts with the faults that cause their violation produces all *MinCutSets* for the top level event (violation of the top level safety property). This is: any combination of two faults occurring in either the temperature or the pressure sensor systems will result in violation of the safety property.

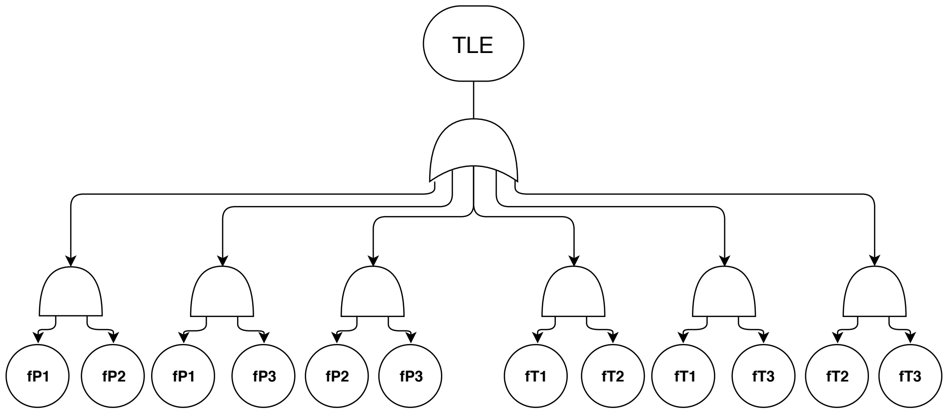


Figure 3.10: Flat Fault Tree for the Sensor Example

A typical fault tree generated by many of the current research tools available shows very little hierarchical information. An example of a flat fault tree is shown in Figure 3.10 and shows only the *MinCutSets* that contribute to the top level event (TLE). (Note: probabilistic information is usually displayed with the fault tree, but not included here for readability.)

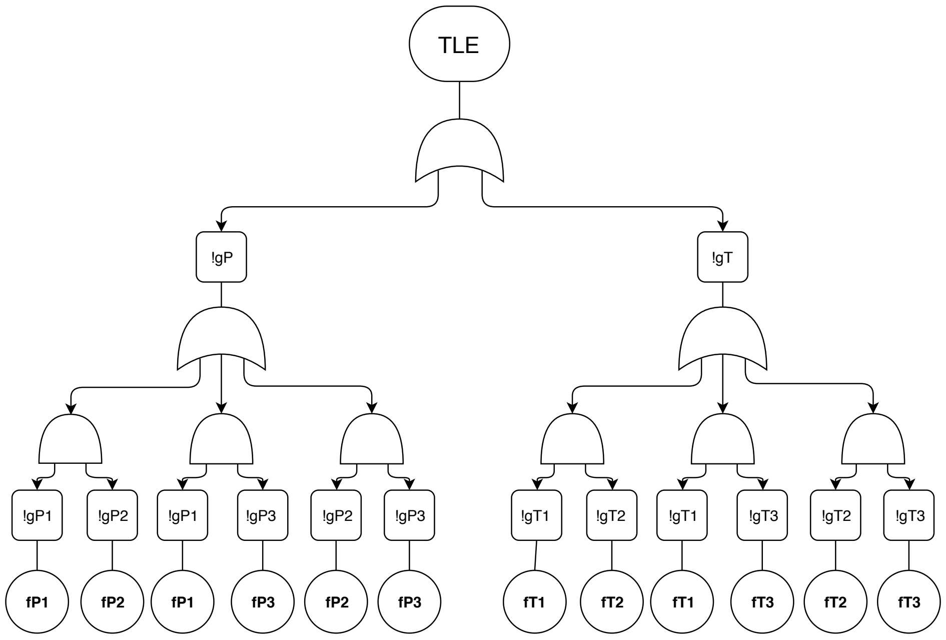


Figure 3.11: Hierarchical Fault Tree for the Sensor Example

An example of a hierarchical fault tree that could be produced by the information gathered through this process is shown in Figure 3.11. Each layer of the architecture is shown in the fault tree and represented by the violated contracts of that layer. The OR-gate between the lower level contracts and the TLE reflects the safety property itself: if either of the subsystems fail, the safety property is violated. The second layer of the tree also reflects the relationship between the sensors and the sensor subsystem: if any pairwise combination of sensors fail, the mid-level contract is violated. This process of using *IVC*s to generate minimal cut sets will provide layer by layer the necessary information to not only collect the *MinCutSets*, but also reflect the hierarchical nature of the system within the fault trees generated.

