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

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

In our increasingly computerized world, the concept of system safety has become of great importance to many different fields. A *complex safety critical system* is one whose safety cannot be shown only through testing, whose logic is difficult to comprehend without the aid of analytical tools, and that may contribute – directly or indirectly – to loss of life, damage of the environment, or large economic losses [1]. Critical systems can be found in the aviation, automotive, nuclear, or medical industries and the process of designing such systems, from inception to deployment in society, presents numerous problems with which researchers have been contending.

System safety has been an important factor in the design of systems for many years, but the birth of system safety as we know it today is associated with problems that the US Air Force experienced with accidents after World War II. Over 7,700 aircraft were lost between the years of 1952 and 1966 and over 8,000 people were killed [64]. At the time, many of the accidents were blamed on pilots, but many flight engineers did not believe the causes were so simple. They posited that safety must be designed and built into the aircraft [78]. Between the growth of nuclear capabilities, the defense industry complex, and the overall increase of computer capabilities, the need to abandon a "fly-fix-fly" approach to safety was imminent [64, 78, 86]. The goal became to avoid accidents, instead of fixing a problem after an accident occurs.

Today, 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. The process meant to guide the development and certification of safety critical systems has been standardized by competent authorities [1, 109, 110].

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 overall system behavior, assess the effect of failures on the system's safety objectives, and construct the accompanying safety analysis artifacts so that safe operation can be ensured and demonstrated [109, 110]. System information and safety artifacts can also reveal missing requirements or be used to strengthen the existing ones, and they give crucial information about how the system responds to faulty components or errors in functionality that cross component boundaries. check this An important goal of the safety assessment process is to show what specific kinds of failures may occur during normal use of the system; both qualitative and quantitative analyses can provide information on how the system is safe (or unsafe) for use.

The development life cycle of critical systems can be roughly seen as two main thrusts that occur in tandem: one side focuses on the system development itself; the hardware and software design, the requirements of the system, and the logical behavior of the components and their interactions. The other side is safety assessment of the system. Safety analysts are concerned with the failure of a system; systems can be unsafe (fail) with or without component or software failures. Safety analysts use information generated during the system design and development process and analyze the system from the perspective of failure; in other words, they focus on what can make the system unsafe. This is used to strengthen the system design and provide feedback into the development process.

Due to the complex nature of this arrangement, these sides are in reality not always done in strict parallel and are rarely synchronized perfectly. Furthermore, the artifacts given to safety analysts from system engineers are not always formal in nature, they may come from various sources, and they often do not clearly define the entire system and its behavior. To address this concern, *model-based system development* (MBSA) caught the attention of researchers in the safety critical system domains [16, 62, 69, 73, 82]. In model-based development, the development efforts are centered on a model of the intended system. Various techniques, such as formal verification, testing, test case 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. The safety assessment process should accurately and efficiently reflect the functionality of the system in the presence of faults and thus be used in the certification process.

Many of the techniques proposed for MBSA require the development of *fault models* specific for safety analysis; that is, the techniques do not rely on the *extension* of existing system models, but rather require purpose-built fault models that are separate entities [14, 20, 62, 68]. Thus there is a system model used by the system engineers and a fault model used by safety analysts. It requires extra manual labor to create a separate fault model that accurately describes the system model itself. 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; just as it is difficult to show that the system model conforms to the actual implemented system. Another problem inherent in this approach is that any changes made in system development are not automatically reflected in the safety analysis process. This brings us right back to a non-model based approach.

Part of the safety assessment process determines how faults can manifest themselves in a particular component, but also how a manifested fault (or *error*) can propagate through a system. Error propagation can be handled a variety of ways; most commonly this is done through the use of signal flow diagrams, a deep understanding of the system components, and the intuition of a good analyst. Various research has attempted to address this gap by providing tools that operate over a model and provide some form of propagation analysis. Often this propagation is done explicitly, but as the size and complexity of industrial sized systems grow, this form of propagation becomes quite unwieldy. To address this problem, *behavioral* propagation has been introduced [14, 118]. Behavioral propagation automates the process of "moving" the error through the system and requires no explicit statements of what effect the error will have on components. In reality, both approaches are beneficial to an analyst. At times, there are effects that are known and easily captured explicitly. Other times, even within the same system, complex interactions make explicit propagation difficult to manage. To provide the most flexibility for an analyst, both approaches are beneficial.

To correctly use MBSA, *verification* of the model and its requirements is necessary. Verification is the process of mathematically proving or disproving the correctness of a system with respect to certain properties or requirements. As a model and the number of system requirements grow, a scalable approach is of utmost concern cite.

Commonly used artifacts in the safety assessment process are *minimal cut sets*, or the minimal sets of faults that violate a system safety property. The automatic generation of these artifacts have been studied in depth, but have lacked in terms of scalability. Some research groups have introduced automating aspects of the safety assessment process and have developed tools to support this [16, 21, 71]; nevertheless, there are gaps in current capabilities we address in this dissertation.

1.1 Objective and Summary of Contributions

The **long range goal** of this research 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 **objectives of this dissertation**, which are logical steps towards the goal, are to define a modeling notation such that the information required for the safety assessment process can be easily captured in the system model. Once this notation was in place, we defined analysis procedures to verify that the system model meets its requirements in the face of failures. Further exploration of the model, component interactions, and problematic fault combinations are incorporated into these analyses in order to fully understand the safety of the system. Domain specific case studies then demonstrate the effectiveness of this approach.

The objectives of this dissertation were accomplished by providing the following contributions:

Defined a modeling notation to capture safety information in a shared model. Before a fault modeling notation was defined, we chose an appropriate modeling language. The Architecture Analysis and Design Language (AADL) is an SAE International standard language that provides a unifying framework for describing the system architecture for performance-critical, embedded, real-time systems [4, 50]. From its conception, AADL has been designed for the design and construction of avionics systems. Rather than being merely descriptive, AADL models can be made specific enough to support system-level code generation; thus, results from analyses conducted, including safety analysis, correspond to the system that will be built from the model. This specificity supports a close relationship between the system development and safety assessment processes. This modeling language was chosen for these reasons.

We extended the AADL with a safety annex extension and kept a few specific fault modeling needs in mind [120]. The extension supports behavioral and explicit fault propagation, flexible fault modeling which allows for modeling various types of realistic component failures, and a back-end model checker o perform the analysis. Within the AADL model, a user can add the safety annex which contains fault definitions for components. The flexibility of the fault definitions allows the definitions of complex or simple fault behavior. This allows analysts to capture realistic faulty components and scenarios in the model. When a fault is activated, it modifies the output of the component. This faulty behavior may lead to a violation of the contracts of other components in the system, including assumptions of downstream components. The model checker analyzes the impact of a fault when the safety analysis is executed on the extended model.

Defined analysis procedures to verify behavior of the model in the presence of faults. Given a safety property or requirement, it is useful to see if that property can be verified when faults are present (or active) in the system model. 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 (*max n fault hypothesis*) or that the only faults to be considered are those whose probability of simultaneous occurrence is above some probability threshold (*probabilistic hypothesis*). In the former case, we assert that the number of simultaneous faults 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 the fault activation variables. If any combination of faults falls within these parameters during analysis, the user can view the state of the system when a violation occurs. This analysis provides valuable system information about the relationship between the requirement of interest and the defined faults in the model.

Provided in depth analysis capabilities that explore system models and compositionally compute minimal cut sets. The minimal sets of faults that violate a safety property, minimal cut sets, are commonly used in the assessment and certification of critical systems. Since the introduction of cut sets in the field of safety analysis, much research has been performed to address their generation [27, 47, 99, 100, 106]. One of the ongoing problems with minimal cut set generation is the inability to scale to industrial-sized systems. As the system gets larger, more minimal cut sets are possible with ever increasing cardinality. In recent years, researchers have leveraged model checking to address this problem. [13, 22, 23, 25, 106, 111]. We have pushed forward on this front and found a way to generate these sets in a *compositional* fashion. Compositional verification performs the proof in a per-architectural-layer approach and to our knowledge, compositional minimal cut set generation has not been previously performed. This research formally proves the relationship between verification results of compositional analysis and minimal cut sets, and implements the algorithms for minimal cut set generation in the Safety Annex.

Explored how the syntax of requirements can change analysis results. Splitting a complex requirement into its constituent pieces can change certain analysis results. Because the compositional generation of minimal cut sets relies on the contracts written for each component, it is natural to question how the written requirements may affect analysis results. We explored this idea by automatically detecting certain patterns in the requirements and rewriting them into semantically equivalent but syntactically different forms. Then we compared analysis results between the original contracts and the rewritten contracts and discussed the findings. We found that as the requirements became more *granular*, the minimal cut sets computed could decrease in size; the faults in them were found to be tied more directly to the system level requirements. The specificity of these requirements is referred to as *granularity* and this idea ties into the broader discussion of the ideas underlying requirement engineering, behavioral modeling, and system development.

Demonstrated the objectives of this proposal by use of case studies. An industrial sized case study from the safety critical aerospace domain illustrates the process of using the safety annex for AADL and demonstrates the capabilities of the implemented analyses described in this research. Numerous subsystem examples are given to illustrate specific capabilities and solutions. This serves to demonstrate how this process works in the domain of aerospace and how it can be applied to various critical system domains.

In summary, this dissertation provides a comprehensive solution to

1.2 Structure of this Document

This dissertation is organized into 7 chapters. The preliminaries in Chapter 2 discusses the background and justification for this research, critical system development and the state of the field, and ends with an overview of formal verification. Chapter provides a detailed look at the Safety Annex and its implementation. Chapter 4 describes the compositional generation of minimal cut sets. Chapter provides the initial research into how a particular form of contract definition can change the results of the analysis; it is followed by a chapter on case studies. Lastly, the conclusion summarizes the research approach in this dissertation.

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 [109, 110].

2.1.1 The V Model

The development of safety critical systems is theoretically guided by the V model process as defined in ARP4754 [110]. 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.

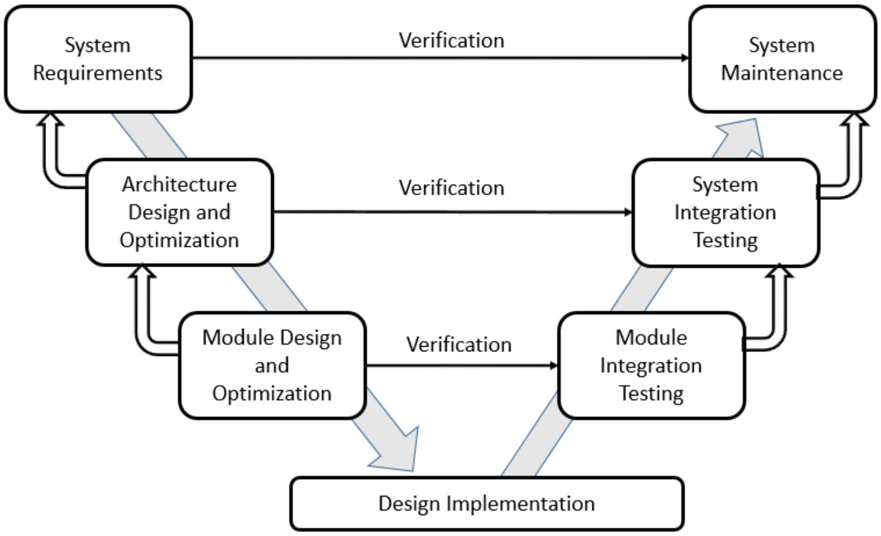

System Requirements
Verification
System Maintenance
Architecture Design and Optimization
Verification
System Integration Testing
Module Design and Optimization
Verification
Module Integration Testing
Design Implementation

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 [110], 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

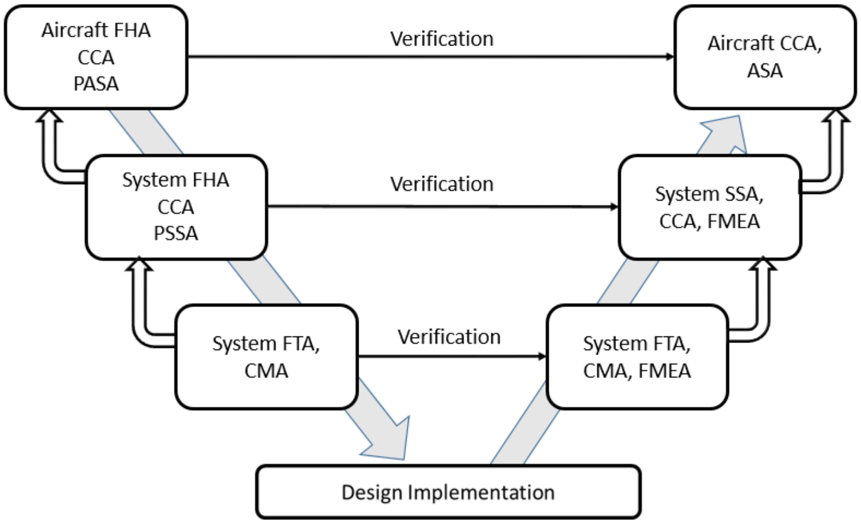


Figure 2.2: The V Model in Safety Assessment

includes identification of failure conditions, identifying the effects of those failures, classification of each failure condition, and assignment to safety objectives.

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.

2.1.3 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. A diagram of this process is shown in Figure 2.3.

