Architectural Modeling and Analysis for Safety Engineering

A DISSERTATION
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

Mats P. E. Heimdahl and Michael W. Whalen

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 Heimdahl 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

List of Figures

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 [86, 87]. 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 [14, 53, 55, 60, 68]. 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 [14, 19, 58]. 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 [**?**, 12, 18, 53]. 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 **??** 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

Preliminaries and Related Work

2.1 Safety Critical Systems Development

As the capabilities of technology grows, so does the complexity and capabilities of mechanical and electrical systems. Many of these systems are safety critical; the loss of correct functioning leads to loss of life, substantial material or environmental damage, or large monetary losses. The development of such complex systems requires a process with clearly defined design and implementation phases which are subdivided into several sub-processes and phases. Certain sets of analyses are required for each of the phases and when the analyses provide satisfactory outcomes, the process transitions into the next phase.

In general, each field relies on various interpretations of the development process. In the field of aerospace technologies, the Aerospace Recommended Practice (ARP) is what is commonly used. The Society of Automotive Engineers (SAE) is an association of engineers and professionals devoted to the standards that guide the development of transportation systems [86,87].

2.1.1 The V Model

The development of safety critical systems is guided by the V model process as defined in ARP4754 [87]. The V model relates steps of the design phase with a post-implementation phase. It describes how the requirements are produced in the design phase and then how those requirements are verified against the implementation in the post-implementation phase. The left side of the V describes the requirements, architecture, and expected component behavior of the system (see Figure 2.1). The right side of the V describes the evaluation of the system implementation in light of the requirements.

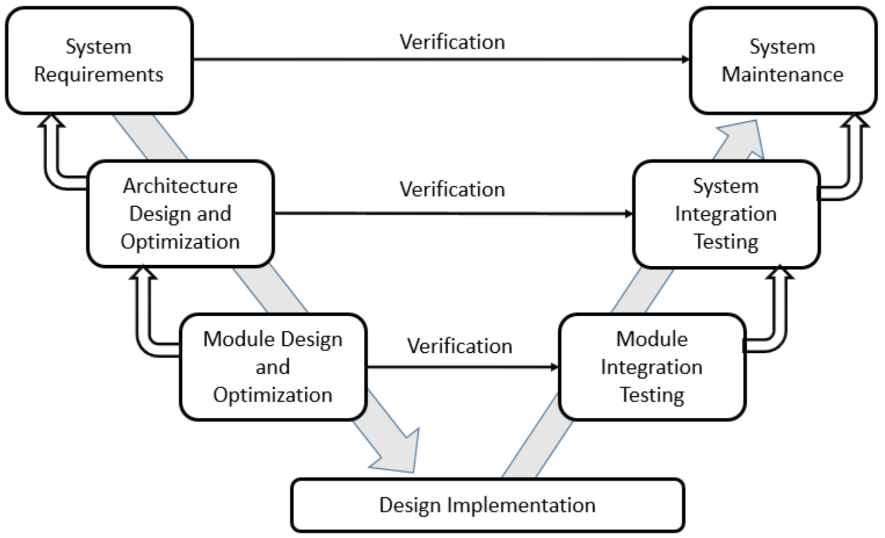

System Requirements
Verification
System Maintenance
Architecture Design and Optimization
Verification
System Integration Testing
Module Design and Optimization
Verification
Module Integration Testing
Design Implementation

Figure 2.1: The V Model in System Development

2.1.2 Traditional Safety Assessment Process

ARP4754A, the Guidelines for Development of Civil Aircraft and Systems [87], provides guidance on applying development assurance at each hierarchical level throughout the development life cycle of highly-integrated/complex aircraft systems. It has been recognized by the Federal Aviation Administration (FAA) as an acceptable method to establish the assurance process. The safety assessment process is a starting point at each hierarchical level of the development life cycle and is tightly coupled with the system development and verification processes. It is used to show compliance with certification requirements and for meeting a company's internal safety standards.

The safety assessment shown in Figure 2.2 integrates each phase of the V model with analyses specific to system hazards and their severity. It also shows how these hazards should be addressed within the design phase. The safety assessment proess is defined in ARP4754A by the following phases:

Functional Hazard Assessment (FHA) examines the functions of the system to identify potential functional failures and classifies the potential hazards associated with them. This includes identification of failure conditions, identifying the effects of those failures, classification of each failure condition, and assignment to safety objectives.

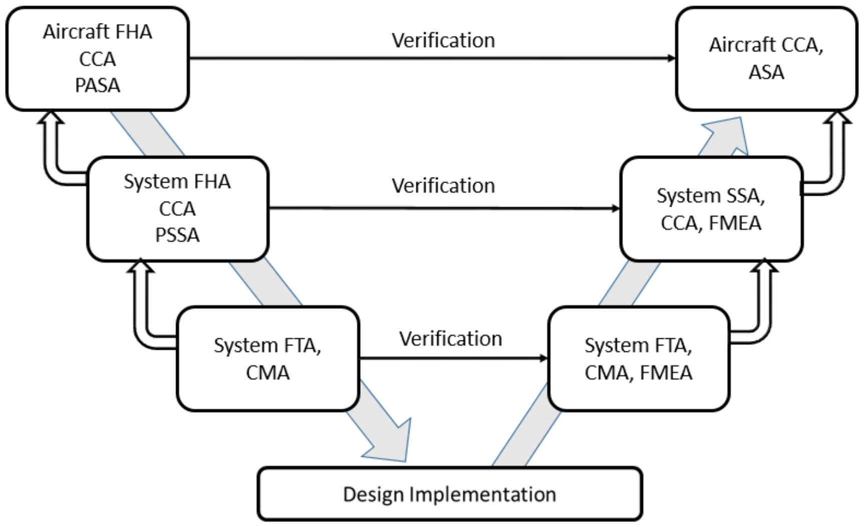


Figure 2.2: The V Model in Safety Assessment

Common Cause Analysis (CCA) verifies and establishes physical and functional separation, isolation, and independence requirements between subsystems and verifies that these requirements have been met.

Preliminary Aircraft Safety Assessment (PASA) establishes aircraft safety requirements and provide a preliminary indication that the aircraft can meet those safety requirements.

Preliminary System Safety Assessment (PSSA) examines the proposed architecture(s) to determine how failures could cause the failure conditions determined by the FHA. The objective is to complete the safety requirements of an aircraft or system and show that the proposed system architecture satisfies the safety requirements. The PSSA is an iterative process that is performed at multiple stages of system development.

Fault Tree Analysis (FTA) is performed to find combinations of faults that lead to the violation of a safety requirement. The fault tree itself shows the logical relation between the sets of faults and the violation of a safety requirement.

Common Mode Analysis (CMA) analyzes designs and implementations for elements that may defeat the redundancy or independence of functions within the design, i.e. if elements are shown as independent in FTA, make sure they are truly independent in the system under consideration.

Failure Modes and Effect Analysis (FMEA) aims at finding the causality relationship between sets of faults, intermediate events, and undesired states in the system. Usually this is represented in tabular form and called an *FMEA table*.

Aircraft Safety Assessment (ASA)/System Safety Assessment (SSA) verifies that the system (or aircraft), as implemented, meets the safety requirements specified by the PSSA.

2.2 Model Based Safety Assessment

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.

2.2.1 Suggested 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; this is shown in Figure 2.3 and 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 check 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

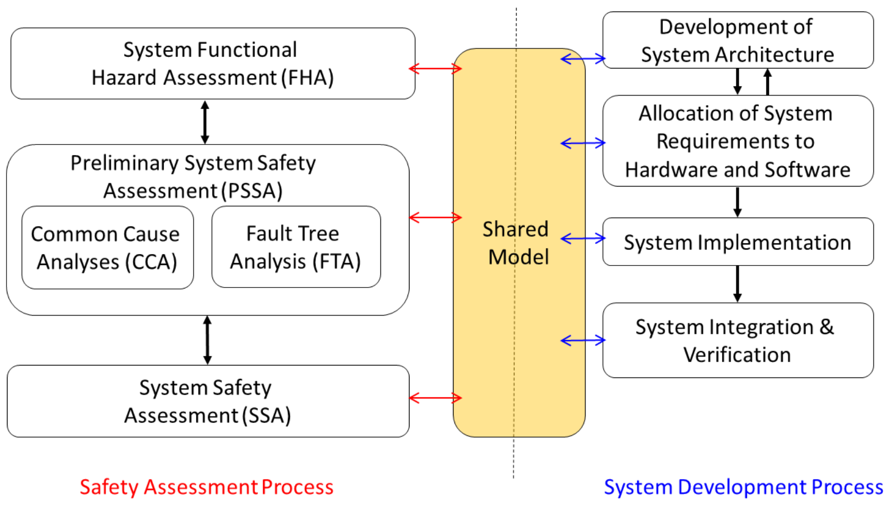

System Functional Hazard Assessment (FHA)
Preliminary System Safety Assessment (PSSA)
Common Cause Analyses (CCA)
Fault Tree Analysis (FTA)
System Safety Assessment (SSA)
Shared Model
Development of System Architecture
Allocation of System Requirements to Hardware and Software
System Implementation
System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

Figure 2.3: Use of the Shared System/Safety Model in the ARP4754A Safety Assessment Process

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), then the tool produces counterexamples or minimal sets 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.