The example should suffice to show the basic outline of the algorithm. The proofs showing that this transformation is logically valid will be provided in the dissertation and the algorithms will be implemented in the Safety Annex.

3.4 Compositional Probabilistic Computations

Safety analysis techniques aim at demonstrating that the system meets the requirements necessary for certification and use in the presence of faults. In many domains, there are two main steps to this process: (1) the generation of all minimal cut sets (*MinCutSets*), i.e. the minimal set of faults that lead to a violation of the top level property (known as a *top-level event* or TLE) and (2) the computation of the corresponding fault probability, i.e., the probability of reaching the TLE, given probabilities for the faults in the system.

The probability of the TLE is used to find the likelihood of the safety hazard that it represents. While evaluation of the fault model with a given probabilistic threshold does provide information on the safety hazards, it is also informative and desirable to find the overall probability of the occurrence of a hazard.

Such computations can be carried out by leveraging the logical formula represented by the disjunction of all *MinCutSets* which are in turn conjunctions of their constituents.

Given a set of *MinCutSets* and a mapping $\mathcal{P}$ that gives the probability of the basic faults in the system f_i, it is possible to compute the probability of occurrence of the TLE. Assuming that the basic faults are independent, the probability of a single *MinCutSet*, σ is given by the product of the probabilities of its basic faults:

$$\mathcal{P}(\sigma) = \prod_{f_i \in \sigma} \mathcal{P}(f_i)$$

For a set of *MinCutSets*, S, the probability can be computed using the following recursive formula:

$$\mathcal{P}(S_1 \cup S_2) = \mathcal{P}(S_1) + \mathcal{P}(S_2) - \mathcal{P}(S_1 \cap S_2)$$

Due to the independence assumption, $\mathcal{P}(S_1 \cap S_2)$ is computed as $\mathcal{P}(S_1) \cdot \mathcal{P}(S_2)$. Using this technique, it is theoretically possible to compute the overall probability of a TLE given all *MinCutSets* and an independence assumption, but in the real world of safety analysis this poses some problems, the largest of which is scalability. Given a very large system with many possible faults, it becomes difficult to compute all *MinCutSets* without pruning of any kind. If one is unable to complete such computations, it is not possible to simply compute the probabilities as described above.

As previously discussed, it is standard practice to consider cut sets only up to a given cardinality. As the cardinality of the cut sets increase, the likelihood of their occurrence decreases and as the system increases in size, the possible combinations of problematic faults will inevitably increase, at times exponentially. In order to simplify these calculations and address the problem of scalability, *MinCutSets* up to a certain cardiality are considered. Everything above that is "safely" ignored, and then specific criteria is used to over-approximate the error. The end result of these computations is above the actual probability (i.e., a safe approximation), but close enough to be significant.

Another drawback to computing an exact probability of the TLE is the problem of model reliability and exactness. For instance, if two groups of engineers each built a model of the same system, the models may not be equivalent; especially in terms of behavioral properties. Since our approach of computing *MinCutSets* depends on the properties over system components, the calculated top-level probability will change. Different representations of the system will alter the computations.

As an example, assume that Figure 3.12 is a snapshot of a given layer in a system designed by the first group of engineers. Component A has a contract, G_A, which is determined by the `All_IVCs` algorithm to depend on a lower level contract, $g_A = a \wedge b$. g_A will be in the set of *IVC*s for the contract G_A. Assume that only a is required for the proof of G_A.

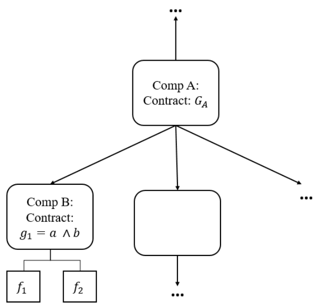


Figure 3.12: Sample System Contract Part I

Two faults are defined on component B: f_1 violates a and f_2 violates b. Since each of these faults will violate the contract g_A, each of these faults will be found in *MinCutSets* for G_A.