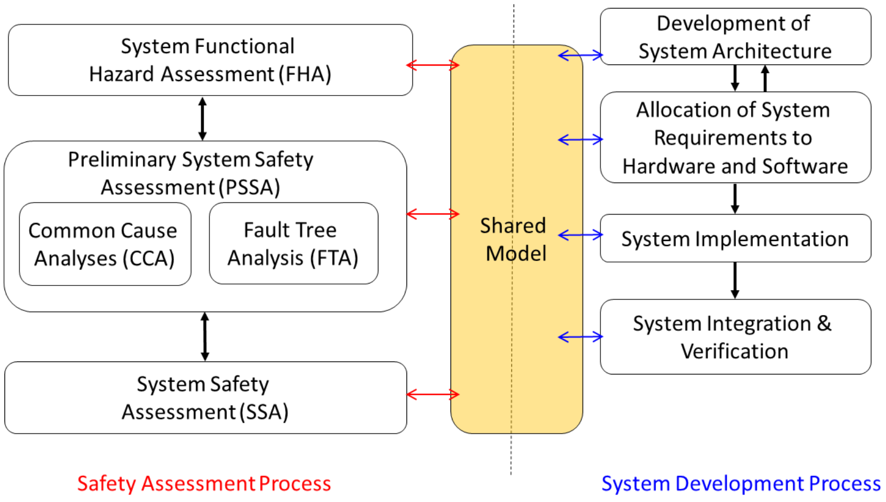


Figure 2.3: Use of the Shared System/Safety Model in the ARP4754A Safety Assessment Process

Industry practitioners have come to realize the benefits of using models in the safety assessment process, and a revision of the ARP4761 to include Model Based Safety Analysis (MBSA) is under way.

2.1.4 Proposed 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 2.4 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 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.

These steps can be viewed as a cyclical process that involves both the system development engineers and the safety engineers of the system. Figure 2.4 shows these steps within the context of the start and end of a project.

Add a bit more information here - pull from the MBSE project for IRAD. Include reasons why this approach is better, how it will help safety analysts, how it benefits the field as a whole.

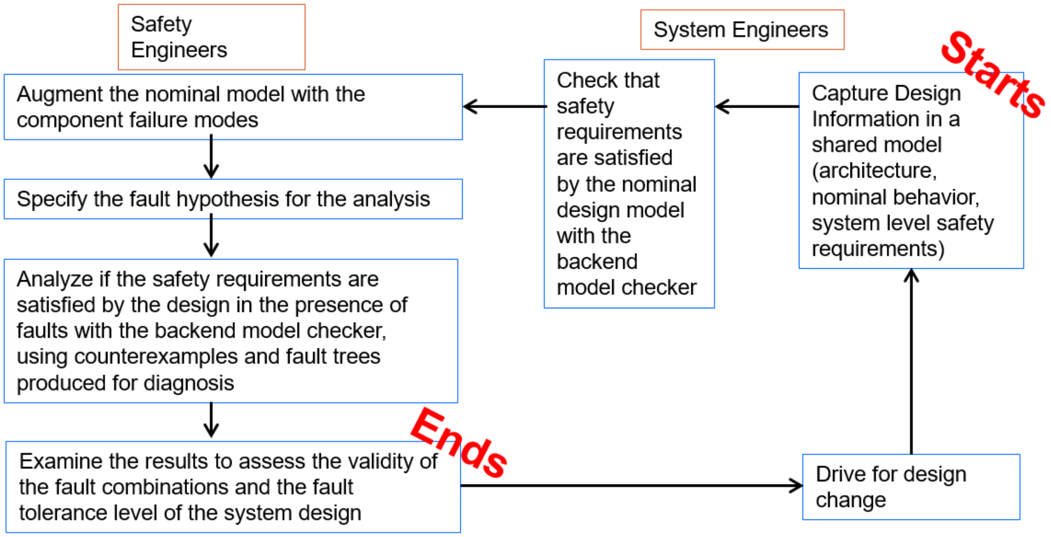

Safety Engineers
System Engineers
Starts
Augment the nominal model with the component failure modes
Check that safety requirements are satisfied by the nominal design model with the backend model checker
Capture Design Information in a shared model (architecture, nominal behavior, system level safety requirements)
Specify the fault hypothesis for the analysis
Analyze if the safety requirements are satisfied by the design in the presence of faults with the backend model checker, using counterexamples and fault trees produced for diagnosis
Ends
Examine the results to assess the validity of the fault combinations and the fault tolerance level of the system design
Drive for design change

Figure 2.4: Proposed Steps of the Safety Assessment Process

Then lead into the next sections with a statement about model checking, verification, etc. subsectionCritical System Development Artifacts

2.2 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 [27, 28, 39, 66, 81]. There arose multiple ways of viewing the system and fault models, various ways of automating the capture of safety pertinent 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 research.

2.2.1 Fault Tree Analysis

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 [107, 122]. 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 [54, 112]. 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) [100]. 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 [30, 101–103, 115]. BDDs are still commonly used to perform quantitative and qualitative fault tree analysis [6, 58, 70].

2.2.2 Model Checking in 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 are still contributing today (e.g., [31, 33, 104, 114].

One of the main methods was the abstraction of 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 [45].

In the early 2000's, model based safety assessment began to make an appearance in literature [27, 71–73]. 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*. Make a figure here - that will help explain these distinctions.

This literature overview is not a complete account of all safety analysis model checking tools available either in industry or research, but 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 [114]. The key idea was that this transition system (more specifically *constraint automata*) could be compiled into Boolean formulae and transformed into a BDD [92]. 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 [12]. For this dialect, the safety assessment, fault tree generation, and functional verification can be performed with the aid of NuSMV model checking [20].

The most recent language update (AltaRica 3.0) uses non-causal semantics [94, 95]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [11]. 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.

AltaRica 3.0 provides automated fault tree generation by translating the model into a reachability graph and then further compiling it into Boolean formula in order to compute minimal cut sets [96].

FSAP, xSAP, and COMPASS

The Formal Safety Analysis Platform (FSAP) was introduced in 2003 [26] 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 [34].

The system model, written in NuSMV, and the fault model, developed graphically in FSAP, are together translated into a finite state machine and eventually into a BDD; fault tree analysis is performed using BDD algorithms implemented in NuSMV.

By 2016, the researchers that developed FSAP (Foundation Bruno Kessler, FBK) released a similar tool called xSAP [14]. 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. More complex SAT-based algorithms were introduced to bypass the BDD method of minimal cut set generation, namely the "anytime approximation" algorithms [16, 85]. These algorithms make clever use of bounded model checking algorithms to explore counterexamples provided to the query ”the top level event never occurs." These explorations are done such that the cut sets generated are of increasing cardinality which allows for an approximation computation to be given even when the state space is too large to compute all minimal cut sets. These are implemented in xSAP [16].

COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [18] 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 [17,24]. 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 [34, 90], MRMC (Markov Reward Model Checker) [75, 87], and RAT (Requirements Analysis Tool) [98]. The safety analysis tool xSAP [14] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [15].

SmartIFlow

SmartIFlow [67, 69] 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" [69].

SAML

The Safety Analysis and Modeling Language (SAML) [62] 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 [34], PRISM (Probabilistic Symbolic Model Checker) [76], or the MRMC probabilistic model checker [75]. SAML itself does not provide the formal verification engines, but instead provides a platform to model the safety aspects of a system and then translate this into the input language for a formal verification engine [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 [83, 108]. 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) [51]. 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 [49].

2.3 Formal Methods of Verification and Validation

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 [48]. Utilizing the recent advancements in automated formal verification within the validation process has become essential to the certification of critical systems [2, 77, 93]. 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 [66]. 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 [52]. 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 [36]. 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 [35, 97]. By representing the system abstractly, the infinite state space is reduced to a finite model. This addresses the undecidability factor [45]. 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 [36,53]. 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 [45]. Nevertheless, the biggest limiting factor of model checking is scalability and much of the recent research in this area attempts to address this problem [36].

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 [45, 52]. Deductive methods of verification include automated theorem provers (e.g., Coq [42], Isabelle [89]) and satisfiability modulo theories (e.g., SMTInterpol [32], Z3 [41], Yices [46]).

2.3.2 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.

2.3.3 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 [80]. 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 [79] and most of them focus on finding a single such subset [7–9]. More recently, algorithms have been introduced that can find all such minimal unsatisfiable subsets [10, 60, 61].

2.3.4 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 [60]. 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`) [10, 61].

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.

2.3.5 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.

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, 109, 110].

A Fault Tree (FT) is a directed acyclic graph whose leaves model component failures and whose gates model failure propagation [107]. 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 [47]. 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 [107].

Figure 2.5 shows a simple example of a fault tree based on SAE ARP4761 [109]. In this example, the top level event corresponds to an aircraft losing all wheel braking. In order for this event to occur, all of the basic events must occur. This is seen through the use of the AND gate below the top level event. The gates in the fault tree describe how failures propagate through the system. Each gate has one output and one or more inputs. In Figure 2.5, 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

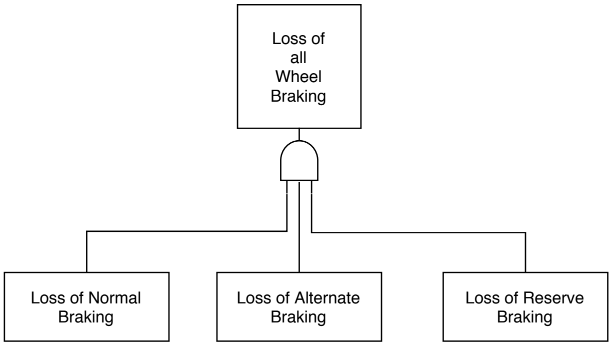


Figure 2.5: A simple fault tree

in Bell Labs in 1961 [47, 107].

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 [107].

Failure Mode and Effects Analysis

Is this section necessary? If I don't talk about FMEA tables again, cut this. 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 [19, 99]. 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 [121]. Performing FMEA is often the first step in the fault tree construction, for it shows possible component failures and hence basic events [107]. 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). 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 [27].

Ordered Binary Decision Diagrams

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

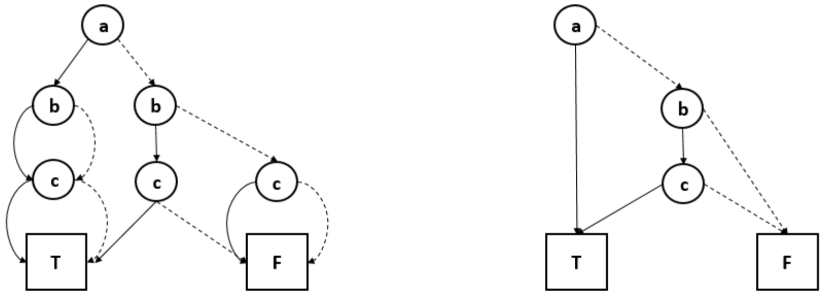


Figure 2.6: Binary Decision Diagrams of the Formula $a \vee (b \wedge c)$

shown in Figure 2.6, 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 2.6 is the reduced form of the BDD on the left.

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). 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 [55], an infinite state model checker. Verification of the program is based on *k-induction* (see Section 2.3.5) and property directed reachability using a back-end SMT solver, e.g., Z3 [41], SMTInterpol [32].

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 [27].

k-induction The k-induction method was introduced as a technique for SAT-based verification of finite and infinite state transition systems [113]. 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 [43] or performing them in parallel [74].

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”
- *Author unknown*

Linear Temporal Logic Temporal logic can be used to express properties of reactive systems [27]. System properties are usually classified into two main categories: *safety* properties and *liveness* properties. Safety properties express the idea that “nothing bad ever happens” where liveness properties state that “something good will eventually happen.”

An example of a safety property is: “it is never the case that the brake pedal is pressed and no hydraulic pressure is supplied at the wheel.” A liveness property, on the other hand, could state: “eventually the process will complete it’s execution.”

Traditionally, two types of temporal logic are used in model checking; Computational Tree Logic (CTL), which is based on a branching logic model, and Linear Temporal Logic (LTL), based on a linear representation of time. This research will focus on LTL.

An LTL formula is built from a set of atomic propositions, logical operators, and basic temporal operators. The formula is evaluated over a linear path or sequence of states, $s_0, s_1, ..., s_i, s_{i+1}, ...$. The following temporal operators are provided:

- Globally (**G**): G_p is true in a state s_i if and only if p is true in all states s_j with $j \geq i$.
- Finally (**F**): F_p is true in state s_i if and only if p is true in some state s_j with $j \geq i$.
- Next (**X**): X_p is true in state s_i if and only if p is true in the state s_{i+1}.
- Until (**U**): pUq is true in state s_i if and only if q is true in some state s_j with $j \geq i$ and p is true in all states s_k such that $i \leq k < j$.

Other temporal operators can be defined on the basis of the operators above [116]. Formal definitions and more information on LTL and CTL can be found in a number of research works [27, 36].

2.3.6 Compositional Model Checking

Compositional analysis of systems was introduced in order to address the scalability of model checking large software systems [38, 65, 91]. Normally, a SAT solver will flatten the hierarchical system model and use all model elements from all layers in order to find proof of a safety property. The analysis can alternatively be performed compositionally following the architecture hierarchy such that analysis at a higher level is based on the components at the next lower level and conducted layer by layer; the components of a system are organized hierarchically and each layer of the architecture is viewed a system. The idea is to partition the formal analysis of a system architecture into verification tasks that correspond into the decomposition of the architecture.

Assume-Guarantee Reasoning Environment

The Assume-Guarantee Reasoning Environment (AGREE) [37] provides a way to perform compositional verification on models that are defined using the Architecture Analysis and Design Language (AADL) [108].

A component contract in an assume-guarantee reasoning environment is an assume-guarantee pair. Intuitively, the meaning of a pair is: if the assumption is true, then the component will ensure that the guarantee is true. The formulation of AGREE uses LTL operators G (globally), H (historically), and Z (in the previous instant).

Formally, a component contract is an assume-guarantee pair (A, P) for propositions A, P. The meaning of a pair is that a component is required to meet it's guarantee only if its assumptions have been true up to the current instant [37]. Stated as an LTL formula, this is $G(H(A) \implies P)$.

Each architectural layer is viewed as a system with inputs, outputs, and components. A system S can be described as its own contract (A_S, P_S) and the contracts of its components C_S. Thus, $S = (A_S, P_S, C_S)$. For each layer, the proof consists of demonstrating that the system guarantee is provable given the guarantees of its direct subcomponents and the system assumptions, or more formally prove $G(H(A_S) \implies P_S)$ given $G(H(A_C) \implies P_C)$ for each component C in the system.