2.3 Formal Methods of Validation and Verification

As the complexity of systems increase, the cost of development and validation consumes more time and resources than ever before; nevertheless, these processes are vital in safety critical systems when the loss of functionality of the system can result in loss of life. Authorities have put in place various thresholds for the likelihood of such events and it is the responsibility of the system developers to show that this is incredibly unlikely to occur [43]. Utilizing the recent advancements in automated formal verification within the validation process has become essential to the certification of critical systems [1, 64, 74]. This section provides a summary of the formal method techniques that are commonly used in the system development and safety assessment processes.

2.3.1 Formal Validation and Verification

Formal validation and verification is a proof-based methodology used to assess the correctness of requirements, system design, and implementation. In the past, this has been performed through manual means, but with the advancement in automated theorem proving and other formal methods, automated formal analyses not only guarantees a higher degree of confidence, but also reduces the time (and thus cost) of carrying out the proofs of correctness. Techniques used in formal validation and verification include automated theorem proving, model checking, and abstract interpretation.

Formal Specification

Formal specification process translates the informal system requirements into a mathematical logic to determine if the system design is correct [54]. This process guarantees an unambiguous description of the requirements which is not possible when using an informal natural language. This formal definition of system requirements includes the system design and its expected behavior as well as the assumptions on environment. A design or implementation can never be considered correct in isolation; it is only correct with respect to the specifications. The expected behavior, system design, and environmental assumptions change and are refined as the system goes through the various stages of development.

Formal Verification

Formal verification is the use of proof methods to show that given the environmental assumptions stated in the formal specification, the formal design of the system meets the requirements. The problem can be reduced to that of property checking: given a program P and a specific property, does the program satisfy the given property [46]. This is an undecidable problem because a program can be represented as an infinite state space. The problem is that of finding a finite set of predicates that support the specified property over an infinite state space. This is an undecidable problem; any algorithm searching for a solution to this problem may not terminate [34]. The approaches used to provide these proofs are usually deductive methods or an exhaustive exploration of the model known as model checking.

Model checking was introduced in the early 1980's and consists of exploring the states and transitions of a model [33, 77]. By representing the system abstractly, the infinite state space is reduced to a finite model. This addresses the undecidability factor [40]. The proofs are generated over an abstract mathematical model of the system, such as finite state machines, labeled transition systems, or timed automata. It takes as input a model of a system and the properties written in propositional temporal logic, then explores the entire state space of the system to determine if the model violates the properties [34,47]. In recent years, model checking takes advantage of abstraction techniques specific to a domain to consider multiple states or transitions in a single operation; this lessens computation time considerably [40]. Nevertheless, the biggest limiting factor of model checking is scalability and much of the recent research in this area attempts to address this problem [34].

Deductive methods of verification consists of generating proof obligations from the specifications of the system and using these obligations in a theorem prover setting. Automated theorem provers have the main objective to show that some statement (conjecture) is a logical consequence of other statements (the axioms and hypotheses). The rules of inference are given as are the set of axioms and hypotheses [40, 46]. Deductive methods of verification include automated theorem provers (e.g., Coq [37], Isabelle [71]) and satisfiability modulo theories (e.g., SMTInterpol [30], Z3 [36], Yices [41]).

2.4 Satisfiability

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). For example, the proposition $a \wedge b$ is satisfiable; when a and b are assigned to $true$, the formula is satisfied. On the other hand, the proposition $a \wedge \neg a$ is unsatisfiable; no such assignment can be found to satisfy both a and $\neg a$. SAT solvers in work over a constraint system of propositional logic to determine satisfiability. Satisfiability Modulo Theories (SMT) solvers also address the SAT problem, but can work over propositional logic or predicate logic with quantifiers. An SMT solver works over a conjunction of literals, as is the case with SAT solvers, but the literals can be expressed as predicates over non-boolean variables, such as $x > 0$. A boolean literal can be satisfied with a finite number of possible assignments; this is not always the case with SMT formula.

2.4.1 UNSAT Cores and Minimal Unsatisfiable Subsets

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 [66]. Given a constraint system, we require some method of determining, for any subset $S \subseteq C$, whether S is *satisfiable* (SAT) or *unsatisfiable* (UNSAT). Given a constraint system C, there are certain subsets of C that are of interest in terms of satisfiability.

For a given unsatisfiable problem, SAT solvers (and SMT solvers) attempt to provide proof of unsatisfiability by providing a subset of UNSAT clauses known as *UNSAT cores*. In general, this is useful information to have regarding the constraint system in question.

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

Definition 2. *UNSAT core: Let C be a finite set of constraints and $U \subseteq C$ an unsatisfiable subset. A constraint $c \in U$ is an UNSAT core for U if $U \setminus \{c\}$ is satisfiable. A set of all unsatisfiability cores of U constitute an MUS for C.*

Intuitively, an MUS is the minimal explanation of the constraint systems infeasability and the UNSAT cores are the building blocks of the MUS. In recent years, a number of efficient algorithms have been introduced to find MUSs [65] and most of them focus on finding a single such subset [5–7]. More recently, algorithms have been introduced that can find all such minimal unsatisfiable subsets [8, 51, 52].

2.4.2 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 [51]. 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`) [8, 52].

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 (MUSs) 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.

2.5 Artifacts, Data Structures, and Other Formalizations

The assessment processes of critical system development produce important artifacts that are used together for certification of the system, but those of importance to this thesis are *Fault Trees* and associated sets called *Minimal Cut Sets*. For this reason, more information is provided in this section on these artifacts.

2.5.1 Fault Trees 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, 86, 87].

A Fault Tree (FT) is a directed acyclic graph whose leaves model component failures and whose gates model failure propagation [85]. 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 [42]. 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 [85].

Figure 2.4 shows a simple example of a fault tree based on SAE ARP4761 [86]. 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

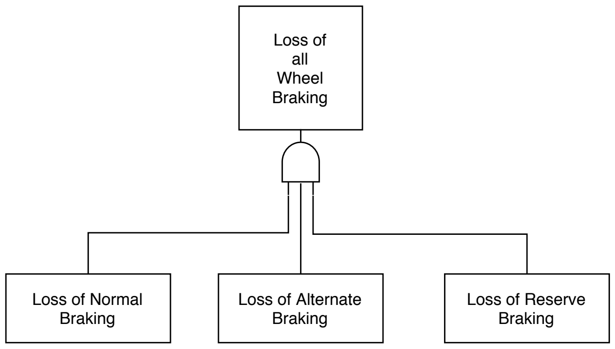


Figure 2.4: A simple fault tree

system. Each gate has one output and one or more inputs. In Figure 2.4, 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 [42, 85].

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 [85].

2.5.2 Failure Mode and Effects Analysis

Failure Mode and Effects Analysis (FMEA) was one of the first systematic ways of performing dependability analysis and is used throughout the safety critical industries [16, 79]. FMEA provides a structured way to list possible failures and their consequences systemwide. If probabilities of failures are known, quantitative analysis can be performed to estimate system reliability and to assign critical significance to potential failure modes or system components [97]. Performing FMEA is often the first step in the fault tree construction, for it shows possible component failures and hence basic events [85]. Typically, the failure modes of the components at a given level are considered; the objective it to identify the effects of the failure modes at that level - and usually higher levels - of the design. The FMEA results are often presented in tabular form (FMEA Table) and as seen in Figure **??** make figure. FMEA tables vary in form, but almost always include failure mode definitions, the operational mode in which the failure can occur, and possible causes of the failure [24].

2.5.3 Ordered Binary Decision Diagrams

A Binary Decision Diagram (BDD) is a data structure used to encode Boolean formulae. As

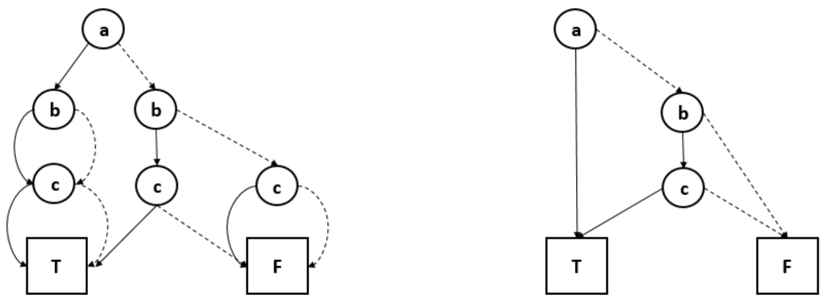


Figure 2.5: Binary Decision Diagrams of the Formula $a \lor (b \land c)$

shown in Figure 2.5, it is a rooted, directed, acyclic graph with internal decision nodes and two terminal nodes (*true* and *false*). Each of the decision nodes is labeled with a Boolean variable and has two child nodes, low child and high child. The edge from a node to its low child represents the assignment of *false*, likewise the edge to the high child represents the assignment of *true*. The BDD is called *ordered* if different variables appear in the same order on all paths from the root. Intuitively, following a path from the root to the *true* terminal node represents a valid assignment to the Boolean formula (invalid in the case of ending on the *false* terminal node).

BDDs are reduced by the removal of isomorphic subgraphs. The BDD shown on the right of Figure 2.5 is the reduced form of the BDD on the left.

2.5.4 State Machines and Their Verification

A finite state machine (or finite state automaton) is a mathematical model of computation and consists of states, represented by nodes, and transitions between them, represented by directed edges. The change from one state to another is called a *transition*. The abstract machine can be in exactly one of a finite number of states at a time (hence, finite). An example of a finite state machine is shown in Figure **??** make figure.