Now assume that Figure 3.13 was the system representation built by the second group of engineers. The basic system structure is the same, but this time there are two contracts for

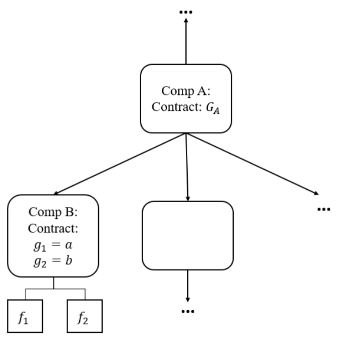


Figure 3.13: Sample System Contract Part II

component B: $g_1 = a$ and $g_2 = b$. Since b is not required for the proof of G_A, only g_1 is found in the *IVC*s for G_A and thus only f_1 will be seen in the *MinCutSets* for this particular contract.

In this simple example, it is easy to see why a single computation of the top-level probability of a system may be misleading and may not reflect the actual probability of the system. To this end, we wish to find a way to accurately obtain higher and lower bounds of what the probability is likely to be.

A lower bound for the probability can be obtained by choosing a maximum cardinality for *MinCutSets* before computations begin, e.g. assume that cardinalities above n are too unlikely to be significant. This will contribute to better scalability in large systems with numerous possible cut sets. A higher bound is commonly assigned by experts of the system, e.g. around 3 orders of magnitude higher than the lower approximation.

It would be beneficial to leverage the *IVC* information to provide both the lower and upper bound approximations and show that these values are significant and trustworthy.

3.5 Evaluation of Results

The goal of this dissertation is to provide usable and reliable safety assessment artifacts and methodologies that can be implemented on industrial scale systems. Furthermore, the artifacts should shed light on the system organizational hierarchy, interacting components, and problematic faults. This is useful within the assessment process and can provide valuable feedback to engineers on both the safety and development side.

To this end, we will implement a large scale system model in the aerospace domain and collect the artifacts and other pertinent information derived from this process. We will show how this information can provide feedback to the system engineers and be useful in the certification process.

The Wheel Brake System (WBS) was initially developed for illustrative purposes by SAE AIR6110 [3] and has been used in numerous case studies and research into Model-Based Safety Analysis [17, 26, 27, 48, 53, 72]. This model is large enough to assess scalability and provides numerous complex component interactions and behaviors. The AADL version of this model will be annotated with `AGREE` contracts that accurately describe the behavior of the system components. Faults will be attached to each of the components as per descriptions in AIR6110. The analysis of this approach will address the four main contributions of the Safety Annex:

1. Maximum n fault threshold compositional analysis
2. Probabilistic threshold monolithic analysis
3. Minimal cut set computations (max n and probabilistic threshold)
4. Probabilistic evaluations and computations

The artifacts generated from each run will be assessed in light of safety assessment and fit into the process as described in Sections 2.1.1 and 2.1.2.

Using other models representative of the aerospace industry, we will explore the necessity of using a shared system model that drives the development and safety processes, and show the fault modeling flexibility of the Safety Annex.

Chapter 4

Conclusion

System safety analysis is cruicial in the development of critical systems and the generation of accurate and scalable results is invaluable to the assessment process. Having multiple ways to capture complex dependencies between faults and the behavior of the system in the presence of these faults is important throughout the entire process. The artifacts generated from such analyses that are used in the certification process of such systems must be generated in a scalable way and provide accurate and important information. This project has developed and implemented the Safety Annex for AADL which provides a way to capture complex relationships between faults in a model and analyze their effects behaviorally through either compositional or monolithic analysis.

Furthermore, we propose the compositional generation of minimal cut sets to be used in the development of various artifacts used in system certification, such as FTA, probabilistic analysis, and single point of failure examinations. This generation is done through the collection of proof elements called *MIVC*s and their transformation. Lastly, we propose to use the minimal cut sets and resulting fault trees generated through the transformation algorithms to calculate the probability of the top level event, or violation of the safety property.

The timeline for the proposed work is as follows:

Jan 2017 - Sept 2017: Preliminary research into integration of fault information into `Lustre` code and prototype implementation of Safety Annex [72].

Sept 2017 - Feb 2018: In depth research into the needs of safety analysts and complex fault modeling capabilities [71].

March 2018 - March 2019: Further implementation of various types of fault behavior (dependent faults, asymmetric faults, etc). [70].

March 2019 - Sept 2019: Research into the theory of the *MinCutSet* transformation and preliminary implementation thereof [69].