This proof is performed one layer at a time starting from the top level of the system. When compared to monolithic analysis (i.e., analysis of the flattened model composed of all components), the compositional approach allows the analysis to scale to much larger systems [38]. AGREE utilizes the JKind model checker [55], an infinite state k-induction model checker.

Verification of the program is performed using a back-end SMT solver, e.g., Z3 [41], SMTInterpol [32].

Chapter 3

Fault Modeling and the Safety Annex

Early on in Section 2.2.1, a model-based safety assessment process was proposed. This process was backed by formal methods and incorporates a shared model into the development and safety analysis processes. A high level description of this cyclical process is shown in Figure 3.1 for your convenience.

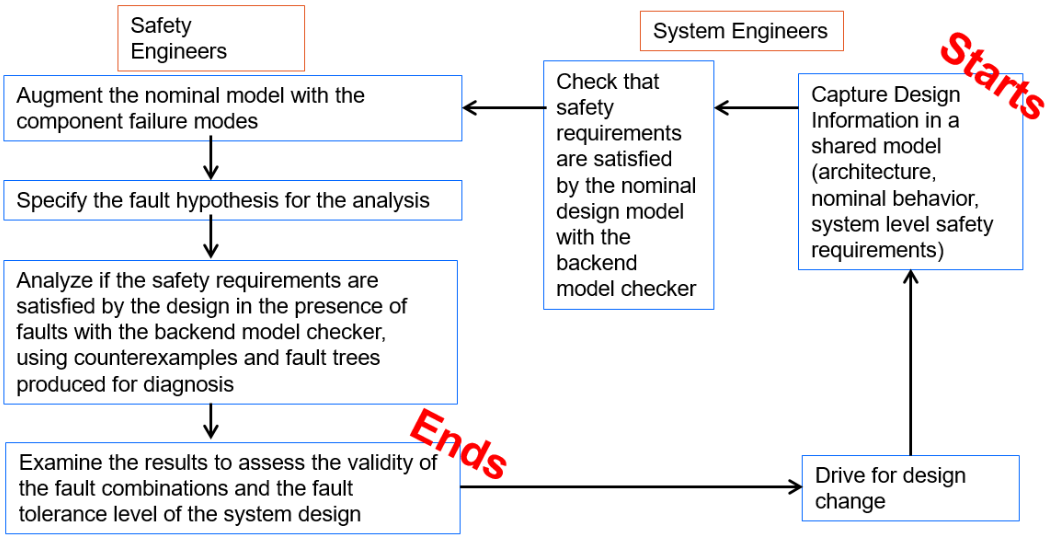

Safety Engineers
System Engineers
Starts
Augment the nominal model with the component failure modes
Check that safety requirements are satisfied by the nominal design model with the backend model checker
Capture Design Information in a shared model (architecture, nominal behavior, system level safety requirements)
Specify the fault hypothesis for the analysis
Analyze if the safety requirements are satisfied by the design in the presence of faults with the backend model checker, using counterexamples and fault trees produced for diagnosis
Ends
Examine the results to assess the validity of the fault combinations and the fault tolerance level of the system design
Drive for design change

Figure 3.1: Proposed Steps of the Safety Assessment Process

There are certain capabilities that are required in order to fully perform all steps of this process. In beginning this research, we outlined what those pieces were and investigated related work to determine if a gap still existed. Based on the related work summary found in Section 2.7, it is clear that this work fills certain gaps that no previous research has addressed:

Shared model using a language expressive enough to describe HW and SW components.

Flexible error propagations through both behavioral and explicit means.

Flexible fault modeling with support for a/symmetric faults, in/dependent faults, etc.

Model checker used to assess and verify the design with or without faults active.

Ability to generate artifacts used in the safety assessment process.

In previous chapters, we discussed how a model checker can be used to generate minimal cut sets in a compositional fashion. In this chapter, the fault modeling process using the Safety Annex for the Architecture Analysis and Design Language (AADL) [4] is described. The Safety Annex was developed with these two broad ideas in mind: (1) how this supports the proposed safety assessment process, and (2) what missing pieces need to be addressed in order to support safety analysts.

3.1 Fault, Failure, and Error Terminology

The usage of the terms error, failure, and fault are defined in ARP4754A and are described here for ease of understanding [110]. 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 the Error Model Annex version 2 for AADL (EMV2) [51], 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 dissertation, 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.

3.2 Component Fault Modeling

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. 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 details on fault definitions and fault modeling capabilities, we refer readers to the Safety Annex Users Guide [119].

Need to either explain the example here or use a different example than WBS. WBS is used A LOT in this section, so perhaps a short description is okay... 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 3.2 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* → *property* in AGREE is true in the initial state and then afterwards it is only true if property holds.)

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 3.3. 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*).

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.

```
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 3.2: An AADL System Type: The Pedal Sensor

```
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 3.3: The Safety Annex for the Pedal Sensor

3.3 Error Propagation

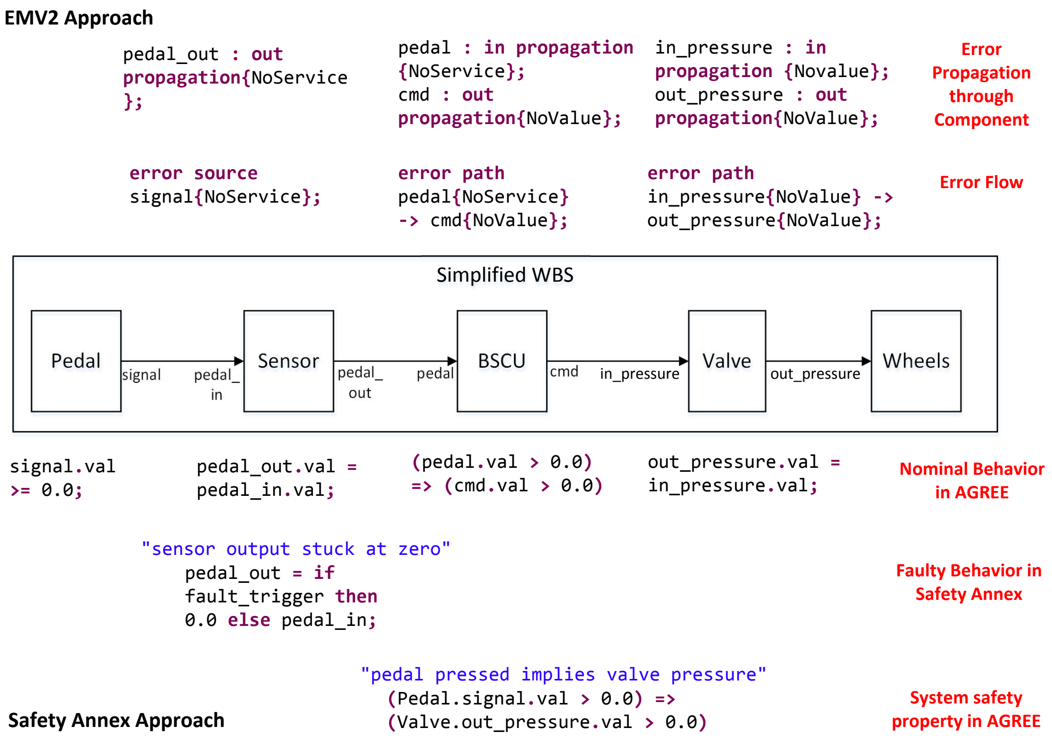
As systems become larger and more complex, it can be difficult knowing all possible error propagations within a model; using a purely explicit approach to error propagation is difficult. To this end, we developed the Safety Annex to primarily use *behavioral* propagation. In this approach, the faults are attached to a component's output and "turned on" in a manner of speaking. The effects and propagation of the active fault is revealed through the behavioral contracts of the system by use of the model checker.

This section outlines the Safety Annex approach to implicit error propagation and also describes how one can model an explicit propagation by defining dependent faults.

3.3.1 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) [51] 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.



```
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 3.4: Differences between Safety Annex and EMV2

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.

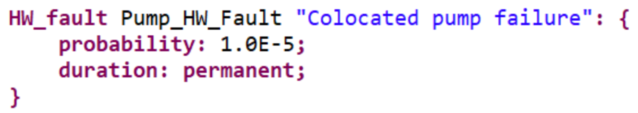
In the EMV2 approach (top half of Figure 3.4), the “NoService” fault is explicitly propagated through all of the components. These fault types are essentially tokens that do not capture any analyzable behavior. At the system level, analysis tools supporting the EMV2 annex can aggregate the propagation 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 3.4), 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*.

3.3.2 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:

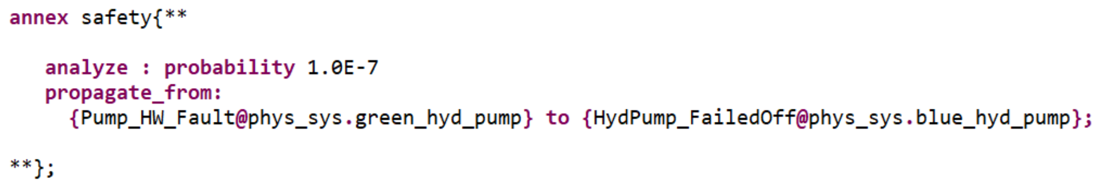


```
HW_fault Pump_HW_Fault "Colocated pump failure": {
    probability: 1.0E-5;
    duration: permanent;
}
```

Figure 3.5: Hardware Fault Definition

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 3.5. The activation of this fault triggers the activation of related faults as seen in the *propagate_to* statement shown in Figure 3.6. Notice that these pumps need not be connected through a data port in order to specify this propagation.



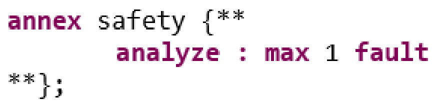
```
annex safety{**

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

**};
```

Figure 3.6: Hardware Fault Propagation Statement

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).

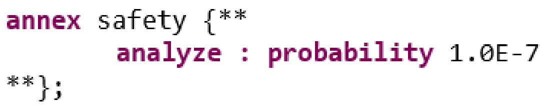
3.4 Fault Analysis Statements

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).

3.5 Asymmetric Fault Modeling

A *asymmetric* or *Byzantine* fault is a fault that presents different symptoms to different observers [44]. 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. A thorough description of the asymmetric modeling capability of the Safety Annex is shown in Chapter 6 using a process ID example.

3.5.1 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 3.7 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 3.7 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 3.7.

An asymmetric fault is defined for Component A as in Figure 3.8. 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

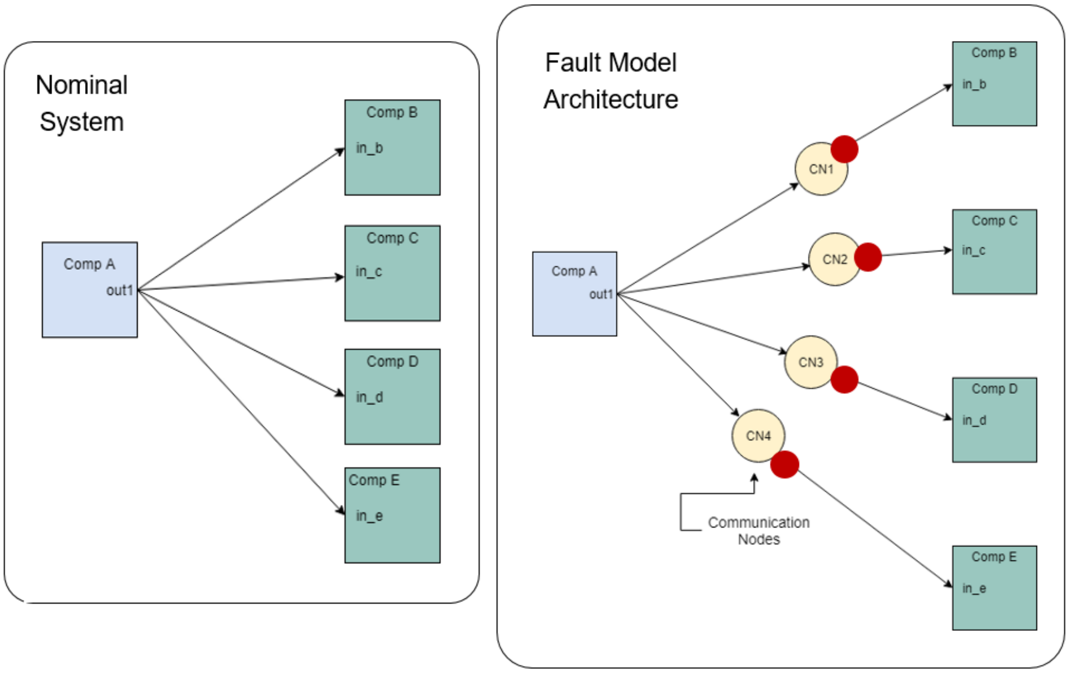
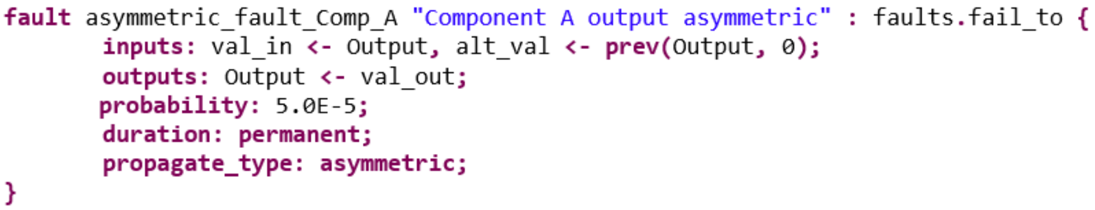


Figure 3.7: Communication Nodes in Asymmetric Fault Implementation



```
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 3.8: Asymmetric Fault Definition in the Safety Annex

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.

3.5.2 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.

This additional feature of the Safety Annex allows users to state contracts of the form: if `sensor_failed` then `do_something`.

3.6 Implementation of the Safety Annex

The Safety Annex is written in Java as a plug-in for the OSATE AADL toolset, which is built on Eclipse. It is not designed as a stand-alone extension of the language, but works with behavioral contracts specified using the AGREE annex for AADL [38]. The architecture of the Safety Annex is shown in Figure 3.9.