An infinite state machine has much more power in representation due to the ability to deal with infinite states. In domains like model checking, this is required since many of the variables used are from infinite domains (e.g., real numbers, integers). The expressive capabilities of set notation and predicate logic allow finite strings to represent these infinite states. For example, the infinite set of integers greater than zero is described succintly as: $\{x \in \mathbb{Z} : x > 0\}$.

Abstracting a program or system with respect to a state machine is great, but without being able to reason about that abstraction, it is nothing more than slightly interesting. Information commonly required of a state machine representation is if a given state is *reachable*. In other words, reachability determines if is there a sequence of transitions that can lead to a given state.

Model checkers often utilize the expressive power of state machines to verify specifications. One such example important to this thesis is JKind [49], an infinite state model checker. Verification of the program is based on *k-induction* and property directed reachability using a back-end SMT solver, e.g., Z3 [36], SMTInterpol [30].

k-induction

The k-induction method was introduced as a technique for SAT-based verification of finite and infinite state transition systems [89]. Let $I(s)$ and $T(s, s_0)$ be formulae encoding the initial states and transition relation for a system over sets of propositional state variables s and s_0. Additionally, let $P(s)$ be a formula that represents the states satisfying a safety property and k a positive integer. To prove the safety property P by k-induction, there are two steps, the base case and the induction case. The base case must show that P holds in all states reachable from an initial state within k steps, or transitions. More formally, the base case must show that the following formula is unsatisfiable:

$$I(s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge T(s_{k-1}, s_k) \wedge (\overline{P(s_1)} \vee \cdots \vee \overline{P(s_k)})$$

The induction step must show that whenever P holds in k consecutive states, $s_1, \ldots, s_k$, P also holds in the next state s_{k+1} of the system. This is done by showing that the step case formula is unsatisfiable:

$$P(s_1) \wedge T(s_1, s_2) \wedge \cdots \wedge P(s_k) \wedge T(s_k, s_{k+1}) \wedge \overline{P(s_{k+1})}$$

Ever since k-induction was introduced for the purpose of verification of state machines, various methods came about of either combining these two proof steps into one [38] or performing them in parallel [61].

Said a fellow in liquor production
“I’ve a still of ingenious construction
the alcohol boils
through old magnet coils
I’ve dubbed it my Proof by Induction”

2.5.5 Transition Systems

Informally, a transition system is a model of states and transitions between them. Intuitively, finite automata are like transition systems with additional constraints, for instance, defined start and final states. Transition systems are directed graphs with nodes representing reachable states and edges representing transitions between them. They may also be defined with a mapping function that assigns labels to each node; in the context of model checking, these labels are often properties which must hold in the corresponding state.

Labeled transition systems are used extensively in model checking and will be mentioned within that context in later sections. There is much that could be said about transition systems, but for the purpose of this body of work, it is unnecessary. More information about transition systems and their relation to safety analysis and model checking can be found in the comprehensive book written by Bozzano and Villafiorita, Design and Safety Assessment of Critical Systems [24].

2.6 Formal Methods in Safety Analysis: A Brief History and the State of the Practice

Safety analysis has traditionally been performed manually, but with the rise of model checking and the improvement of its capabilities, the world of safety analysis began to see its powerful benefits [24, 25, 35, 54, 67]. There arose multiple ways of viewing the system and fault models, various ways of automating the capture of safety pertinant information, and a number of tools that addressed various issues that arose. In this section, we discuss the state of the practice and how formal methods has been applied in the domain of safety assessment.

2.6.1 Fault Tree Analysis and Binary Decision Diagrams

Since the early days of safety engineering, fault tree analysis has been a primary method of determining safety of a system and showing the behavior of the system (with respect to its requirements) in the presence of faults [85, 98]. Fault tree analysis requires one to explore the faults of the system and their effects on system behavior to determine minimal fault configurations (minimal cut sets) that cause violation of requirements. From the beginning of fault tree analysis in the '60's, algorithms worked directly with the fault tree structure to produce MinCutSets [48, 88]. As the years progressed, it was clear that this approach could not sufficiently address the problem of computation time. In 1993, Rauzy et al. developed a new approach that converted the fault tree structure into a binary decision diagram (BDD) [80]. This was a natural way to reduce the Boolean formula into something far more computationally efficient and reduceable to even simpler forms. Numerous algorithms were developed to perform variable ordering and minimization of the BDD; this resulted in better computation of MinCutSets and began the process of automating a complex manual safety analysis task [27, 81–83, 91]. BDDs are still commonly used to perform quantitative and qualitative fault tree analysis [4, 50, 57].

2.6.2 Model Checking and Model Based Safety Analysis

From the beginnings of model checking, there was a slow increase in its application to the domain of safety analysis, but a few research groups contributed immensely to this branch of study. Separately, these researchers began to contribute to safety analysis through the use of model checking starting in the '90's and still contributing today (e.g., [29, 31, 84, 90].

One of the main methods used was to abstract the system into a formal transition system; this provided a means of defining a precise mathematical model of the system and simplifying mathematical operations through the use of abstraction techniques on the transition system. This helped to shrink the entire state space into something more digestible by computational techniques [40].

In the early 2000's, model based safety assessment began to make an appearance in the literature [24, 58–60]. This applied model checking and model based system development to safety analysis at the same time. 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 *explicit propagation*, or through existing behavioral modeling, which we call *failure effect modeling* (FEM), or *implicit propagation*. 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*.

This literature overview is not a complete account of all safety analysis model checking tools available either in industry or research, but only highlights some of the most influential safety assessment methods and tools currently available.

AltaRica

AltaRica was one of the first model checking tools specifically aimed at safety analysis of critical systems. The first iteration of AltaRica (1.0) performed over a transition system of the model, used dataflow (*causal*) semantics, and could capture the hierarchy of a system [90]. The key idea was that this transition system (more specifically constraint automata) could be compiled into Boolean formulae and transformed into a BDD [73]. The literature for performing fault tree analysis over BDDs was rich with algorithms; this was how much of the safety analysis artifacts were generated. The dataflow dialect (AltaRica 1.0) has substantial tool support, including the commercial Cecilia OCAS tool from Dassault [10]. For this dialect the safety assessment, fault tree generation, and functional verification can be performed with the aid of NuSMV model checking [18].

The most recent language update (AltaRica 3.0) uses non-causal semantics [75,76]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [9]. AltaRica 3.0 has support for simulation and Markov model generation through the OpenAltaRica (www.openaltarica.fr) tool suite; it is a *FEM*-based, *purpose-built*, *monolithic* safety analysis language.

FSAP, xSAP, and COMPASS

The Formal Safety Analysis Platform (FSAP) was introduced in 2003 [23] and supported failure mode definitions, safety requirements in temporal logic formulae, automated fault tree construction, and counterexample traces. The platform used NuSMV, a BDD-based model checker [32]. The system model, written in NuSMV, and the fault model, developed graphically in FSAP, are together translated ito a finite state machine and eventually into a BDD; fault tree analysis is performed using algorithms implemented in NuSMV.

By 2016, the researchers that developed FSAP (Foundation Bruno Kessler, FBK) released a similar tool called xSAP [12]. xSAP extends FSAP in many ways: xSAP can handle infinite state machines, it is textual language rather than graphical, allows for richer fault modeling and definitions, and implements more than just BDD computations (e.g., SAT- and SMT-based routines). xSAP was integrated into the COMPASS toolsuite to take advantage of the algorithms it supports.

COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [15] is a mixed *FLM/FEM*-based, *causal compositional* tool suite that uses the SLIM language, which is based on a subset of the Architecture Analysis and Design Language (AADL), for its input models [17,21]. 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 [32, 72], MRMC (Markov Reward Model Checker) [62,70], and RAT (Requirements Analysis Tool) [78]. The safety analysis tool xSAP [12] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [13]. 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

SmartIFlow [55, 56] 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” [55].

SAML

The Safety Analysis and Modeling Language (SAML) [53] is a *FEM*-based, *purpose-built*, *monolithic causal* safety analysis language that was developed in 2010. 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 [32], PRISM (Probabilistic Symbolic Model Checker) [63], or the MRMC probabilistic model checker [62].

Error Model Annex for AADL

The SAE (Society of Automotive Engineers) released the aerospace standard AS5506, named Architecture Analysis and Design Language (AADL), which is a mature industry-standard for embedded systems and has proved to be efficient for architecture modeling [2,69]. AADL supports safety analysis by adding EMA (Error Model Annex) as an extension to the language. EMA allows the user to annotate system hardware and software architectures with hazard, error propagation, failure modes and effects due to failures. Around 2016, Version 2 of the Error Model Annex was released (EMV2) [45]. EMV2 is an *FLM*-based *ESM* approach. The faults and error propagations are explicitly defined and the fault tree analysis is performed by traversing propagation paths in reverse to find the original fault that caused the problem [44].

Chapter 3

Compositional Minimal Cut Set Generation

??

Chapter 4

Fault Modeling and the Safety Annex

?? This chapter covers modeling techniques/needs, the safety annex, and its capabilities.

4.1 Component Fault Modeling

The usage of the terms error, failure, and fault are defined in ARP4754A and are described here for ease of understanding [87]. 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 terminology used in EMV2 differs slightly for an error: an error is a corrupted state caused by a fault. The error propagates through a system and can manifest as a failure. In this report, we use the ARP4754A terminology with the added definition of *error propagation* as used in EMV2. An error is a mistake made in design or code and an error propagation is the propagation of the corrupted state caused by an active fault.