Sept 2019 - Dec 2019: Drafts of theoretical paper showing *MinCutSet* transformations.

Dec 2019 - Feb 2020: Outline and prove probabilistic methods and algorithms for compositional computations.

Dec 2019 - Feb 2020: Complete the implementation of both minimal cut set generation and probabilistic computations.

Feb - March 2020: Complete the implemention of the graphical representation of hierarchical fault trees.

Dec 2019 - April 2020: Write the dissertation.

May 2020: Complete any requested edits on dissertation.

June 2020: Defend dissertation and complete final changes.

Upon completion of the proposed research, we will have the theoretical framework of compositional generation of minimal cut sets and associated probabilistic computations, and these will be implemented in the Safety Annex. The Safety Annex will provide safety analysts a way to represent both behavioral and explicit fault modeling in the context of the system model and provide valuable feedback in the development process. The results from the analysis provided by the Safety Annex will be mathematically sound and descriptive enough to use in the certification process for critical systems.

References

[1] Federal Aviation Administration (FAA). `https://www.faa.gov/regulations_policies/faa_regulations/`. Accessed: 2010-11-19.

[2] SAE International. `https://www.sae.org/`. Accessed: 2010-11-19.

[3] AIR 6110. Contiguous Aircraft/System Development Process Example, Dec. 2011.

[4] AS5506C. Architecture Analysis & Design Language (AADL), Jan. 2017.

[5] J. Bendík, E. Ghassabani, M. Whalen, and I. Černá. Online enumeration of all minimal inductive validity cores. In *International Conference on Software Engineering and Formal Methods*, pages 189–204. Springer, 2018.

[6] P. Bieber, C. Bougnol, C. Castel, J. P. Heckmann, C. Kehren, S. Metge, and C. Seguin. Safety Assessment with Altarica - Lessons Learnt Based on Two Aircraft System Studies. In *In 18th IFIP World Computer Congress*, 2004.

[7] P. Bieber, C. Bougnol, C. Castel, J.-P. H. C. Kehren, S. Metge, and C. Seguin. Safety assessment with altarica. In *Building the Information Society*, pages 505–510. Springer, 2004.

[8] P. Bieber, C. Castel, and C. Seguin. Combination of fault tree analysis and model checking for safety assessment of complex system. In *European Dependable Computing Conference*, pages 19–31. Springer, 2002.

[9] P. Bieber, J.-L. Farges, X. Pucel, L.-M. Sèjeau, and C. Seguin. Model - based safety analysis for co-assessment of operation and system safety: application to specific operations of unmanned aircraft. In *ERTS2*, 2018.

[10] B. Bittner, M. Bozzano, R. Cavada, A. Cimatti, M. Gario, A. Griggio, C. Mattarei, A. Micheli, and G. Zampedri. The xSAP Safety Analysis Platform. In *TACAS*, 2016.

[11] M. Bozzano, H. Bruintjes, A. Cimatti, J.-P. Katoen, T. Noll, and S. Tonetta. The compass 3.0 toolset (short paper). In *IMBSA 2017*, 2017.

[12] M. Bozzano, A. Cimatti, A. Griggio, and C. Mattarei. Efficient Anytime Techniques for Model-Based Safety Analysis. In *Computer Aided Verification*, 2015.

[13] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In *Computer Safety, Reliability, and Security*. Springer Berlin Heidelberg, 2009.

[14] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Yen Nguyen, T. Noll, and M. Roveri. Model-based codesign of critical embedded systems. 507, 2009.

[15] M. Bozzano, A. Cimatti, O. Lisagor, C. Mattarei, S. Mover, M. Roveri, and S. Tonetta. Symbolic Model Checking and Safety Assessment of Altarica Models. In *Science of Computer Programming*, volume 98, 2011.

[16] M. Bozzano, A. Cimatti, C. Mattarei, and S. Tonetta. Formal safety assessment via contract-based design. In *Automated Technology for Verification and Analysis*, 2014.

[17] M. Bozzano, A. Cimatti, A. F. Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta. Formal Design and Safety Analysis of AIR6110 Wheel Brake System. In *CAV 2015, Proceedings, Part I*, pages 518–535, 2015.

[18] M. Bozzano, A. Cimatti, M. Roveri, J. P. Katoen, V. Y. Nguyen, and T. Noll. Codesign of dependable systems: A component-based modeling language. In *2009 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, 2009.