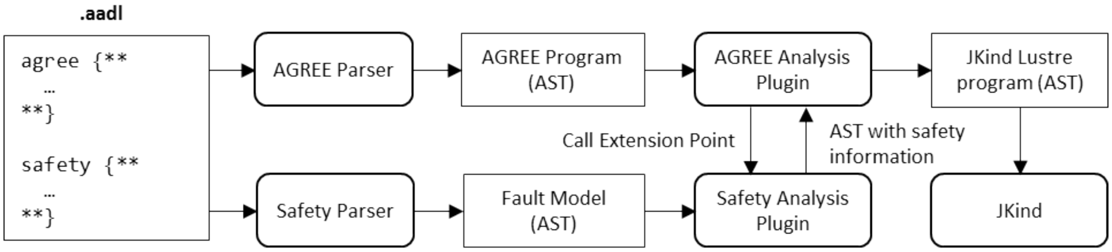
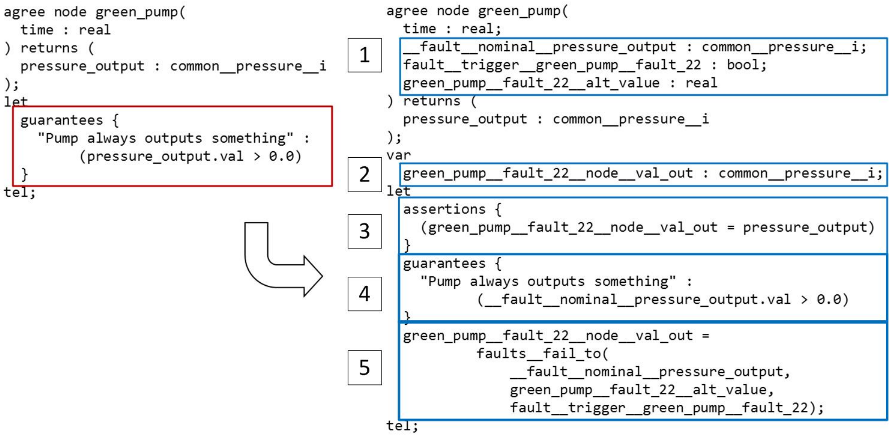


Figure 3.9: Safety Annex Plug-in Architecture

AGREE contracts are used to define the nominal behaviors of system components as *guarantees* that hold when *assumptions* about the values the component's environment are met. When an AADL model is annotated with AGREE contracts and the fault model is created using the Safety Annex, the model is transformed through AGREE into a Lustre model [63] containing the behavioral extensions defined in the AGREE contracts for each system component.

When performing fault analysis, the Safety Annex extends the AGREE contracts to allow faults to modify the behavior of component inputs and outputs. An example of a portion of an initial AGREE node and its extended contract is shown in Figure 3.10. The left column of the figure shows the nominal Lustre pump definition is shown with an AGREE contract on the output; and the right column shows the additional local variables for the fault (boxes 1 and 2), the assertion binding the fault value to the nominal value (boxes 3 and 4), and the fault node definition (box 5). Once augmented with fault information, the AGREE model (translated into the Lustre dataflow language [63]) follows the standard translation path to the model checker JKind [55], an infinite-state model checker for safety properties.



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

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

Figure 3.10: Nominal AGREE Node and Extension with Faults

There are two different types of fault analysis that can be performed on a fault model. The Safety Annex plugin intercepts the AGREE program and add fault model information to the model depending on which form of fault analysis is being run.

Verification in the Presence of Faults: This analysis returns one counterexample when fault activation per the fault hypothesis can cause violation of a property. The augmentation from Safety Annex to the AGREE program includes traceability information so that when counterexamples are displayed to users, the active faults for each component are visualized.

Generate Minimal Cut Sets: This analysis collects all minimal set of fault combinations that can cause violation of a property. As described in Chapter 4, the first step of MinCutSet generation is to collect the minimal IVCs for each property. Given the compositional nature of the verification, each level of the system is extended in a slightly different way. The leaf nodes of a system contribute only constrained faults to the `All_IVCs` algorithm as shown in Figure 4.2.

In the non-leaf layers of the program, both contracts and constrained faults are considered as shown in Figure 4.3. The reason for this is that the contracts are used to prove the properties at the next highest level and are necessary for the verification of the properties.

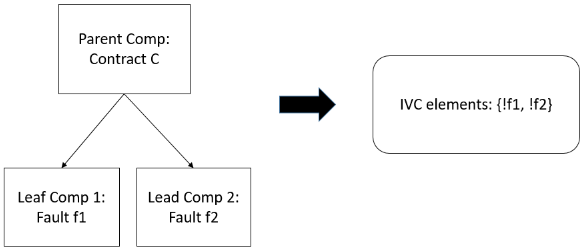


Figure 3.11: IVC Elements used for Consideration in a Leaf Layer of a System

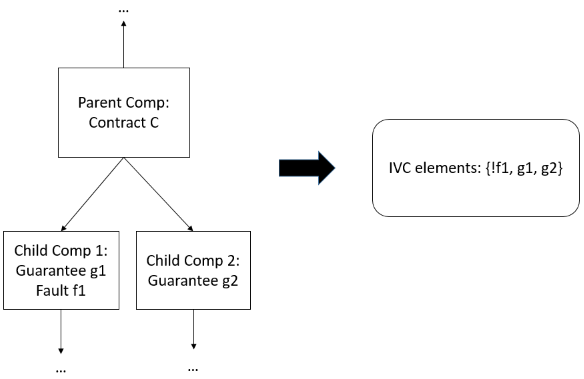


Figure 3.12: IVC Elements used for Consideration in a Middle Layer of a System

The `All_IVCs` algorithm returns the minimal set of these elements necessary to prove the properties. This equates to any contracts or inactive faults that must be present in order for the verification of properties in the model. From here, we transform all MIVCs into minimal cut sets.

Chapter 4

Compositional Minimal Cut Set Generation

Given the difficulty in minimal cut set computations for industrial sized systems, much research has been done on their generation. As described in preliminary sections, the methods of cut set generation have varied greatly depending on how the system is modeled (e.g., transition systems, state machines, decision diagrams) and what kind of model checking is performed (e.g., symbolic algorithms, bounded model checking). Compositional minimal cut set generation has not, to our knowledge, been previously explored.

One of the reasons for this is due to the compositional nature of the verification it is hard to see how faults that are *not* present in the current layer will affect the component behaviors and proofs of the current layer. This information needs to get passed through the layers of the model somehow.

The main contribution of this dissertation is providing a means for the compositional generation of minimal cut sets. In order to perform this kind of computation compositionally, a pre-existing framework was required.

(1) A rich modeling language was needed that allowed for the hierarchical definition of the model. The Architecture Analysis and Design Language (AADL) [108] was chosen; AADL is an SAE International standard language that provides a unifying framework for describing the system architecture for performance-critical, embedded, real-time systems [4,50]. Furthermore, AADL supports language extensions called *annexes* and there are open-source verification options that reason over AADL models.

(2) A compositional verification framework was required; this framework needed to utilize a model checker to verify AADL models. The Assume-Guarantee Reasoning Environment

(AGREE) is a tool for formal analysis of behaviors in AADL models [38]. It is implemented as an AADL annex and annotates AADL components with formal behavioral contracts. Each component's contracts can include assumptions and guarantees about the component's inputs and outputs respectively, as well as predicates describing how the state of the component evolves over time.

AGREE translates an AADL model and the behavioral contracts into the dataflow programming language Lustre [63] and then queries the model checker JKind [55] to conduct the back-end analysis. The analysis can be performed compositionally or monolithically.

(3) A safety specific language was required that allows for the definitions of faults over component outputs and works hand in hand with the model checker to provide safety specific information and proofs about the fault model. In the early stages of this research, the Safety Annex for AADL was developed. The Safety Annex for AADL and its supporting extensions to the AADL tools provide the ability to reason about faults and faulty component behaviors in AADL models [117, 118, 120]. In the Safety Annex approach, formal assume-guarantee (AGREE) contracts are used to define the nominal behavior of system component and the nominal model is verified using AGREE/JKind. The Safety Annex implementation weaves faults into the nominal model and analyzes the behavior of the system in the presence of faults. The tool supports behavioral specification of faults and their implicit propagation through behavioral relationships in the model and provides support to capture binding relationships between hardware and software components of the system. For more information on the Safety Annex, see Chapter 3.

The remainder of this chapter describes compositional minimal cut set generation. First, the formal background and definitions required to understand the approach are supplied, then the proofs and general algorithms are given. The implementation of these algorithms in the Safety Annex are then described.

4.1 The High Level Idea and Approach

Recently, Ghassabani et al. developed an algorithm used in SAT-based model checking that traces a safety property to a minimal set of model elements necessary for proof; this is called the *all minimal inductive validity core* (`All_IVCs`) algorithm [10, 60, 61]. In this dissertation, the IVC information is leveraged to provide insight on not just the minimal elements necessary for a *proof* of the safety property, but the minimal model elements necessary for the *violation* of the safety property. In this case, the model elements included in the reasoning process of the `All_IVCs` algorithm are the faults as well as the contracts of the system. In this way, the

`All_IVCs` algorithm can produce safety related information at each level of the architecture of the system. This information provides a way for compositional generation of minimal cut sets.

Before the specifics of the algorithm and proofs can be discussed, some background definitions are required.

4.2 Definitions

The Boolean Satisfiability (SAT) problem attempts to determine if there exists a total truth assignment to a given propositional formula, that evaluates to TRUE. Generally, a propositional formula is any combination of the disjunction and conjunction of literals (as an example, a and $\neg a$ are literals). For a given unsatisfiable problem, solvers try to generate a proof of unsatisfiability; this is generally more useful than a proof of satisfiability. Such a proof is dependent on identifying a subset of clauses that make the problem unsatisfiable (UNSAT).

SAT solvers in model checking work over a constraint system to determine satisfiability. A *constraint system* C is an ordered set of n abstract constraints $\{C_1, C_2, ..., C_n\}$ over a set of variables. The constraint C_i restricts the allowed assignments of these variables in some way [80]. Given a constraint system, we require some method of determining, for any subset $S \subseteq C$, whether S is *satisfiable* (SAT) or *unsatisfiable* (UNSAT). When a subset S is SAT, this means that there exists an assignment allowed by all $C_i \in S$; when no such assignment exists, S is considered UNSAT. Given a constraint system C, there are certain subsets of C that are of interest in terms of satisfiability. Definitions 2-4 are taken from research by Liffiton et al., [80].

Definition 3. *: A Minimal Unsatisfiable Subset (MUS) M of a constraint system C is a subset $M \subseteq C$ such that M is unsatisfiable and $\forall c \in M : M \setminus \{c\}$ is satisfiable.*

An MUS can be intuitively understood as the minimal explanation of the constraint systems infeasability.

Definition 4. *: A Minimal Correction Set (MCS) M of a constraint system C is a subset $M \subseteq C$ such that $C \setminus M$ is satisfiable and $\forall S \subset M : C \setminus S$ is unsatisfiable.*

A MCS can be seen to “correct” the infeasability of the constraint system by the removal from C the constraints found in an MCS.

A duality exists between the MUSs of a constraint system and the MCSs as established by Reiter [105]. This duality is defined in terms of *Minimal Hitting Sets* (*MHS*). A hitting set of a collection of sets A is a set H such that every set in A is “hit” by H; H contains at least one element from every set in A. Every MUS of a constraint system is a minimal hitting set of the system's MCSs, and likewise every MCS is a minimal hitting set of the system's MUSs [40, 80, 105].

Definition 5. *: Given a collection of sets K, a hitting set for K is a set $H \subseteq \cup_{S \in K} S$ such that $H \cap S \neq \emptyset$ for each $S \in K$. A hitting set for K is minimal if and only if no proper subset of it is a hitting set for K.*

Since we are interested in sets of active faults that cause violation of the safety property, we turn our attention to Minimal Cut Sets. Though these have been previously defined, for convenience the definition is provided again.

Definition 6. *A Minimal Cut Set (MinCutSet) is a minimal collection of faults that lead to the violation of the safety property. Furthermore, any subset of a MinCutSet will not cause this property violation.*

We define a minimal cut set consistently with much of the research in this field [47, 107].

4.3 Formalization of the Method

The main idea that we present in this research is as follows. The MIVCs are MUSs (Minimal Unsatisfiable Subsets) of a constraint system that normally consists of the assumptions and contracts of system components and the negation of the safety property of interest. The MCSs (Minimal Correction Sets), the sets that contain the minimal "correction" of the infeasible constraint system, can then be obtained from all MUSs. Here is the key idea that we present: if the constraint system is defined to take into account fault activation literals as well as component contracts, these MCSs and the information they contain can be transformed into MinCutSets. A small example will illustrate this point nicely.

Let a constraint system C consist of guarantees g_1, g_2, fault activation literals constrained to $false$, $\neg f_1$, $\neg f_2$, and the safety property P also constrained to $false$, $\neg P$.

$$C = \{\neg f_1, \neg f_2, g_1, g_2, \neg P\} \tag{4.1}$$

Now, since we assume that the nominal model proves, it is no surprise that C is UNSAT. The MIVC algorithm returns the minimal unsatisfiable subsets, say $MIVC_1 = \{\neg f_1, g_2\}$ and $MIVC_2 = \{\neg f_2\}$. Let us focus on $MIVC_1$. This can be understood in two ways: (1) if f_1 does not occur and g_2 holds, then the property P can be proven, or (2) $MIVC_1$ is UNSAT, it cannot be the case that f_1 does not occur and the guarantee g_2 holds.

Next we look at the MCSs for this example:

$$MCS_1 = \{\neg f_1, \neg f_2\}, MCS_2 = \{\neg f_2, g_2\}$$

This means that $C \setminus MCS_1$ is SAT. So by removing those constraints from C, we get a SAT constraint system: $C \setminus MCS_1 = \{f_1, f_2, g_1, g_2, \neg P\}$. When both faults f_1 and f_2 are active, $\neg P$ can be proven; i.e., this is a cut set for $\neg P$. The minimality of the MCS gives the minimality of the cut set.