The Safety Annex is used to add possible faulty behaviors to a component model. 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. A library of common fault nodes has been written and is available in the project GitHub repository [94]. Examples of such faults include valves being stuck open or closed, output of a software component being nondeterministic, or power being cut off. When the fault analysis requires fault definitions that are more complex, these nodes can easily be written and used in the model.

When a fault is activated by its specified triggering conditions, it modifies the output of the component. This faulty behavior may violate the contracts of other components in the system, including assumptions of downstream components. The impact of a fault is computed by the AGREE model checker when the safety analysis is run on the fault model.

The majority of faults that are connected to outputs of components are known as *symmetric*. That is, whatever components receive this faulty output will receive the same faulty output value. Thus, this output is seen symmetrically. An alternative fault type is *asymmetric*. This pertains to a component with a 1-n output: one output which is sent to many receiving components. This fault can present itself differently to the receiving components. For instance, in a boolean setting, one component might see a true value and the rest may see false. This is also possible to model using the keyword *asymmetric*. For more information on fault definitions and modeling possibilities, we refer readers to the Safety Annex Users Guide [94].

As an illustration of fault modeling using the Safety Annex, we look at one of the components important to the inadvertent braking property: the brake pedal. When the mechanical pedal is pressed, a sensor reads this information and passes an electronic signal to the BSCU which then commands hydraulic pressure to the wheels.

Figure 4.1 shows the AADL pedal sensor component with a contract for its nominal behavior. The sensor has only one input, the mechanical pedal position, and one output, the electrical pedal position. A property that governs the behavior of the component is that the mechanical position should always equal the electronic position. (The expression *true* $\rightarrow$ *property* in AGREE is true in the initial state and then afterwards it is only true if property holds.)

```
system SensorPedalPosition
  features
       -- Input ports for subcomponent
       mech_pedal_pos : in data port Base_Types::Boolean;
       elec_pedal_pos : in data port Base_Types::Boolean;

  -- Behavioral contracts for subcomponent
  annex agree {**

     guarantee "Mechanical and electrical pedal position is equivalent" :
       true -> (mech_pedal_position = elec_pedal_position;
  };
```

Figure 4.1: An AADL System Type: The Pedal Sensor

One possible failure for this sensor is inversion of its output value. This fault can be triggered with probability 5.0×10^{-6} as described in AIR6110 (in reality, the component failure probability is collected from hardware specification sheets). The Safety Annex definition for this fault is shown in Figure 4.2. Fault behavior is defined through the use of a fault node called *inverted_fail*. When the fault is triggered, the nominal output of the component (*elec_pedal_position*) is replaced with its failure value (*val_out*).

```
annex safety {**
  fault SensorPedalPosition_ErroneousData "Inverted boolean fault" : faults.inverted_fail {
       inputs: val_in <- elec_pedal_position;
       outputs: elec_pedal_position <- val_out;
       probability: 5.0E-6 ;
       duration: permanent;
  }
};
```

Figure 4.2: The Safety Annex for the Pedal Sensor

The WBS fault model expressed in the Safety Annex contains a total of 33 different fault types and 141 fault instances. The large number of fault instances is due to the redundancy in the system design and its replication to control 8 wheels.

4.2 Error Propagation

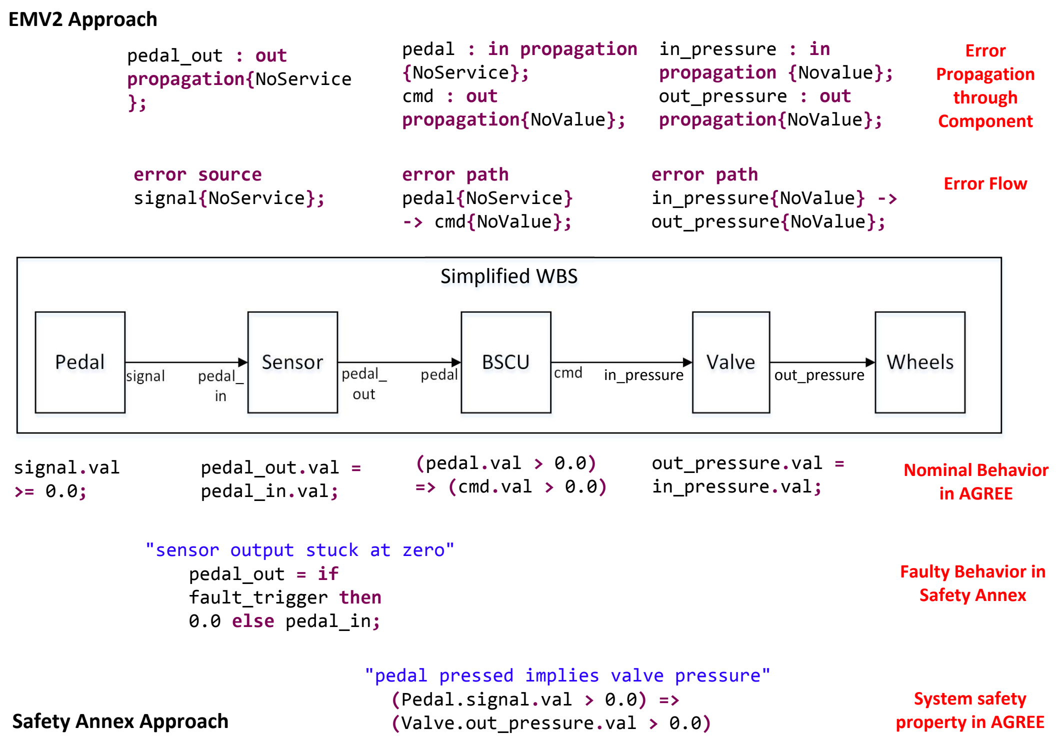
4.2.1 Implicit Propagation

In the Safety Annex approach, faults are captured as faulty behaviors that augment the system behavioral model in AGREE contracts. No explicit error propagation is necessary since the faulty behavior itself propagates through the system just as in the nominal system model. The effects of any triggered fault are manifested through analysis of the AGREE contracts.

On the contrary, in the AADL Error Model Annex, Version 2 (EMV2) [45] approach, all errors must be explicitly propagated through each component (by applying fault types on each of the output ports) in order for a component to have an impact on the rest of the system. To illustrate the key differences between implicit error propagation provided in the Safety Annex and the explicit error propagation provided in EMV2, we use a simplified behavioral flow from the WBS example using code fragments from EMV2, AGREE, and the Safety Annex.

In this simplified WBS system, the physical signal from the Pedal component is detected by the Sensor and the pedal position value is passed to the Braking System Control Unit (BSCU) components. The BSCU generates a pressure command to the Valve component which applies hydraulic brake pressure to the Wheels.

In the EMV2 approach (top half of Figure 4.3), the “NoService” fault is explicitly propagated through all of the components. These fault types are essentially tokens that do not capture



```
pedal_out : out
propagation{NoService
};
```

```
pedal : in propagation
{NoService};
cmd : out
propagation{NoValue};
```

```
in_pressure : in
propagation {Novalue};
out_pressure : out
propagation{NoValue};
```

```
error source
signal{NoService};
```

```
error path
pedal{NoService}
-> cmd{NoValue};
```

```
error path
in_pressure{NoValue} ->
out_pressure{NoValue};
```

```
signal.val
>= 0.0;
```

```
pedal_out.val =
pedal_in.val;
```

```
(pedal.val > 0.0)
=> (cmd.val > 0.0)
```

```
out_pressure.val =
in_pressure.val;
```

```
"sensor output stuck at zero"
    pedal_out = if
    fault_trigger then
    0.0 else pedal_in;
```

```
"pedal pressed implies valve pressure"
  (Pedal.signal.val > 0.0) =>
  (Valve.out_pressure.val > 0.0)
```

Figure 4.3: Differences between Safety Annex and EMV2

any analyzable behavior. At the system level, analysis tools supporting the EMV2 annex can aggregate the propagation information from different components to compose an overall fault flow diagram or fault tree.

When a fault is triggered in the Safety Annex (bottom half of Figure 4.3), the output behavior of the Sensor component is modified. In this case the result is a “stuck at zero” error. The behavior of the BSCU receives a zero input and proceeds as if the pedal has not been pressed. This will cause the top level system contract to fail: *pedal pressed implies brake pressure output is positive*.

4.2.2 Explicit Propagation

Failures in hardware (HW) components can trigger behavioral faults in the system components that depend on them. For example, a CPU Failure may trigger faulty behavior in the threads bound to that CPU. In addition, a failure in one HW component may trigger failure in other HW components located nearby, such as overheating, fire, or 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. The explicit propagation to non behavioral faults is similar to that provided in EMV2.

To better model faults at the system level dependent on HW failures, a fault model element is introduced called a *hardware fault*. Users are not required to specify behavioral effects for the HW faults, nor are data ports necessary on which to apply the fault definition. An example of a model component fault declaration is shown below:

```
HW_fault Pump_HW_Fault "Colocated pump failure": {
    probability: 1.0E-5;
    duration: permanent;
}
```

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 dependent faults. This allows a simple propagation from the faulty HW component to the SW components that rely on it, affecting the behavior on the outputs of the affected SW components.