[19] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic fault tree analysis for reactive systems. In *ATVA*, 2007.

[20] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic fault tree analysis for reactive systems. In *International Symposium on Automated Technology for Verification and Analysis*, pages 162–176. Springer, 2007.

[21] M. Bozzano and A. Villafiorita. Improving system reliability via model checking: The fsap/nusmv-sa safety analysis platform. In *International Conference on Computer Safety, Reliability, and Security*, pages 49–62. Springer, 2003.

[22] M. Bozzano and A. Villafiorita. *Design and Safety Assessment of Critical Systems*. Auerbach Publications, Boston, MA, USA, 1st edition, 2010.

[23] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[24] D. Chen, N. Mahmud, M. Walker, L. Feng, H. Lönn, and Y. Papadopoulos. Systems Modeling with EAST-ADL for Fault Tree Analysis through HiP-HOPS*. *IFAC Proceedings Volumes*, 46(22):91 – 96, 2013.

[25] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2000.

[26] A. Cimatti, R. Demasi, and S. Tonetta. Tightening the contract refinements of a system architecture. *Formal Methods in System Design*, 52(1):88–116, 2018.

[27] A. Cimatti and S. Tonetta. A temporal logics approach to contract-based design. In *2016 Architecture-Centric Virtual Integration (ACVI)*, pages 1–3. IEEE, 2016.

[28] D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional Verification of Architectural Models. In *NFM 2012*, volume 7226, pages 126–140, April 2012.

[29] J. De Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial intelligence*, 32(1):97–130, 1987.

[30] C. Ericson. Fault tree analysis - a history. In *Proceedings of the 17th International Systems Safety Conference*, 1999.

[31] P. Feiler and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.

[32] P. Feiler, J. Hudak, J. Delange, and D. Gluch. Architecture fault modeling and analysis with the error model annex, version 2. Technical Report CMU/SEI-2016-TR-009, Software Engineering Institute, 06 2016.

[33] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The JKind Model Checker. *CAV 2018*, 10982, 2018.

[34] A. Gainer-Dewar and P. Vera-Licona. The minimal hitting set generation problem: algorithms and computation. *SIAM Journal on Discrete Mathematics*, 31(1):63–100, 2017.

[35] E. Ghassabani, A. Gacek, and M. W. Whalen. Efficient generation of inductive validity cores for safety properties. *CoRR*, abs/1603.04276, 2016.

[36] E. Ghassabani, M. W. Whalen, and A. Gacek. Efficient generation of all minimal inductive validity cores. *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 31–38, 2017.

[37] M. Gudemann and F. Ortmeier. A framework for qualitative and quantitative formal model-based safety analysis. In *HASE 2010*, 2010.

[38] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. In *IEEE*, volume 79(9), pages 1305–1320, 1991.

[39] R. Heckel. Compositional verification of reactive systems specified by graph transformation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 138–153. Springer, 1998.

[40] E. J. Henley and H. Kumamoto. Probabilistic risk assessment and management for engineers and scientists. *IEEE Press (2nd Edition)*, 1996.

[41] P. Hönig, R. Lunde, and F. Holzapfel. Model Based Safety Analysis with smartIflow. *Information*, 8(1), 2017.

[42] P. Hönig, R. Lunde, and F. Holzapfel. Model Based Safety Analysis with smartIflow. *Information*, 8(1), 2017.

[43] A. Joshi and M. P. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *SAFECOMP*, volume 3688 of *LNCS*, page 122, 2005.

[44] A. Joshi and M. P. Heimdahl. Behavioral Fault Modeling for Model-based Safety Analysis. In *Proceedings of the 10th IEEE High Assurance Systems Engineering Symposium (HASE)*, 2007.

[45] A. Joshi, S. P. Miller, M. Whalen, and M. P. Heimdahl. A Proposal for Model-Based Safety Analysis. In *In Proceedings of 24th Digital Avionics Systems Conference*, 2005.

[46] W.-S. Jung, S.-H. Han, and J.-E. Yang. Fast bdd truncation method for efficient top event probability calculation. *Nuclear Engineering and Technology*, 40(7):571–580, 2008.

[47] J.-P. Katoen, M. Khattri, and I. S. Zapreev. A markov reward model checker. In *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, QEST '05. IEEE Computer Society, 2005.