Throughout the remainder of this section, we provide the formal background necessary to show how this approach works.

In the case of a nominal model augmented with faults, a constraint system can be defined as follows. Let F be the set of all fault activation literals defined in the system and G be the set of component contracts (guarantees of component output behavior).

Definition 7. *A constraint system $C = \{C_1, C_2, ..., C_n\}$ where for $i \in \{1, ..., n\}$, C_i has the following constraints for any $f_j \in F$ and $g_k \in G$ with regard to the top level property P:*

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

The `All_IVCs` algorithm collects all minimal unsatisfiable subsets of a given transition system in terms of the *negation* of the top level property [10, 61]. Assuming that the nominal model proves (no faults are active), it is not surprising that the guarantees and the negation of the safety property is UNSAT. The MUSs are the minimal explanation of the infeasibility of this constraint system; equivalently, these are the minimal sets of model elements necessary for proof of the safety property.

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

Because of the duality between MUSs and MCSs, all MCSs can be obtained by finding the hitting sets. The MCS can be seen to correct the infeasibility of the constraint system and provides the minimal such correction. By removing the constraints from C that are found in any MCS, C becomes satisfiable. In terms of the constraint system that includes fault activation literals, by *activating* the faults in the MCS and *violating* the contracts in the MCS, we can prove the *negation* of the property P. This is the precise information we require to find the minimal cut sets of a system. If the contracts in the MCS are replaced with the faults that cause

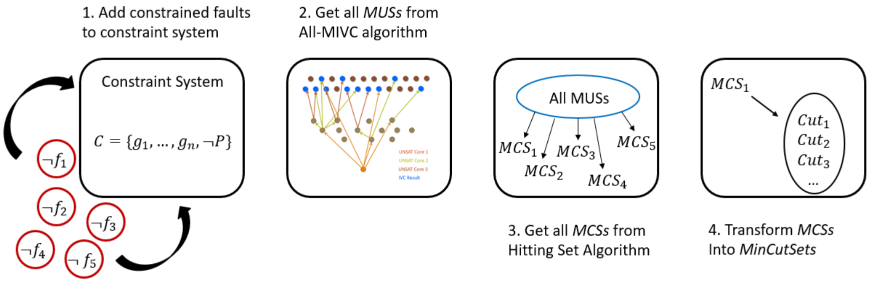

1. Add constrained faults to constraint system
Constraint System
$C = \{g_1, \dots, g_n, \neg P\}$
$\neg f_1$
$\neg f_2$
$\neg f_3$
$\neg f_4$
$\neg f_5$
2. Get all *MUSs* from All-MIVC algorithm
UNSAT Core 1
UNSAT Core 2
UNSAT Core 3
IVC Result
All MUSs
MCS_1
MCS_2
MCS_3
MCS_4
MCS_5
3. Get all *MCSs*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

Figure 4.1: Steps of the Transformation Process

its violation, the MCS is transformed into a MinCutSet. A high level summary of the steps of this transformation are shown in Figure 4.1.

The MCSs are then transformed into Minimal Cut Sets according to the following theoretical results. We assume that the nominal model proves a safety property P. Furthermore, the nominal model consists of all its component contracts, G, and all faults constrained to false, F. Thus, the constraint system $C = F \cup G \cup \{\neg P\}$ is UNSAT.

Lemma 1. *An MCS with all constraints removed proves the negation of the safety property; notationally,* $\overline{MCS} \vdash \neg P$.

Proof. Let $\overline{MCS}$ consist of the elements of MCS with constraints removed and let $C = E \cup \{\neg P\}$ for all model elements E. Since C is UNSAT, clearly $E \nvdash \neg P$.

Since $MCS \subseteq E$, we can further decompose E: let $E = E' \cup MCS$.

It follows easily that since $E \nvdash \neg P$, we know that $E' \nvdash \neg P$ and $MCS \nvdash \neg P$.

By the definition of MCS, $C \setminus MCS \vdash \neg P$.

Let $\overline{MCS}$ be MCS with all constraints removed. Then we can represent $C \setminus MCS$ as the following: $C = E' \cup \overline{MCS} \cup \{\neg P\}$. Since $E' \nvdash \neg P$, it must be the case that $\overline{MCS} \vdash \neg P$. □

In order to transform the MCS into a minimal cut set, any contracts found in the MCS must be replaced with faults that cause their violation. We show that making this replacement still proves $\neg P$.

Assume that there exists a $g \in G$ where $g \in MCS$. Given the assumption that $G \cup \{P\}$ is SAT (i.e., the nominal model proves), $\neg g$ can only occur by activation of faults. For the set of all faults in the system, F, let $F_G \subseteq F$ such that $F_G \vdash \neg g$ and let F_G be a minimal such set (i.e. F_G is a minimal cut set for g). Replace $g \in \overline{MCS}$ with F_G; call this new set MCS_{repl}.

Lemma 2. $MCS_{repl} \vdash \neg P$.

Proof. By the assumption that $F_G \vdash \neg g$ and lemma 1, the result is immediate. □

Lemma 3. *MCS_{repl} is minimal.*

Proof. The minimality follows directly from the definition of MCS.

□

Once all replacements have been made, the set MCS_{repl} contains only faults. Since $MCS_{repl} \vdash \neg P$, these faults are a cut set for $\neg P$. Thus, iterative replacement of each contract in an MCS with a minimal cut set of that contract and removal of all constraints of the elements in MCS results in minimal cut set.

Theorem 1. *A minimal correction set can be transformed into a minimal cut set.*

Proof. Result follows directly from lemmas 1-3. □

For more information on the implementation of these theoretical results, see section 4.4.

4.4 Implementation and Algorithms

The implementation of this idea requires changing what information the `All_IVCs` algorithm uses to complete the proofs and generate MUSs. At each layer of analysis, the `All_IVCs` algorithm views the model as a constraint system consisting of the negation of the property in question (guarantees at lower levels, top-level properties at the highest level), and the supporting guarantees/assumptions from the direct child level. The information provided to this algorithm changes slightly when performing the minimal cut set algorithms.

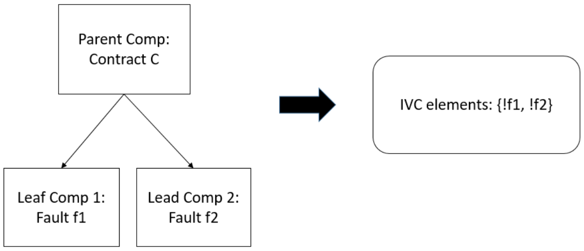


Figure 4.2: IVC Elements used for Consideration in a Leaf Layer of a System

Implementation

In this approach, we use the all MIVCs algorithm and provide it a constraint system consisting of the negation of the top level safety property, the contracts of system components, as well as the faults in each layer constrained to false. It then collects the MUSs of this constraint system.

Different layers of the architecture (and hence proof) provide slightly different information to the `All_IVCs` algorithm. This is “given” to the IVC algorithm by the insertion of a Lustre statement with the keyword `%IVC` followed by the fault activation literal.

```
--%IVC __fault__independently__active__sensor
```

The constraints on that literal are given through the use of an assert statement in Lustre.

```
assert (__fault__independently__active__sensor = false)
```

The leaf nodes contribute only constrained faults to the IVC elements as shown in Figure 4.2. In the non-leaf layers of the program, both contracts and constrained faults are considered as shown in Figure 4.3. The reason for this is that the contracts are used to prove the properties at the next highest level and are necessary for the verification of the properties. The faults are used to provide safety pertinant information for the minimal cut sets.

The all MIVCs algorithm returns the minimal set of these elements necessary to prove the properties. This equates to any contracts or inactive faults that must be present in order for the verification of properties in the model. From here, we perform a number of algorithms to transform all MIVCs into minimal cut sets.

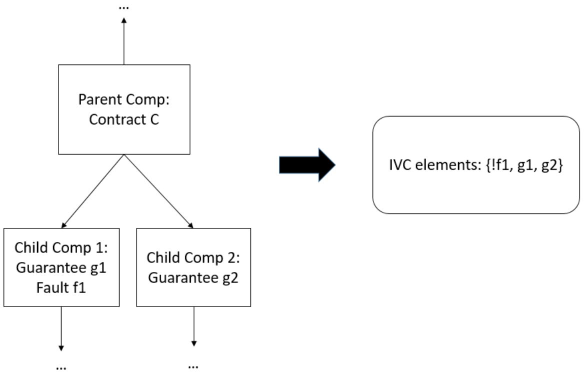


Figure 4.3: IVC Elements used for Consideration in a Middle Layer of a System

Algorithms

The generation of *MIVCs* traverses the program in a top down fashion. Likewise, the transformation of *MIVCs* to MinCutSets traverses this program layer by layer if and only if all MIVCs have been generated. It is a requirement of the minimal hitting set algorithm that *all* MUSs are used to find the MCSs [56, 80, 88]. Thus, once all the MIVCs have been found and the minimal hitting set algorithm has completed, the MinCutSet Generation algorithm can begin.

The MinCutSet Generation Algorithm begins with a list of MCSs specific to a top level property. These MCSs may contain a mixture of fault activation literals constrained to *false* and and subcomponent contracts constrained to *true*. We remove all constraints from each MCS and call the resulting sets I, for *Intermediate* set. Replacement of subcomponent contracts with their respective minimal cut sets can then proceed. For each of those contracts in I, we check to see if we have previously obtained a MinCutSet for that contract. If so, replacement is performed. If not, we recursively call this algorithm to obtain the list of all MinCutSets associated with this subcomponent contract. At a certain point, there will be no more contracts in the set I in which case we have a minimal cut set for the current property. When this set is obtained, we store it in a lookup table keyed by the given property that this I is associated with.

A small example will illustrate this process. Do I want to have a more concrete example? If so, pull from thesis proposal sensor example. If a shorter more abstract example is okay, do that.

Algorithm 1 describes this process.

The number of replacements R that are made in this algorithm are constrained by the number of minimal cut sets there are for all α contracts within the initial MCS.

Algorithm 1: MinCutSets Generation Algorithm

```
Function replace($P$):
    $List(I)$:= $List(MCS)$ for $P$ with all constraints removed ;
    for all $I \in List(I)$ do
        if there exists contracts $g \in I$ then
            for all constrained contracts $g \in I$ do
                if there exists $MinCutSets$ for $g$ in lookup table then
                    for all $minCut(g)$ do
                        $I_{repl} = I$ ;
                        $I_{repl}$ := replace $g$ with $minCut(g)$ ;
                        add $I_{repl}$ to $List(I)$ ;
                else
                    replace($g$) ;
        else
            add $I$ as $minCut(g)$ for $P$ ;
```

We call the set of all minimal cut sets for a contract g: $Cut(g)$. The following formula defines an upper bound on the number of replacements. The validity of this statement follows directly from the general multiplicative combinatorial principle. The number of replacements R is bounded by the following formula:

$$R \leq \sum_{i=1}^{\alpha}(\prod_{j=1}^{i} |Cut(g_j)|) \tag{4.2}$$

It is also important to note that the cardinality of $List(I)$ is bounded, i.e. the algorithm terminates. Every new I that is generated through some replacement of a contract with its minimal cut set is added to $List(I)$ in order to continue the replacement process for all contracts in I. Adding to this set requires proof regarding termination.

Theorem 2. *Algorithm 1 terminates*

Proof. No infinite sets are generated by the `All_IVCs` or minimal hitting set algorithms [61,88]; therefore, every MCS produced is finite. Thus, every $MinCutSet$ of every contract g is finite. Furthermore, a bound exists on the number of additional intermediate sets I that are added to $List(I)$:
$|List(I)| \leq R$ (Equation 4.2). □

The reason for this upper bound is that for a contract g_1 in MCS, we make $|Cut(g_1)|$ replacements and add the resulting lists to $List(I)$. Then we move to the next contract g_2 in I. We must additionally make $|Cut(g_1)| \times |Cut(g_2)|$ replacements and add all of these resulting lists to $List(I)$, and so on throughout all contracts. Through the use of basic combinatorial principles, we end with the above formula for the upper bound on the number of additional intermediate sets.

Pruning to Address Scalability

The MinCutSets are filtered during this process based on a fault hypothesis given before analysis begins. The Safety Annex provides the capability to specify a type of verification in what is called a *fault hypothesis statement*. These come in two forms: maximum number of faults or probabilistic analysis. Algorithm 1 is the general approach, but the implementation changes slightly depending on which form of analysis is being performed. This pruning improves performance and diminishes the problem of combinatorial explosions in the size of minimal cut sets for larger models.

Max N Analysis Pruning This statement restricts the number of faults that can be independently active simultaneously and verification is run with this restriction present. For example, if a max 2 fault hypothesis is specified, two or fewer faults may be active at once. In terms of minimal cut sets, this statement restricts the cardinality of minimal cut sets generated.

If the number of faults in an intermediate set I exceeds the threshold N, any further replacement of remaining contracts in that intermediate set can never decrease the total number of faults in I; therefore, this intermediate set is eliminated from consideration.

Probabilistic Analysis Pruning The second type of hypothesis statement restricts the cut sets by use of a probabilistic threshold. Any cut sets with combined probability higher than the given probabilistic threshold are removed from consideration. The allowable combinations of faults are calculated before the transformation algorithm begins; this allows for a pruning of intermediate sets during the transformation. If the faults within an intermediate set are not a subset of any allowable combination, that intermediate set is pruned from consideration and no further replacements are made.

Chapter 5

Granularity

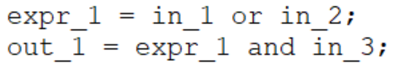
As anyone in requirements engineering would state, the way requirements are stated matters. In the AADL/AGREE/Safety Annex approach of system development and safety assessment of models, we capture these requirements in *guarantees* and *assumptions* for the system components and the safety property is written as a top level guarantee or lemma. The process of verification proceeds by attempting to prove the safety property by use of the guarantees and assumptions at the lower system levels. A question that has come up in previous research work is how the formulation of the guarantees can affect the IVC analysis results [59].