In the WBS example, assume that both the green and blue hydraulic pumps are located in the same compartment in the aircraft and an explosion in this compartment rendered both pumps inoperable. The HW fault definition can be modeled first in the green hydraulic pump component as shown in Figure **??**. The activation of this fault triggers the activation of related faults as seen in the *propagate_to* statement shown below. Notice that these pumps need not be connected through a data port in order to specify this propagation.

```
annex safety{**

   analyze : probability 1.0E-7
   propagate_from:
     {Pump_HW_Fault@phys_sys.green_hyd_pump} to {HydPump_FailedOff@phys_sys.blue_hyd_pump};

**};
```

The fault dependencies are specified in the system implementation where the system configuration that causes the dependencies becomes clear (e.g., binding between SW and HW components, co-location of HW components).

4.3 Fault Analysis Statements

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:

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

or that the only faults to be considered are those whose probability of simultaneous occurrence is above some probability threshold:

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

Tying back to the fault tree analysis in traditional safety analysis, the former is analogous to restricting the cutsets to a specified maximum number of terms, and the latter is analogous to restricting the cutsets to only those whose probability is above some set value. In the former case, we assert that the sum of the true *fault__trigger* variables is at or below some 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__trigger* variables. With the introduction of dependent faults, 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).

4.4 Asymmetric Fault Modeling

A *asymmetric* or *Byzantine* fault is a fault that presents different symptoms to different observers [39]. In our modeling environment, asymmetric faults may be associated with a component that has a 1-n output to multiple other components. In this configuration, a *symmetric* fault will result in all destination components seeing the same faulty value from the source component. To capture the behavior of asymmetric faults ("different symptoms to different observers"), it was necessary to extend our fault modeling mechanism in AADL.

Implementation of Asymmetric Faults To illustrate our implementation of asymmetric faults, assume a source component A has a 1-n output connected to four destination components (B-E) as shown in Figure 4.4 under "Nominal System." If a symmetric fault was present on this output, all four connected components would see the same faulty behavior. An asymmetric fault should be able to present arbitrarily different values to the connected components.

To this end, "communication nodes" are inserted on each connection from component A to components B, C, D, and E (shown in Figure 4.4 under "Fault Model Architecture." From the users perspective, the asymmetric fault definition is associated with component A's output and the architecture of the model is unchanged from the nominal model architecture. Behind the scenes, these communication nodes are created to facilitate potentially different fault activations on each of these connections. The fault definition used on the output of component A will be inserted into each of these communication nodes as shown by the red circles at the communication node output in Figure 4.4.

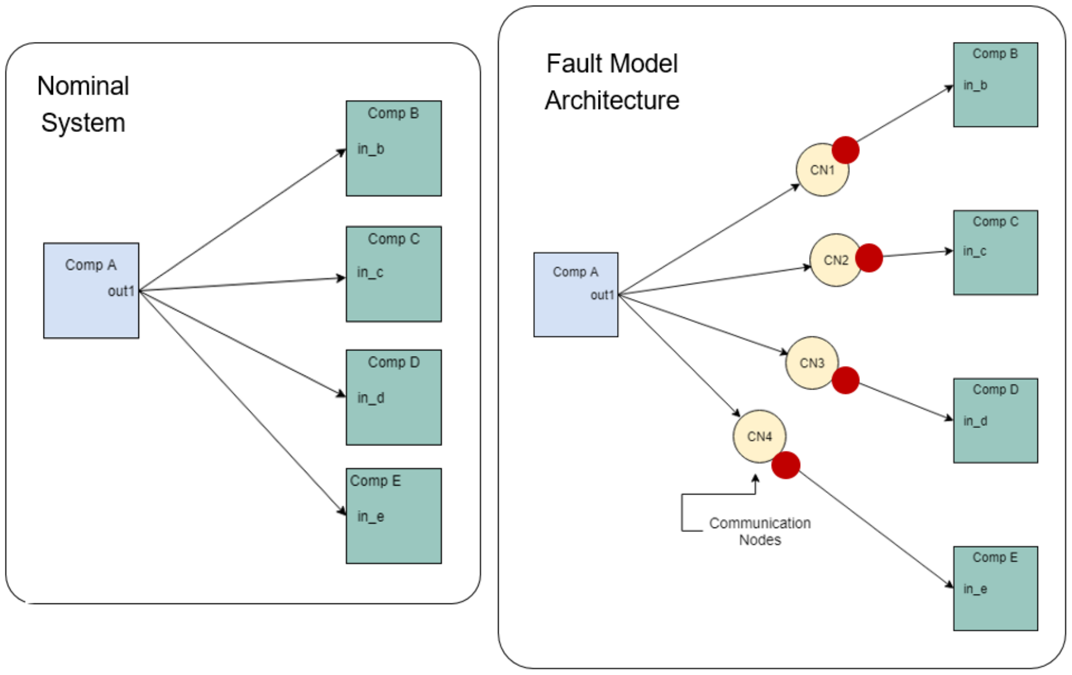


Figure 4.4: Communication Nodes in Asymmetric Fault Implementation

```
fault asymmetric_fault_Comp_A "Component A output asymmetric" : faults.fail_to {
        inputs: val_in <- Output, alt_val <- prev(Output, 0);
        outputs: Output <- val_out;
        probability: 5.0E-5;
        duration: permanent;
        propagate_type: asymmetric;
}
```

Figure 4.5: Asymmetric Fault Definition in the Safety Annex

An asymmetric fault is defined for Component A as in Figure 4.5. This fault defines an asymmetric failure on Component A that when active, is stuck at a previous value (*prev(Output, 0)*). This can be interpreted as the following: some connected components may only see the previous value of Comp A output and others may see the correct (current) value when the fault is active. This fault definition is injected into the communication nodes and which of the connected components see an incorrect value is completely nondeterministic. Any number of the communication node faults (0. . . all) may be active upon activation of the main asymmetric fault.

Process ID Example The illustration of asymmetric fault implementation can be seen through a simple example where 4 nodes report to each other their own process ID (PID). Each node has a 1-3 connection and thus each node is a candidate for an asymmetric fault. Given this architecture, a top level contract of the system is simply that all nodes report and see the correct PID of all other nodes. Naturally in the absence of faults, this holds. But when one asymmetric fault is introduced on any of the nodes, this contract cannot be verified. What is desired is a protocol in which all nodes agree on a value (correct or arbitrary) for all PIDs.

The Agreement Protocol Implementation in AGREE In order to mitigate this problem, special attention must be given to the behavioral model. Using the strategies outlined in previous research [26, 39], the agreement protocol is specified in AGREE to create a model resilient to one active Byzantine fault.

The objective of the agreement protocol is for all correct (non-failed) nodes to eventually reach agreement on the PID values of the other nodes. There are n nodes, possibly f failed nodes. The protocol requires $n > 3f$ nodes to handle a single fault. The point is to achieve distributed agreement and coordinated decisions. The properties that must be verified in order to prove the protocol works as desired are as follows:

- All correct (non-failed) nodes eventually reach a decision regarding the value they have been given. In this solution, nodes will agree in $f + 1$ time steps or rounds of communication.
- If the source node is correct, all other correct nodes agree on the value that was originally sent by the source.
- If the source node is failed, all other nodes must agree on some predetermined default value.

The updated architecture of the PID example is shown in Figure 4.6.

Each node reports its own PID to all other nodes in the first round of communication. In the second round, each node informs the others what they saw in terms of everyone's PIDs. The

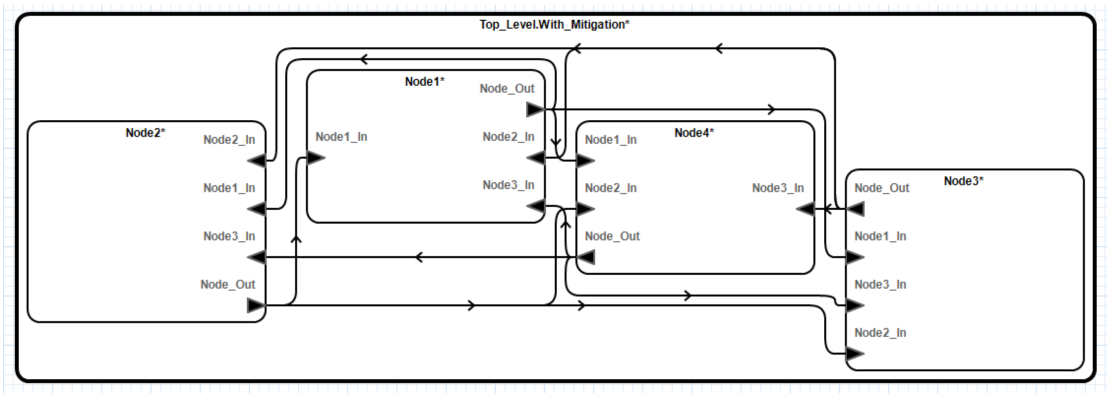


Figure 4.6: Updated PID Example Architecture

outputs from a node are described in Figure 4.7. These outputs are modeled as a nested data

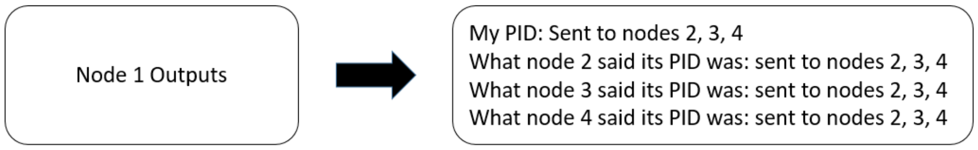

Node 1 Outputs
My PID: Sent to nodes 2, 3, 4
What node 2 said its PID was: sent to nodes 2, 3, 4
What node 3 said its PID was: sent to nodes 2, 3, 4
What node 4 said its PID was: sent to nodes 2, 3, 4