[48] M. Konrad, S. Sheard, C. Weinstock, and W. R. Nichols. Faa research project on system complexity effects on aircraft safety: Testing the identified metrics. *White paper, Software Engineering Institute, Carnegie Mellon University*, 2016.

[49] M. Kwiatkowska, G. Norman, and D. Parker. PRiSM 4.0: Verification of Probabilistic Real-time Systems. In *In Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of LNCS, 2011.

[50] M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible mus enumeration. *Constraints*, 21(2):223–250, 2016.

[51] O. Lisagor, T. Kelly, and R. Niu. Model-based safety assessment: Review of the discipline and its challenges. In *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, 2011.

[52] V. Matuzas and S. Contini. Dynamic labelling of bdd and zbdd for efficient non-coherent fault tree analysis. *Reliability Engineering & System Safety*, 144:183–192, 2015.

[53] C. McMillan, A. Crapo, M. Durling, M. Li, A. Moitra, P. Manolios, M. Stephens, and D. Russell. Increasing development assurance for system and software development with validation and verification using assert^{TM}. Technical report, SAE Technical Paper, 2019.

[54] S.-i. Minato. Zero-suppressed bdds and their applications. *International Journal on Software Tools for Technology Transfer*, 3(2):156–170, 2001.

[55] MRMC: Markov Rewards Model Checker. http://wwwhome.cs.utwente.nl/ zapreevis/mrmc/.

[56] K. Murakami and T. Uno. Efficient algorithms for dualizing large-scale hypergraphs. In *2013 Proceedings of the Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 1–13. SIAM, 2013.

[57] NuSMV Model Checker. http://nusmv.itc.it.

[58] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, pages 123–144. Springer, 1985.

[59] T. Prosvirnova, M. Batteux, P.-A. Brameret, A. Cherfi, T. Friedlhuber, J.-M. Roussel, and A. Rauzy. The AltaRica 3.0 Project for Model-Based Safety Assessment. *IFAC*, 46(22), 2013.

[60] RAT: Requirements Analysis Tool. http://rat.itc.it.

[61] M. Rausand and A. Høyland. *System reliability theory: models, statistical methods, and applications*, volume 396. John Wiley & Sons, 2003.

[62] A. Rauzy. New algorithms for fault trees analysis. *Reliability Engineering & System Safety*, 40(3):203–211, 1993.

[63] A. Rauzy. Binary decision diagrams for reliability studies. In *Handbook of performability engineering*, pages 381–396. Springer, 2008.

[64] R. Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1):57–95, 1987.

[65] E. Ruijters and M. Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer science review*, 15-16:29–62, 5 2015.

[66] SAE ARP 4761. Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

[67] SAE ARP4754A. Guidelines for Development of Civil Aircraft and Systems, December 2010.

[68] A. Schäfer. Combining real-time model-checking and fault tree analysis. In *International Symposium of Formal Methods Europe*, pages 522–541. Springer, 2003.

[69] D. Stewart, J. Liu, M. Heimdahl, M. Whalen, D. Cofer, and M. Peterson. Architectural modeling and analysis for safety engineering (AMASE), NASA final report. `https://github.com/loonwerks/AMASE/tree/master/doc/AMASE_Final_Report_2019`, 2019.

[70] D. Stewart, J. Liu, M. Heimdahl, M. Whalen, D. Cofer, and M. Peterson. The Safety Annex for Architecture Analysis and Design Language. In *10th Edition European Congress Embedded Real Time Systems*, to appear 2020.

[71] D. Stewart, J. Liu, M. Whalen, D. Cofer, and M. Peterson. Safety Annex for Architecture Analysis Design and Analysis Language. Technical Report 18-007, University of Minnesota, March 2018.

[72] D. Stewart, M. Whalen, D. Cofer, and M. P. Heimdahl. Architectural Modeling and Analysis for Safety Engineering. In *IMBSA 2017*, pages 97–111, 2017.

[73] W. Vesely, F. Goldberg, N. Roberts, and D. Haasl. Fault tree handbook-nureg-0492209. Technical report, Technical report, US Nuclear Regulatory Commission, 1981.

[74] M. Volk, S. Junges, and J.-P. Katoen. Fast dynamic fault tree analysis by model checking techniques. *IEEE Transactions on Industrial Informatics*, 14(1):370–379, 2017.