As described in Chapter 2, a transition relation is considered to be a conjunction of Boolean formulas. The granularity of these formulas will substantially affect the analysis results. Depending on how contracts are specified in the model, it is possible to have a "complete" specification, i.e., all of the equations in the model are required to determine the validity of the property. However, in certain cases, subexpressions of equations may be irrelevant. If the equation is decomposed into smaller pieces, this incompleteness becomes visible and the model is no longer completely covered.[1] It is often the case that splitting an equation of the model into more conjuncts, or equivalently, making the model more *granular*, leads to lower coverage of the model. This would be reflected in the IVCs generated for a given safety property; the IVCs in a more granular model would theoretically reflect only the necessary equations required for property verification, and thus would provide more specific analysis results.

Interestingly, similar but opposite work has been done in test case generation, specifically *Modified Condition and Decision Coverage* (MC/DC). It was found that MC/DC over implementations with structurally complex Boolean expressions are generally larger and more effective than MC/DC over functionally equivalent but structurally simpler implementations [57]. An

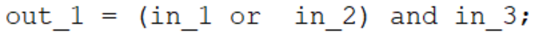
[1] Simply put, coverage is a metric that determines how well properties cover the design of a model.

automated technique called *inlining* provides a restructuring of the model by inlining simpler Boolean expressions into a single, now more complex, expression. An example of an unlined implementation is:



```
expr_1 = in_1 or in_2;
out_1 = expr_1 and in_3;
```

And the associated inlined implementation is:



```
out_1 = (in_1 or  in_2) and in_3;
```

Inlining results in a behaviorally equivalent implementation with different structure. The reason MC/DC provides much greater coverage in terms of test case generation is because MC/DC on an inlined system will require specific combinations of input that will not be required to achieve coverage of the noninlined system [57].

Inductive validity cores, on the other hand, attempt to answer a different kind of question about the model than test coverage. In the IVC case, the goal is to find the minimal sets of model elements that contribute to a proof of a safety property. When these model elements are pulled from the model in terms of guarantees and assumptions, the *granularity* of these logical statements matters in the opposite way. Instead of making the contracts more complex (inlining), the interest here is to simplify the contracts (un-inlining). In this way, the IVCs are more specific with regard to the requirement parts.

Granularity of contracts for IVCs has been previously discussed by Ghassabani [59], but to our knowledge has not been discussed in any other previous work – in particular related to minimal cut set generation. Since IVC generation is a required first step of our minimal cut set algorithms, it is important to discuss how the granularity of the model will affect the cut sets generated through this approach.

As described in Chapter 4, the backend model checker used in this transformation is `JKind` which performs k-induction over a transition system defined with a Lustre program. Ghassabani did a preliminary investigation of granularity within the context of the Lustre language which provides a nice formalism for this discussion because it is top-level conjunctive, equational, and *referentially transparent* [63]. This means that the behavior of a Lustre program is defined by a system of equations and any subexpression on the right side of an equation can be extracted and assigned a fresh variable[2] which is substituted into the original equation without changing the meaning of the program. In this context, *granular refinement* is defined as an extraction of a subexpression into a new equation assigning a fresh variable.

[2] A fresh variable is a variable with an identifier that has not been used within the program.

5.1 Illustrative Example

To see how different representations of the system will alter the IVC and MinCutSet computations, let us examine a simple sensor example system. In this simple AADL architecture, the environmental temperature and pressure is sent to a subsystem of sensors which contains a temp sensor and a pressure sensor, both of which receive the respective environmental inputs. If the temperature (or pressure) surpasses a given threshold, then the temp (pressure) sensor outputs a HIGH_LEVELS flag. It also outputs the temperature (pressure) reading seen at the sensor. A diagram showing the two levels of the AADL implementation is shown in Figure 5.1.

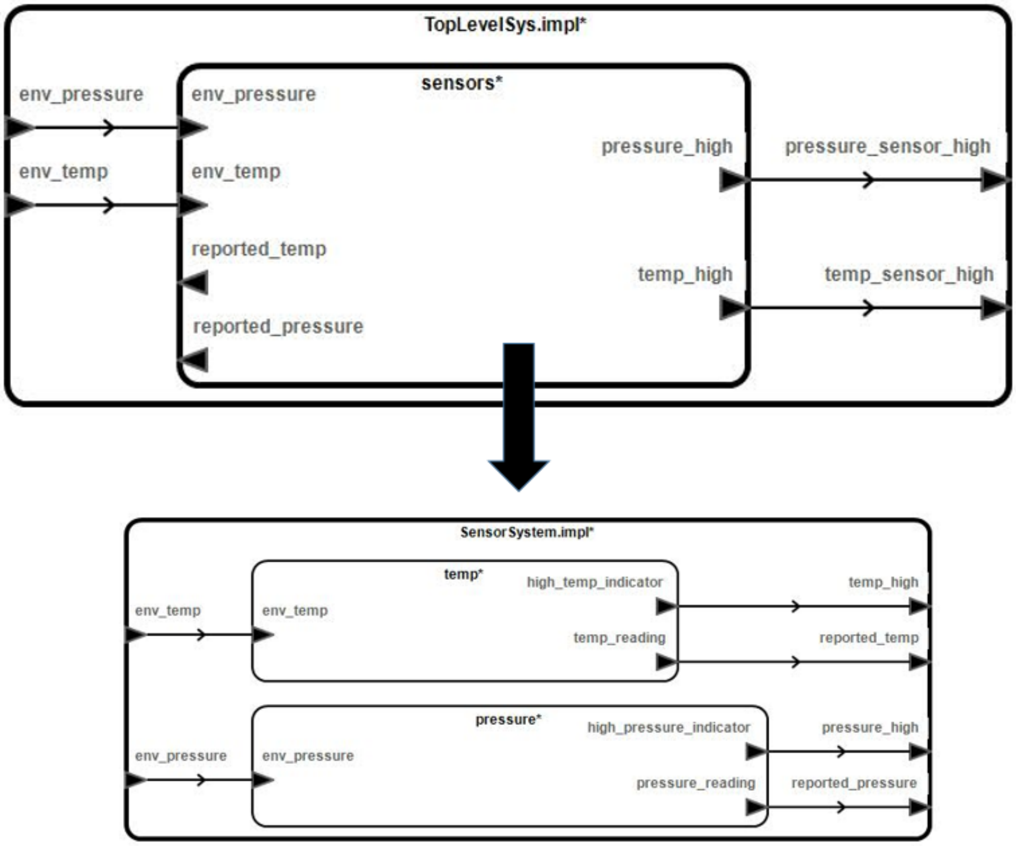


Figure 5.1: Tempurature Sensor System

Now that the basic architecture is in place, we focus our attention on the requirements of each component. The behavior we wish to enforce at the temperature sensor level corresponds to the following two guarantees, G_1 and G_2. (Note: pressure sensor behavior is quite similar and for this reason, we will stick to the temperature for this example.)

G_1: If environmental temperature surpasses threshold, then output high temperature indication: $(temp > THRESHOLD) \implies (high\_temp\_indicator)$

G_2: Temperature read is equivalent to temperature in[3]: $temp = temp\_reading$

These can be seen in the model as two distinct guarantees over the output of the sensor component. Now, as the contracts work their way up the system, there are distinct ways of writing them. For this, we will look to two metaphorical engineers who will provide the higher layer contracts to us.

Let us assume that system A is built by engineer A. The top level safety property states:

If environmental temperature reaches 90 degrees, then system reports high temperature.

The direct subcomponent is the sensor system which contains the outputs: (1) a high temp indicator, and (2) the actual temperature. Engineer A chose to write the supporting contract in the subsystem as follows:

$$(T \geq 90 \implies temp\_high) \wedge (temp\_indicator = T)$$

The example temp sensor system contract hierarchy is shown in Figure 5.2.

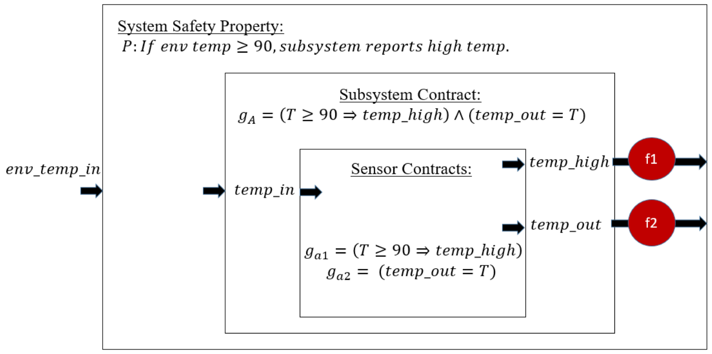

System Safety Property:
P: If env temp ≥ 90, subsystem reports high temp.
Subsystem Contract:
$g_A = (T \geq 90 \Rightarrow temp\_high) \wedge (temp\_out = T)$
env_temp_in
temp_in
Sensor Contracts:
temp_high
temp_out
$g_{a1} = (T \geq 90 \Rightarrow temp\_high)$
$g_{a2} = (temp\_out = T)$
f1
f2

Figure 5.2: Temp Sensor System Contract Part I

The safety property at the top level requires the contract g_A for proof of validity. Thus, the `All_IVCs` should contain the contract g_A as an IVC.

[3] This example eliminated the possibility of noise in the temp reading for simplicity's sake.

There are two faults defined for the temperature subsystem; one for each of the outputs. Fault f_1 affects the $temp\_high$ output and fault f_2 affects the $temp\_out$ output. Since each of these faults will violate the contract g_A, each of them should be found in the *MinCutSet* for G_A.

Now assume that Figure 5.3 was the system contract representation built by engineer B.

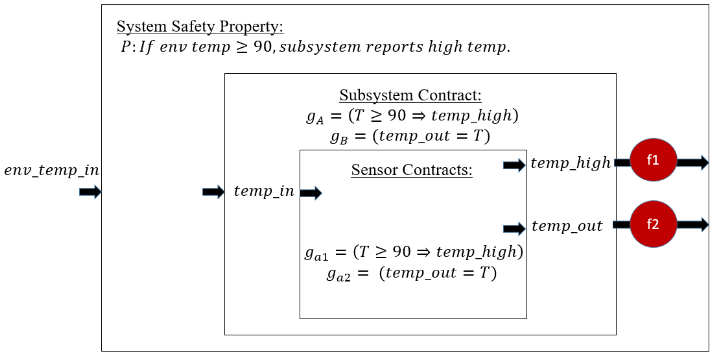

System Safety Property:
P: If env temp ≥ 90, subsystem reports high temp.
Subsystem Contract:
$g_A = (T \geq 90 \Rightarrow temp\_high)$
$g_B = (temp\_out = T)$
env_temp_in
temp_in
Sensor Contracts:
$g_{a1} = (T \geq 90 \Rightarrow temp\_high)$
$g_{a2} = (temp\_out = T)$
temp_high
temp_out
f1
f2

Figure 5.3: Temp Sensor System Contract Part II

The behavior and architecture of the system is the same, but the contract for the subsystem is more *granular*; it is stated as two separate contracts:

$$g_A = (T \geq 90 \implies temp\_high)$$
$$g_B = (temp\_indicator = T)$$

Since g_B is not required for the proof of the system safety property, only g_A is found in the *IVC*s and thus only f_1 will be seen in the *MinCutSets* for this particular contract.

In this simple example, it is easy to see how the granularity of the contracts may greatly affect the results of analysis.

5.2 Algorithms and Results

Introduce the section and give outline or brief summary of what it includes.

5.2.1 Algorithms

As described in section whatever section I outline activation literals, the IVC algorithm requires specific model elements to be considered for analysis. In that section, the additional elements of fault activation literals were added to the IVC analysis. In this case, we look at both the guarantees and the faults that are added for IVC consideration.

The simplest restructuring that could be done on the model was to split any guarantees containing an $\wedge$ at the highest level of the binary Boolean statement and create additional guarantees from this split. For instance, if GUARANTEE0 = A and B, then split this into GUARANTEE1 = A, GUARANTEE2 = B. This provided an easy way to test that our assumption was correct regarding the types of minimal cut sets generated given various forms of contracts. Algorithm 2 shows this process.

Algorithm 2: Split guarantees on logical AND operator

```
Function $splitOnAnd$ (expression):
    Program $P$ ;
    Guarantees $list_G$ ;
    for all $g \in list_G$ do
        if binary statement with operator $\wedge$ then
            insert into $P \rightarrow$ new guarantee (left) ;
            insert into $P \rightarrow$ new guarantee (right) ;
            splitOnAnd(left) ;
            splitOnAnd(right) ;
```

5.2.2 Results

5.2.3 Discussion

Chapter 6

Case Studies

6.1 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 [16,21,23, 71]. The preliminary work for the Safety Annex was based on a simple model of the WBS [120]. 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 [23].

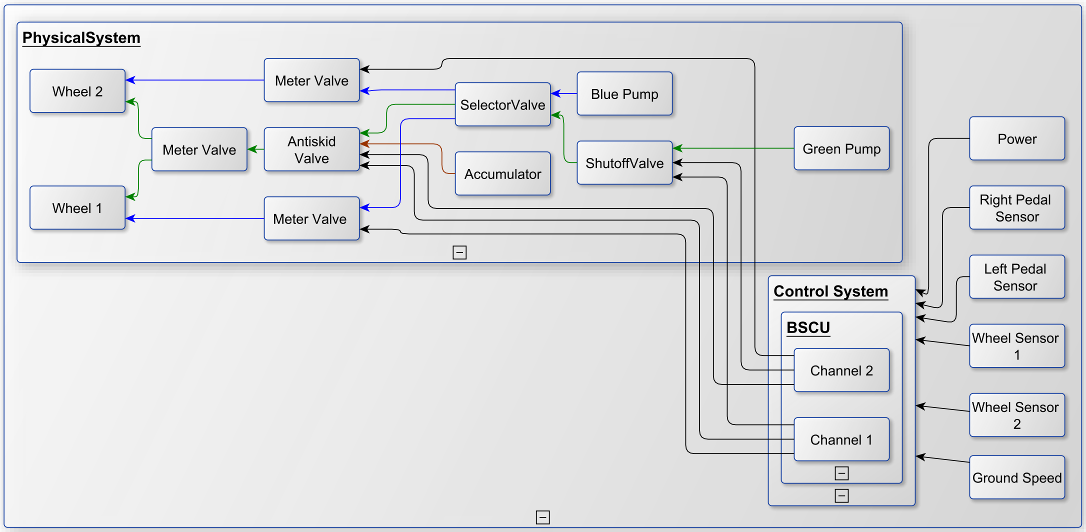


Figure 6.1: Simplified Two-wheel Wheel Brake System

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 6.1 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 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*.