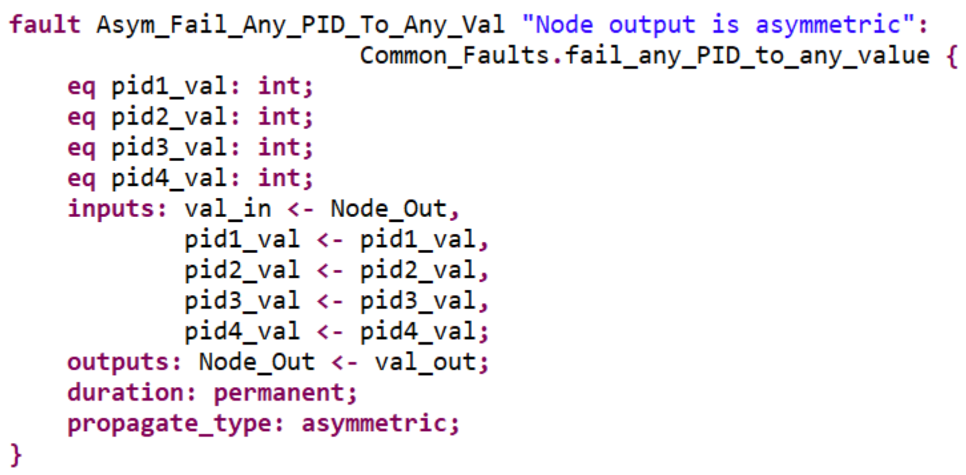
Figure 4.7: Description of the Outputs of Each Node in the PID Example

implementation in AADL and each field corresponds to a PID from a node. The AADL code fragment defining this data implementation is shown in Figure 4.8.

```
data implementation Node_Msg.Impl
    subcomponents
        Node1_PID_from_Node1: data Integer;
        Node2_PID_from_Node2: data Integer;
        Node3_PID_from_Node3: data Integer;
        Node4_PID_from_Node4: data Integer;
end Node_Msg.Impl;
```

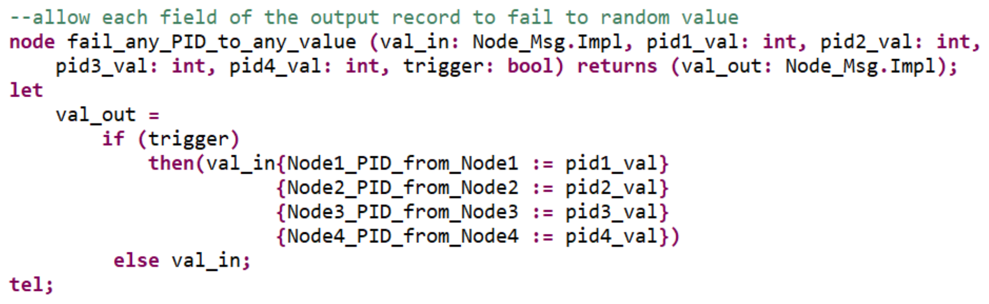
Figure 4.8: Data Implementation in AADL for Node Outputs

The fault definition for each node's output and can effect the data fields arbitrarily. This is a nondeterministic fault in two ways. It is nondeterministic how many receiving nodes see incorrect values and it is nondeterministic how many of the data fields are affected by this fault. This can be accomplished through the fault definition shown in Figure 4.9 and the fault node definition in Figure 4.10.



```
fault Asym_Fail_Any_PID_To_Any_Val "Node output is asymmetric":
                        Common_Faults.fail_any_PID_to_any_value {
    eq pid1_val: int;
    eq pid2_val: int;
    eq pid3_val: int;
    eq pid4_val: int;
    inputs: val_in <- Node_Out,
            pid1_val <- pid1_val,
            pid2_val <- pid2_val,
            pid3_val <- pid3_val,
            pid4_val <- pid4_val;
    outputs: Node_Out <- val_out;
    duration: permanent;
    propagate_type: asymmetric;
}
```

Figure 4.9: Fault Definition on Node Outputs for PID Example



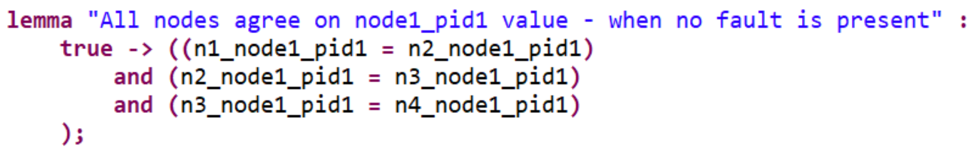
```
--allow each field of the output record to fail to random value
node fail_any_PID_to_any_value (val_in: Node_Msg.Impl, pid1_val: int, pid2_val: int,
    pid3_val: int, pid4_val: int, trigger: bool) returns (val_out: Node_Msg.Impl);
let
    val_out =
        if (trigger)
            then(val_in{Node1_PID_from_Node1 := pid1_val}
                       {Node2_PID_from_Node2 := pid2_val}
                       {Node3_PID_from_Node3 := pid3_val}
                       {Node4_PID_from_Node4 := pid4_val})
         else val_in;
tel;
```

Figure 4.10: Fault Node Definition for PID Example

Once the fault model is in place, the implementation in AGREE of the agreement protocol is developed. As stated previously, there are two cases that must be considered in the contracts of this system.

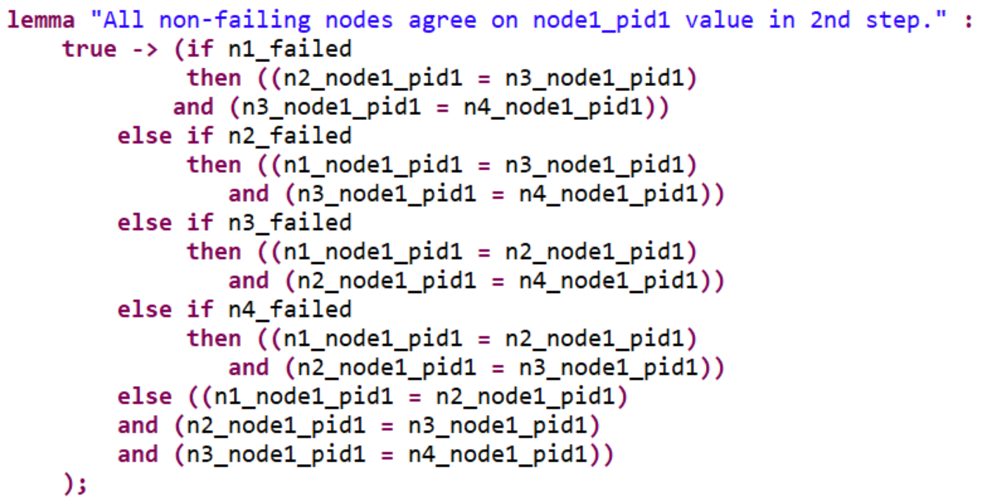
- In the case of no active faults, all nodes must agree on the correct PID of all other nodes.
- In the case of an active fault on a node, all non-failed nodes must agree on a PID for all other nodes.

These requirements are encoded in AGREE through the use of the following contracts. Figure 4.11 and Figure 4.12 show example contracts regarding Node 1 PID. There are similar contracts for each node's PID.



```
lemma "All nodes agree on node1_pid1 value - when no fault is present" :
    true -> ((n1_node1_pid1 = n2_node1_pid1)
        and (n2_node1_pid1 = n3_node1_pid1)
        and (n3_node1_pid1 = n4_node1_pid1)
    );
```

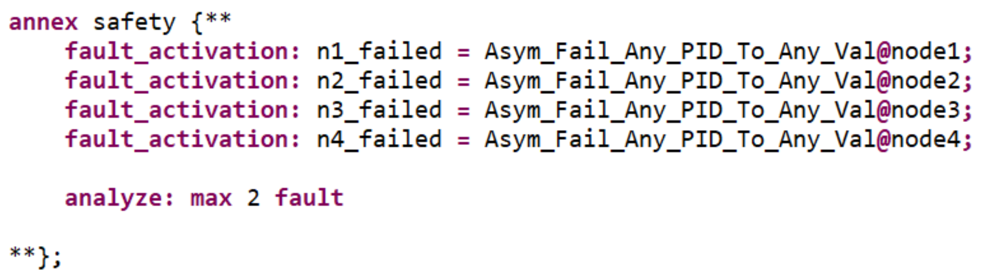
Figure 4.11: Agreement Protocol Contract in AGREE for No Active Faults



```
lemma "All non-failing nodes agree on node1_pid1 value in 2nd step." :
    true -> (if n1_failed
                then ((n2_node1_pid1 = n3_node1_pid1)
              and (n3_node1_pid1 = n4_node1_pid1))
        else if n2_failed
                then ((n1_node1_pid1 = n3_node1_pid1)
                   and (n3_node1_pid1 = n4_node1_pid1))
        else if n3_failed
                then ((n1_node1_pid1 = n2_node1_pid1)
                   and (n2_node1_pid1 = n4_node1_pid1))
        else if n4_failed
                then ((n1_node1_pid1 = n2_node1_pid1)
                   and (n2_node1_pid1 = n3_node1_pid1))
        else ((n1_node1_pid1 = n2_node1_pid1)
        and (n2_node1_pid1 = n3_node1_pid1)
        and (n3_node1_pid1 = n4_node1_pid1))
    );
```

Figure 4.12: Agreement Protocol Contract in AGREE Regarding Non-failed Nodes

Referencing Fault Activation Status To fully implement the agreement protocol, it must be possible to describe whether or not a subcomponent is failed by specifying if any faults defined for the subcomponents is activated. In the Safety Annex, this is made possible through the use of a *fault activation* statement. Users can declare boolean *eq* variables in the AGREE annex of the AADL system where the AGREE verification applies to that system's implementation. Users can then assign the activation status of specific faults to those *eq* variables in Safety Annex of the AADL system implementation (the same place where the fault analysis statement resides). This assignment links each specified AGREE boolean variable with the activation status of the specified fault activation literal. The AGREE boolean variable is true when and only when the fault is active. An example of this for the PID example is shown in Figure 4.13. Each of the *eq* variables declared in AGREE (i.e., *n1_failed, n2_failed, n3_failed, n4_failed*) is linked to the fault activation status of the *Asym_Fail_Any_PID_To_Any_Value* fault defined in a node subcomponent instance of the AADL system implementation (i.e., *node1*, *node2*, *node3*, *node4*).



```
annex safety {**
    fault_activation: n1_failed = Asym_Fail_Any_PID_To_Any_Val@node1;
    fault_activation: n2_failed = Asym_Fail_Any_PID_To_Any_Val@node2;
    fault_activation: n3_failed = Asym_Fail_Any_PID_To_Any_Val@node3;
    fault_activation: n4_failed = Asym_Fail_Any_PID_To_Any_Val@node4;

    analyze: max 2 fault

**};
```

Figure 4.13: Fault Activation Statement in PID Example

PID Example Analysis Results The nominal model verification shows that all properties are valid. Upon running verification of the fault model (*Verify in the Presence of Faults*) with one active fault, the first four properties stating that all nodes agree on the correct value (Figure 4.11) fail. This is expected since this property is specific to the case when no faults are present in the model. The remaining 4 top level properties (Figure 4.12) state that all non-failed nodes reach agreement in two rounds of communication. These are verified valid when any one asymmetric fault is present. This shows that the agreement protocol was successful in eliminating a single point of asymmetric failure from the model. Furthermore, when changing the number of allowed faults to two, these properties do not hold. This is expected given the theoretical result that $3f + 1$ nodes are required in order to be resilient to f faults and that $f + 1$ rounds of communication are needed for successful protocol implementation. A summary of the results follows.

- Nominal model: All top level guarantees are verified. All nodes output the correct value and all agree.

- Fault model with one active fault: The first four guarantees fail (when no fault is present, all nodes agree: shown in Figure 4.11). This is expected if faults are present. The last four guarantees (all non-failed nodes agree) are verified as true with one active fault.

- Fault model with two active faults: All 8 guarantees fail. This is expected since in order to be resilient up to two active faults ($f = 2$), we would need $3f + 1 = 7$ nodes and $f + 1 = 3$ rounds of communication.

This model is in Github and is called PIDByzantineAgreement [94].

Chapter 5

Case Studies

??

5.1 Wheel Brake System

?? To demonstrate the fault modeling capabilities of the Safety Annex we will use the Wheel Brake System (WBS) described in AIR6110 [3]. This system is a well-known example that has been used as a case study for safety analysis, formal verification, and contract based design [14, 19, 20, 58]. The preliminary work for the safety annex was based on a simple model of the WBS [96]. To demonstrate a more complex fault modeling process, we constructed a functionally and structurally equivalent AADL version of the more complex WBS NuSMV/xSAP models [20].

The WBS is composed of two main parts: the Line Replaceable Unit control system and the electro-mechanical physical system. The control system electronically controls the physical system and contains a redundant channel of the Braking System Control Unit (BSCU) in case a detectable fault occurs in the active channel. It also commands antiskid braking. The physical system consists of the hydraulic circuits running from hydraulic pumps to wheel brakes as well as valves that control the hydraulic fluid flow. This system provides braking force to each of the eight wheels of the aircraft. The wheels are all mechanically braked in pairs (one pair per landing gear). For simplicity, Figure 5.1 displays only two of the eight wheels.

There are three operating modes in the WBS model:

- In *normal* mode, the system is composed of a *green* hydraulic pump and one meter valve per each of the eight wheels. Each of the meter valves are controlled through electronic

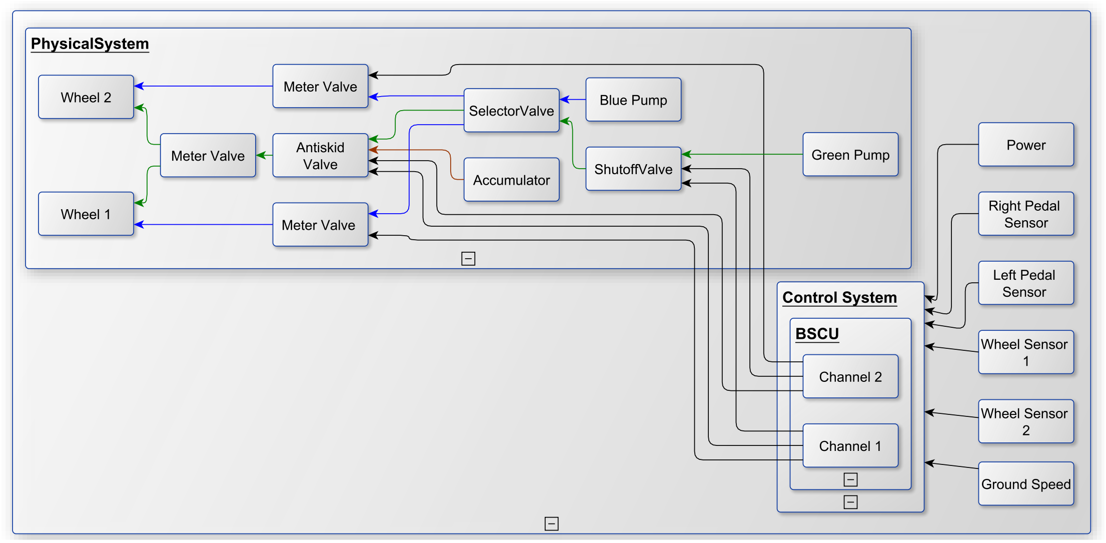


Figure 5.1: Wheel Brake System

commands coming from the active channel of the BSCU. These signals provide braking and antiskid commands for each wheel. The braking command is determined through a sensor on the pedal and the antiskid command is determined by the *Wheel Sensors*.

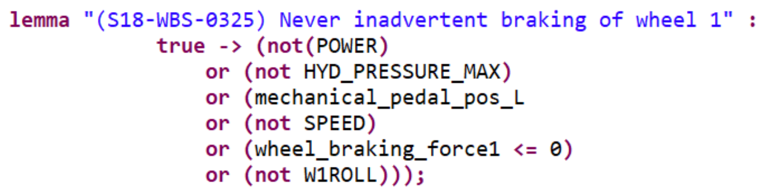
- In *alternate* mode, the system is composed of a *blue* hydraulic pump, four meter valves, and four antiskid shutoff valves, one for each landing gear. The meter valves are mechanically commanded through the pilot pedal corresponding to each landing gear. If the selector detects lack of pressure in the green circuit, it switches to the blue circuit.

- In *emergency* mode, the system mode is entered if the *blue* hydraulic pump fails. The accumulator pump has a reserve of pressurized hydraulic fluid and will supply this to the blue circuit in emergency mode.

The WBS architecture model in AADL contains 30 different kinds of components, 169 component instances, and a model depth of 5 hierarchical levels.

The behavioral model is encoded using the AGREE annex and the behavior is based on descriptions found in AIR6110. The top level system properties are given by the requirements and safety objectives in AIR6110. All of the subcomponent contracts support these system safety objectives through the use of assumptions on component input and guarantees on the output. The WBS behavioral model in AGREE annex includes one top-level assumption and 11 top-level system properties, with 113 guarantees allocated to subsystems.

An example system safety property is to ensure that there is no inadvertent braking of any of the wheels. This is based on a failure condition described in AIR6110 is *Inadvertent wheel braking on one wheel during takeoff shall be less than 1E-9 per takeoff*. Inadvertent braking means that braking force is applied at the wheel but the pilot has not pressed the brake pedal. In addition, the inadvertent braking requires that power and hydraulic pressure are both present, the plane is not stopped, and the wheel is rolling (not skidding). The property is stated in AGREE such that inadvertent braking does *not* occur, as shown in Figure 5.2.



```
lemma "(S18-WBS-0325) Never inadvertent braking of wheel 1" :
            true -> (not(POWER)
                or (not HYD_PRESSURE_MAX)
                or (mechanical_pedal_pos_L
                or (not SPEED)
                or (wheel_braking_force1 <= 0)
                or (not W1ROLL)));
```

Figure 5.2: AGREE Contract for Top Level Property: Inadvertent Braking

Chapter 6

Related Work

A model-based approach for safety analysis was proposed by Joshi et. al in [58–60]. 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) [45] and HiP-HOPS for EAST-ADL [28] are *FLM*-based *ESM* approaches. As previously discussed, given many possible faults, these propagation relationships require substantial user effort and become more complex. In addition, it becomes the analyst's responsibility to determine whether faults can propagate; missing propagations lead to unsound analyses. In our Safety Annex, propagations occur through system behaviors (defined by the nominal contracts) with no additional user effort.

Closely related to our work is the model-based safety assessment toolset called COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [15]. 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 [17, 21]. 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 [32, 72], MRMC (Markov Reward Model Checker) [62, 70], and RAT (Requirements Analysis Tool) [78]. The safety analysis tool xSAP [12] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [13]. 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 [**?**] 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".