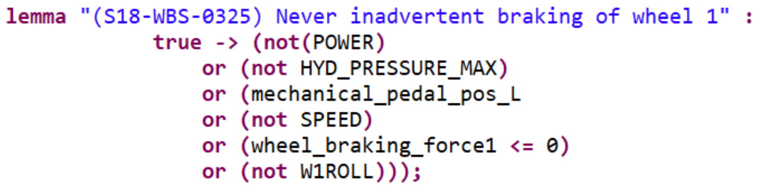
- In *alternate* mode, the system is composed of a *blue* hydraulic pump, four meter valves, and four antiskid shutoff valves, one for each landing gear. The meter valves are mechanically commanded through the pilot pedal corresponding to each landing gear. If the selector detects lack of pressure in 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 6.2.



```
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 6.2: AGREE Contract for Top Level Property: Inadvertent Braking

6.1.1 Nominal Model Analysis

Before performing fault analysis, users should first check that the safety properties are satisfied by the nominal design model. This analysis can be performed monolithically or compositionally in AGREE. Using monolithic analysis, the contracts at the lower levels of the architecture are flattened and used in the proof of the top level safety properties of the system. Compositional analysis, on the other hand, will perform the proof layer by layer top down, essentially breaking the larger proof into subsets of smaller problems. For a more comprehensive description of these types of proofs and analyses, see additional publications related to AGREE [5, 37]

The WBS has a total of 13 safety properties at the top level that are supported by subcomponent assumptions and guarantees. These are shown in Table 6.1. Given that there are 8 wheels, contract S18-WBS-0325-wheelX is repeated 8 times, one for each wheel. The behavioral model in total consists of 36 assumptions and 246 supporting guarantees.

6.1.2 Fault Model Analysis

There are two main options for fault model analysis using the Safety Annex. The first option injects faulty behavior allowed by faulty hypothesis into the AGREE model and returns this model to JKind for analysis. This allows for the activity of faults within the model and traceability information provides a way for users to view a counterexample to a violated contract in the presence of faults. The second option is used to generate minimal cut sets for the model. The model is annotated with fault activation that are constrained to false as well as intermediate level guarantees as model elements for consideration for the all Minimal Inductive Validity

S18-WBS-R-0321 Loss of all wheel braking during landing or RTO shall be less than 5.0×10^{-7} per flight.
S18-WBS-R/L-0322 Asymmetrical loss of wheel braking (Left/Right) shall be less than 5.0×10^{-7} per flight.
S18-WBS-0323 Never inadvertent braking with all wheels locked shall be less than 1.0×10^{-9} per takeoff.
S18-WBS-0324 Never inadvertent braking with all wheels shall be less than 1.0×10^{-9} per takeoff.
S18-WBS-0325-wheelX Never inadvertent braking of wheel X shall be less than 1.0×10^{-9} per takeoff. .

Table 6.1: Safety Properties of WBS

Cores (All-MIVCs) algorithm. The All-MIVCs traces the minimal set of model elements used to produce minimal cut sets. This subsection presents these options and discusses the analytical results obtained.

Verification in the Presence of Faults: Max N Analysis

Using a max number of faults for the hypothesis, the user can constrain the number of simultaneously active faults in the model. The faults are added to the AGREE model for the verification. Given the constraint on the number of possible simultaneously active faults, the model checker attempts to prove the top level properties given these constraints. If this cannot be done, the counterexample provided will show which of the faults (N or less) are active and which contracts are violated.

The user can choose to perform either compositional or monolithic analysis using a max N fault hypothesis. In compositional analysis, the analysis proceeds in a top down fashion. To prove the top level properties, the properties in the layer directly beneath the top level are used to perform the proof. The analysis proceeds in this manner. Users constrain the maximum number of faults within each layer of the model by specifying the maximum fault hypothesis statement to that layer. If any lower level property failed due to activation of faults, the property verification at the higher level can no longer be trusted because the higher level properties were proved based on the assumption that the direct sub-level contracts are valid. This form of analysis is helpful to see weaknesses in a given layer of the system.

In monolithic analysis the layers of the model are flattened, which allows a direct correspondence between all faults in the model and their effects on the top level properties. As with compositional analysis, a counterexample shows these N or less active faults.

Verification in the Presence of Faults: Probabilistic Analysis

Given a probabilistic fault hypothesis, this corresponds to performing analysis with the combinations of faults whose occurrence probability is less than the probability threshold. This is done by inserting assertions that allow those combinations in the Lustre code. If the model checker proves that the safety properties can be violated with any of those combinations, one of such combination will be shown in the counterexample.

Probabilistic analysis done in this way must utilize the monolithic AGREE option. For compositional probabilistic analysis, see Section 12.

To perform this analysis, it is assumed that the non-hardware faults occur independently and possible combinations of faults are computed and passed to the Lustre model to be checked by the model checker. As seen in Algorithm 1, the computation first removes all faults from consideration that are too unlikely given the probability threshold. The remaining faults are arranged in a priority queue $\mathcal{Q}$ from high to low. Assuming independence in the set of faults, we take a fault with highest probability from the queue (step 5) and attempt to combine the remainder of the faults in $\mathcal{R}$ (step 7). If this combination is lower than the threshold (step 8), then we do not take into consideration this set of faults and instead remove the tail of the remaining faults in $\mathcal{R}$.

Algorithm 3: Monolithic Probability Analysis

```
$\mathcal{F} = \{\}$ : fault combinations above threshold ;
$\mathcal{Q}$ : faults, $q_i$, arranged with probability high to low ;
$\mathcal{R} = \mathcal{Q}$ , with $r \in \mathcal{R}$;
while $\mathcal{Q} \neq \{\} \wedge \mathcal{R} \neq \{\}$ do
    $q =$ removeTopElement($\mathcal{Q}$) ;
    for $i = 0 : |\mathcal{R}|$ do
        $prob = q \times r_i$ ;
        if $prob < threshold$ then
            removeTail($\mathcal{R}, j = i : |\mathcal{R}|$);
        else
            add($\{q, r_i\}, \mathcal{Q}$);
            add($\{q, r_i\}, \mathcal{F}$);
```

In this calculation, we assume independence among the faults, but in the Safety Annex it is possible to define dependence between faults using a fault propagation statement. After fault combinations are computed using Algorithm 3, the triggered dependent HW faults are added to the combination as appropriate. The dependencies are implemented in the *Verify in the Presence of Faults* options for analysis, but not yet implemented in the *Generate Minimal Cut Sets* analysis options.

Generate Minimal Cut Sets: Max N Analysis

As described in Chapter 4, the generation of MinCutSets uses the All-MIVCs algorithm to provide a full enumeration of all minimal set of model elements necessary for the proof of each top-level safety property in the model, and then transforms all MIVCs into all minimal cut sets. In Max N analysis, the minimal cut sets are pruned to include only those with at cardinality less or equal to the max N number specified in the fault hypothesis and displayed to the user.

Generate MinCutSet analysis was performed on the Wheel Brake System and results are shown in Table 6.2. Notice in Table 6.2, the label across the top row refers to the cardinality (C) and the corresponding column shows how many cut sets are generated of that cardinality. When the analysis is run, the user specifies the value N. This gives cut sets of cardinality *less than or equal to* N. (For the full text of the properties, see Table 6.1.)

Property	$c = 1$	$c = 2$	$c = 3$	$c = 4$	$c = 5$	$c = 6$	$c = 7+$
R-0321	6	0	0	1	144	7776	-
R-0322	32	0	0	0	0	0	-
L-0322	32	0	0	0	0	0	-
0323	90	0	0	0	0	0	-
0324	8	3,401	6,800	66,472	435,358	1,892,832	-
0325-WX	20	0	0	0	0	0	-

Table 6.2: WBS MinCutSet Analysis Results for Cardinality c

Due to the increasing number of possible fault combinations at $N = 6$, the computational time increases quickly. The WBS analysis was only run to $N = 6$ for this reason.

Generate Minimal Cut Sets: Probabilistic Analysis

Both probabilistic analysis and max N analysis use the same minimal cut set generation algorithm, except that in probabilistic analysis, the minimal cut sets are pruned to include only those fault combinations whose probability of simultaneous occurrence exceed the given threshold in the probability hypothesis. Note that with probablistic hypothesis, *Verify in the Presence of Faults* is performed using only monolithic analysis, but generating minimal cut sets is performed using compositional analysis.

The probabilistic analysis for the WBS was given a top level threshold of 1.0×10^{-9} as stated in AIR6110. The faults associated with various components were all given probability of occurrence compatible with the discussion in this same document.

As shown in Table 6.3, the number of allowable combinations drops considerably when given probabilistic threshold as compared to just fault combinations of certain cardinalities. For example, one contract (inadvertent wheel braking of all wheels) had over a million minimal cut sets produced when looking at it in terms of max N analysis, but after taking probabilities into account, it is seen that only one combination of faults can violate this property. (For the full text of the properties, see Table 6.1.)

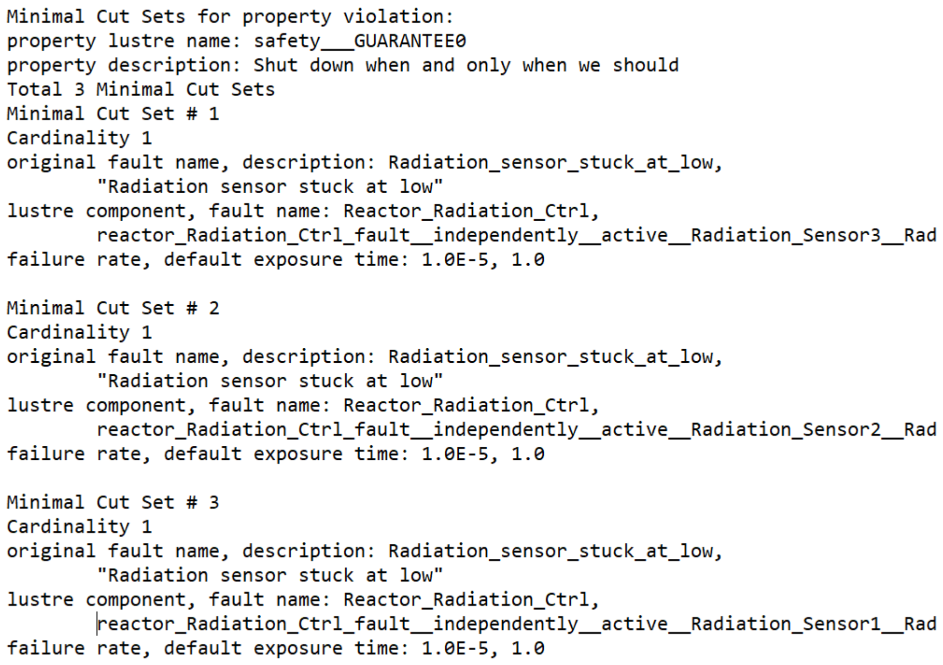
Results from Generate Minimal Cut Sets

Results from Generate Minimal Cut Sets analysis can be represented in one of the following forms.

Property	$c = 1$	$c = 2$	$c = 3$	$c = 4$	$c = 5$	$c = 6$	$c = 7$	$c = 8$
R-0321	0	0	0	0	0	0	0	0
R-0322	32	0	0	0	0	0	0	0
L-0322	32	0	0	0	0	0	0	0
0323	90	0	0	0	0	0	0	0
0324	0	1	0	0	0	0	0	0
0325-WX	20	0	0	0	0	0	0	0

Table 6.3: WBS MinCutSet Results for Probabilistic Analysis

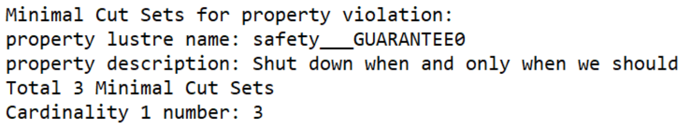
1. The minimal cut sets can be presented in text form with the total number per property, cardinality of each, and description strings showing the property and fault information. A sample of this output is shown in Figure 6.3.



```
Minimal Cut Sets for property violation:
property lustre name: safety___GUARANTEE0
property description: Shut down when and only when we should
Total 3 Minimal Cut Sets
Minimal Cut Set # 1
Cardinality 1
original fault name, description: Radiation_sensor_stuck_at_low,
        "Radiation sensor stuck at low"
lustre component, fault name: Reactor_Radiation_Ctrl,
        reactor_Radiation_Ctrl_fault__independently__active__Radiation_Sensor3__Rad
failure rate, default exposure time: 1.0E-5, 1.0

Minimal Cut Set # 2
Cardinality 1
original fault name, description: Radiation_sensor_stuck_at_low,
        "Radiation sensor stuck at low"
lustre component, fault name: Reactor_Radiation_Ctrl,
        reactor_Radiation_Ctrl_fault__independently__active__Radiation_Sensor2__Rad
failure rate, default exposure time: 1.0E-5, 1.0

Minimal Cut Set # 3
Cardinality 1
original fault name, description: Radiation_sensor_stuck_at_low,
        "Radiation sensor stuck at low"
lustre component, fault name: Reactor_Radiation_Ctrl,
        reactor_Radiation_Ctrl_fault__independently__active__Radiation_Sensor1__Rad
failure rate, default exposure time: 1.0E-5, 1.0
```

Figure 6.3: Detailed Output of MinCutSets

2. The minimal cut set information can be presented in tally form. This does not contain the fault information in detail, but instead gives only the tally of cut sets per property. This is useful in large models with many cut sets as it reduces the size of the text file. An example of this output type is seen in Figure 6.4.