The Safety Analysis and Modeling Language (SAML) [53] 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 [32], PRISM (Probabilistic Symbolic Model Checker) [63], or the MRMC probabilistic model checker [62].

AltaRica [11, 75] 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 [10]. For this dialect the Safety assessment, fault tree generation, and functional verification can be performed with the aid of NuSMV model checking [18]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [9]. AltaRica 3.0 has support for simulation and Markov model generation through the OpenAltaRica (www.openaltarica.fr) tool suite.

Formal verification tools based on model checking have been used to automate the generation of safety artifacts [12, 18, 22]. This approach has limitations in terms of scalability and readability of the fault trees generated. Work has been done towards mitigating these limitations by the scalable generation of readable fault trees [19].

Chapter 7

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 [96].

Sept 2017 - Feb 2018: In depth research into the needs of safety analysts and complex fault modeling capabilities [95].

March 2018 - March 2019: Further implementation of various types of fault behavior (dependent faults, asymmetric faults, etc). [93].

March 2019 - Sept 2019: Research into the theory of the *MinCutSet* transformation and preliminary implementation thereof [92].

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] Defence Standard. Standard 00-55. *Requirements for Safety Related Software in Defence Equipment, Ministry of Defence*, 1999.

[2] S. Aerospace. Sae as5506b: Architecture analysis & design language (aadl) standard document. *SAE International*, 2012.

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

[4] R. Banov, Z. Šimić, and D. Grgić. A new heuristics for the event ordering in binary decision diagram applied in fault tree analysis. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, page 1748006X19879305, 2019.

[5] A. Belov, H. Chen, A. Mishchenko, and J. Marques-Silva. Core minimization in sat-based abstraction. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1411–1416. EDA Consortium, 2013.

[6] A. Belov, I. Lynce, and J. Marques-Silva. Towards efficient mus extraction. *AI Communications*, 25(2):97–116, 2012.

[7] A. Belov and J. Marques-Silva. Muser2: An efficient mus extractor. *Journal on Satisfiability, Boolean Modeling and Computation*, 8(3-4):123–128, 2012.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. Safety, dependability and performance analysis of extended aadl models. *Comput. J.*, 54(5), May 2011.

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

[18] 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.

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

[20] 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.

[21] 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.

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

[23] 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.

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

[25] M. Bozzano, A. Villafiorita, O. Åkerlund, P. Bieber, C. Bougnol, E. Böde, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi, et al. Esacs: an integrated methodology for design and safety analysis of complex systems. In *Proc. ESREL*, volume 2003. Balkema Publisher, 2003.

[26] G. Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

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

[28] 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.

[29] A. Chiappini, A. Cimatti, C. Porzia, G. Rotondo, R. Sebastiani, P. Traverso, and A. Villafiorita. Formal specification and development of a safety-critical train management system. In *International Conference on Computer Safety, Reliability, and Security*, pages 410–419. Springer, 1999.

[30] J. Christ, J. Hoenicke, and A. Nutz. Smtinterpol: An interpolating smt solver. In *Proceedings of the 19th International Conference on Model Checking Software*, SPIN'12, pages 248–254, Berlin, Heidelberg, 2012. Springer-Verlag.

[31] A. Cimatti. Industrial applications of model checking. In *Summer School on Modeling and Verification of Parallel Processes*, pages 153–168. Springer, 2000.

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

[33] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[34] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking*. MIT press, 2018.

[35] O. Coudert and J. C. Madre. Fault tree analysis: 10/sup 20/prime implicants and beyond. In *Annual Reliability and Maintainability Symposium 1993 Proceedings*, pages 240–245. IEEE, 1993.

[36] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] T. C. development team. *The Coq proof assistant reference manual*. LogiCal Project, 2019. Version 8.10.2.

[38] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software verification using k-induction. In *International Static Analysis Symposium*, pages 351–368. Springer, 2011.

[39] K. Driscoll, H. Sivencrona, and P. Zumsteg. Byzantine Fault Tolerance, from Theory to Reality. In *SAFECOMP*, LNCS, 2003.

[40] V. D'silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[41] B. Dutertre. Yicesä2.2. In *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*, pages 737–744, Berlin, Heidelberg, 2014. Springer-Verlag.

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

[43] Federal Aviation Administration. System design and analysis document information. *FAA Advisory Circular 25*, 2002.

[44] P. Feiler and J. Delange. Automated fault tree analysis from aadl models. *ACM SIGAda Ada Letters*, 36(2):39–46, 2017.

[45] 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.

[46] M. Fitting. *First-order logic and automated theorem proving*. Springer Science & Business Media, 2012.

[47] G. Fraser, F. Wotawa, and P. E. Ammann. Testing with model checkers: a survey. *Software Testing, Verification and Reliability*, 19(3):215–261, 2009.

[48] J. B. FUSSELL. A new methodology for obtaining cut sets for fault trees. *Trans. Am. Nucl. Soc.*, 1972.

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

[50] D. Ge, M. Lin, Y. Yang, R. Zhang, and Q. Chou. Quantitative analysis of dynamic fault trees using improved sequential binary decision diagrams. *Reliability Engineering & System Safety*, 142:289–299, 2015.

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

[52] 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.

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

[54] M. G. Hinchey and J. P. Bowen. *Industrial-strength formal methods in practice*. Springer Science & Business Media, 2012.

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

[56] P. Hönig and R. Lunde. A new modeling approach for automated safety analysis based on information flows. In *Proceedings of the 25th International Workshop on Principles of Diagnosis (DX14), Graz, Austria*, pages 8–11, 2014.

[57] W. Jiang, S. Zhou, L. Ye, D. Zhao, J. Tian, W. E. Wong, and J. Xiang. An algebraic binary decision diagram for analysis of dynamic fault tree. In *2018 5th International Conference on Dependable Systems and Their Applications (DSA)*, pages 44–51. IEEE, 2018.

[58] 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.

[59] 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.

[60] 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.

[61] T. Kahsai and C. Tinelli. Pkind: A parallel k-induction based model checker. *arXiv preprint arXiv:1111.0372*, 2011.

[62] 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.

[63] 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.

[64] D. C. Latham. Department of defense trusted computer system evaluation criteria. 1986.

[65] M. H. Liffiton, Z. Andraus, and K. Sakallah. From max-sat to min-unsat: Insights and applications. *Ann Arbor*, 1001:48109–2122, 2005.

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

[67] P. Liggesmeyer and M. Rothfelder. Improving system reliability with automatic fault tree generation. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224)*, pages 90–99. IEEE, 1998.

[68] 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.

[69] Y. Liu, G. Shen, F. Wang, J. Si, and Z. Wang. Research on aadl model for qualitative safety analysis of embedded systems. *International Journal of Multimedia and Ubiquitous Engineering*, 11(6):153–170, 2016.

[70] MRMC: Markov Rewards Model Checker. http://wwwhome.cs.utwente.nl/ zapreevis/m-rmc/.

[71] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

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

[73] G. Point and A. Rauzy. AltaRica: Constraint automata as a description language. *Journal Européen des Systèmes Automatisés*, 33(8–9):1033–1052, 1999.

[74] M. R. Prasad, A. Biere, and A. Gupta. A survey of recent advances in sat-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[75] 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.

[76] T. Prosvirnova and A. Rauzy. Altarica 3.0 project: Compiling guarded transition systems into fault trees. In *Proceedings of the European Safety and Reliability conference, ESREL 2013*, Amsterdam, The Netherlands, September-October 2013. CRC Press.

[77] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

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

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

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

[81] A. Rauzy. Computation of prime implicants of a fault tree within aralia. *Reliability Engineering and System Safety*, 1996.

[82] A. Rauzy, J. Gauthier, and X. Leduc. Assessment of large automatically generated fault trees by means of binary decision diagrams. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, 221(2):95–105, 2007.

[83] K. A. Reay and J. D. Andrews. A fault tree analysis strategy using binary decision diagrams. *Reliability engineering & system safety*, 78(1):45–56, 2002.

[84] J. D. Reese and N. G. Leveson. Software deviation analysis: A "safeware" technique. In *AIChe 31st Annual Loss Prevention Symposium*. Citeseer, 1997.

[85] 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.

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

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

[88] S. N. Semanderes. " elraft" a computer program for the efficient logic reduction analysis of fault trees. *IEEE Transactions on Nuclear Science*, 18(1):481–487, 1971.

[89] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *International conference on formal methods in computer-aided design*, pages 127–144. Springer, 2000.

[90] J.-P. Signoret, S. Lajeunesse, G. Point, P. Thomas, A. Griffault, and A. Rauzy. The Altarica Language. In Lydersen, Hansen, and Sandtorv, editors, *Proceedings of European Safety and Reliability Association Conference, ESREL'98*, pages 1327–1334, Trondheim, Norway, June 1998. Balkerna, Rotterdam.

[91] R. M. Sinnamon and J. Andrews. New approaches to evaluating fault trees. *Reliability Engineering & System Safety*, 58(2):89–96, 1997.

[92] 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.

[93] 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.

[94] D. Stewart, J. Liu, M. Whalen, D. Cofer, and M. Peterson. Safety annex for aadl repository. `https://github.com/loonwerks/AMASE`, 2018.

[95] 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.

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

[97] U.S. Department of Defense. Procedures for performing a failure mode, effects and criticality analysis (MIL-P-1629A), 1949.

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