```
Minimal Cut Sets for property violation:
property lustre name: safety___GUARANTEE0
property description: Shut down when and only when we should
Total 3 Minimal Cut Sets
Cardinality 1 number: 3
```

Figure 6.4: Tally Output of MinCutSets

3. The tool can also generate fault tree and minimal cut set information formatted as input to the SOTERIA tool [84] to produce hierarchical fault trees that are consistent with the system architecture/component verification layers, or flat fault trees consist of minimal cut sets only, both in graphical form. A sample graphical fault tree output from the SOTERIA tool is shown in Figure 6.5. The SOTERIA tool is also able to compute the probabilities for the top level event from a given fault tree. However, based on experience with the WBS example, our tool was a more scalable solution as it produces minimal cut sets for more complex systems, also in shorter amount of time. The text format of the minimal cut sets seemed anectodally easier to read than the graphical format for larger systems.

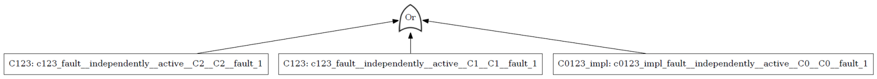

Or
C123: c123_fault__independently__active__C2__C2__fault_1
C123: c123_fault__independently__active__C1__C1__fault_1
C0123_impl: c0123_impl_fault__independently__active__C0__C0__fault_1

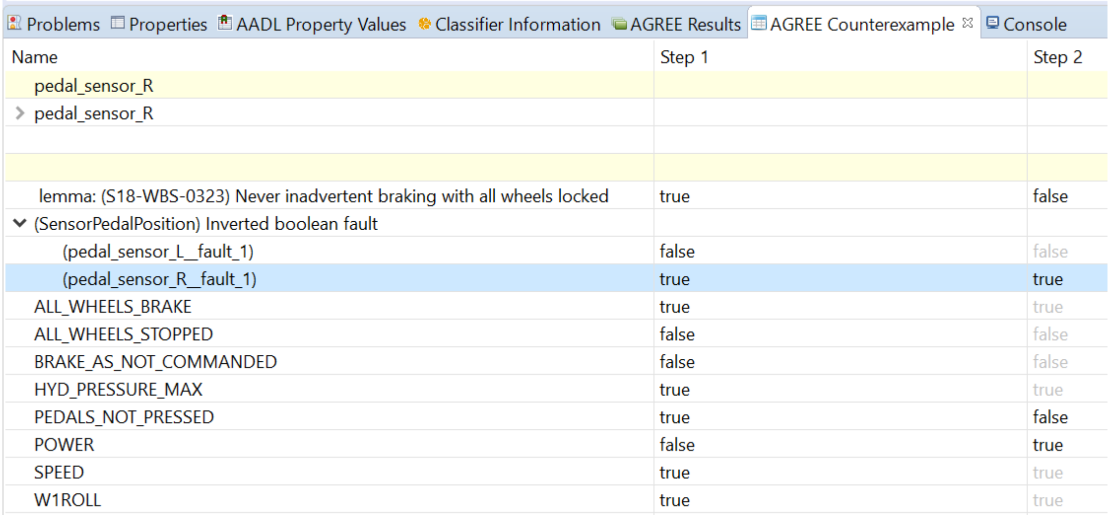
Figure 6.5: Example SOTERIA Fault Tree

Use of Analysis Results to Drive Design Change

We use a single top level requirement of the WBS: S18-WBS-0323 (Never indadvertent braking with all wheels locked) to illustrate how Safety Annex can be used to detect design flaws and how faults can affect the behavior of the system). Upon running max n compositional fault analysis with $n = 1$, this particular fault was shown to be a single point of failure for this safety property. A counterexample is shown in Figure 6.6 showing the active fault on the pedal sensor.

Depending on the goals of the system, the architecture currently modeled, and the mitigation strategies that are desired, various strategies are possible to mitigate the problem.

- Possible mitigation strategy 1: Monitor system can be added for the sensor: A monitor sub-component can be modeled in which it accesses the mechanical pedal as well as the signal from the sensor. If the monitor finds discrepancies between these values, it



Name	Step 1	Step 2
pedal_sensor_R		
pedal_sensor_R		
lemma: (S18-WBS-0323) Never inadvertent braking with all wheels locked	true	false
(SensorPedalPosition) Inverted boolean fault		
(pedal_sensor_L__fault_1)	false	false
(pedal_sensor_R__fault_1)	true	true
ALL_WHEELS_BRAKE	true	true
ALL_WHEELS_STOPPED	false	false
BRAKE_AS_NOT_COMMANDED	false	false
HYD_PRESSURE_MAX	true	true
PEDALS_NOT_PRESSED	true	false
POWER	false	true
SPEED	true	true
W1ROLL	true	true

Figure 6.6: AGREE counterexample for inadvertent braking safety property

can send an indication of invalid sensor value to the top level of the system. In terms of the modeling, this would require a change to the behavioral contracts which use the sensor value. This validity would be taken into account through the means of $valid \land pedal\_sensor\_value$.

- Possible mitigation strategy 2: Redundancy can be added to the sensor: A sensor subsystem can be modeled which contains 3 or more sensors. The overall output from the sensor system may utilize a voting scheme to determine validity of sensor reading. There are multiple voting schemes that are possible, one of which is a majority voting (e.g. one sensor fails, the other two take majority vote and the correct value is passed). When three sensors are present, this mitigates the single point of failure problem. New behavioral contracts are added to the sensor system to model the behavior of redundancy and voting.

In the case of the pedal sensor in the WBS, the latter of the two strategies outlined above was implemented. A sensor system was added to the model which held three pedal sensors. The output of this subsystem was constrained using a majority voting scheme. Upon subsequent runs of the analysis (regardless which type of run was used), resilience was confirmed in the system regarding the failure of a single pedal sensor. Figure 6.7 outlines these architectural changes that were made in the model.

As can be seen through this single example, a system as large as the WBS would benefit from many iterations of this process. Furthermore, if the model is changed even slightly on the system development side, it would automatically be seen from the safety analysis perspective

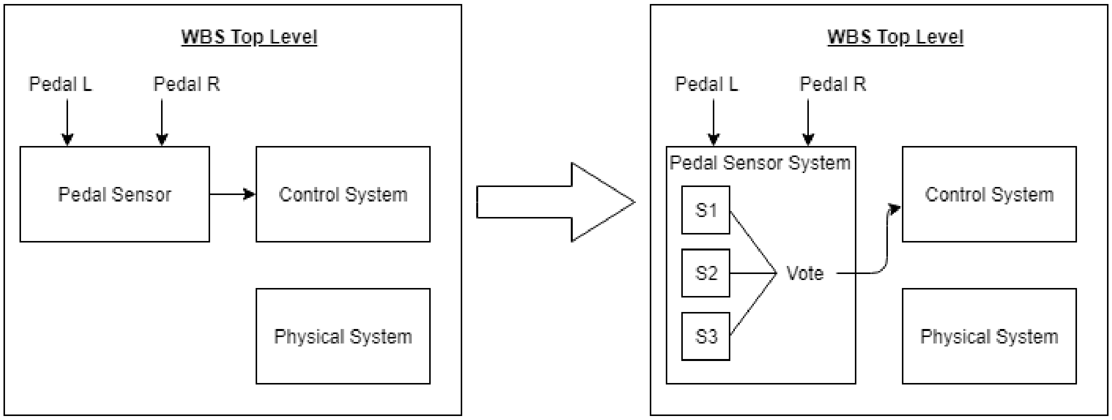


Figure 6.7: Changes in the architectural model for fault mitigation

and any negative outcomes would be shown upon subsequent analysis runs. This effectively eliminates any miscommunications between the system development and analysis teams and creates a new safeguard regarding model changes.

For more information on types of fault models that can be created as well as details on analysis results, see the users guide located in the GitHub repository [119]. This repository also contains all models used in this project.

6.2 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.

6.2.1 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 [29, 44], 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 6.8.

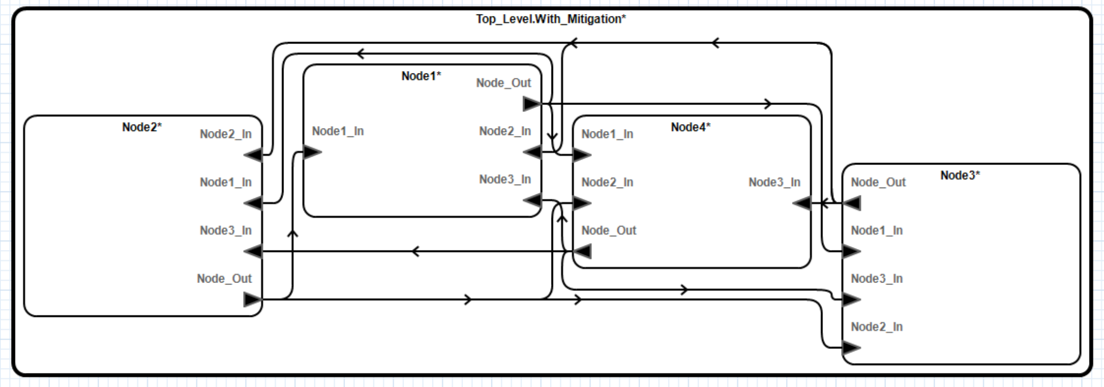
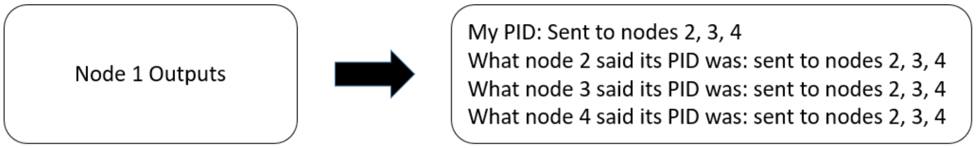


Figure 6.8: Updated PID Example Architecture

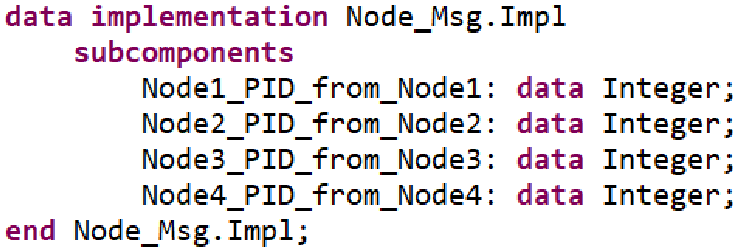
Each node reports its own PID to all other nodes in the first round of communication. In the second round, each node informs the others what they saw in terms of everyone's PIDs. The outputs from a node are described in Figure 6.9. These outputs are modeled as a nested data


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 6.9: Description of the Outputs of Each Node in the PID Example

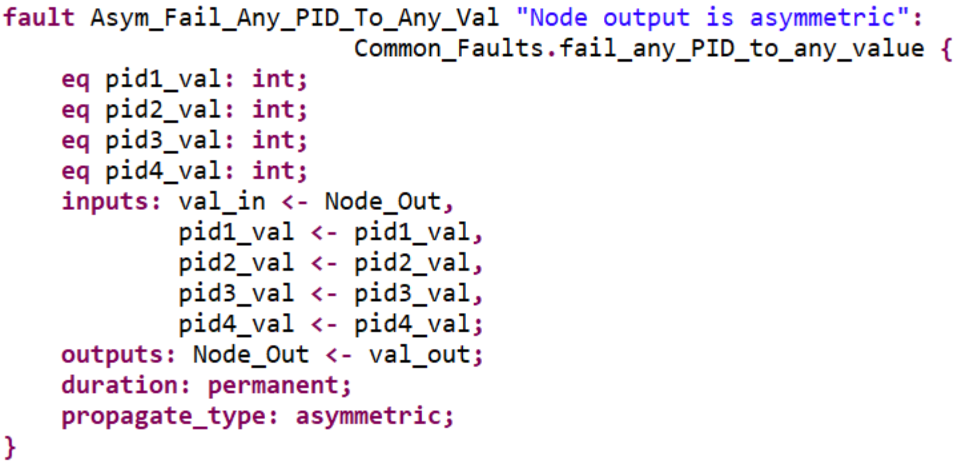
implementation in AADL and each field corresponds to a PID from a node. The AADL code fragment defining this data implementation is shown in Figure 6.10.



```
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 6.10: 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 6.11 and the fault node definition in Figure 6.12.

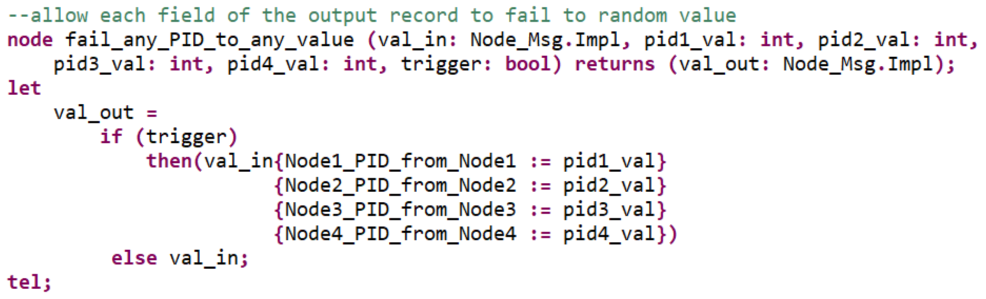


```
fault Asym_Fail_Any_PID_To_Any_Val "Node output is asymmetric":
                        Common_Faults.fail_any_PID_to_any_value {
    eq pid1_val: int;
    eq pid2_val: int;
    eq pid3_val: int;
    eq pid4_val: int;
    inputs: val_in <- Node_Out,
            pid1_val <- pid1_val,
            pid2_val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

Figure 6.11: Fault Definition on Node Outputs 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.

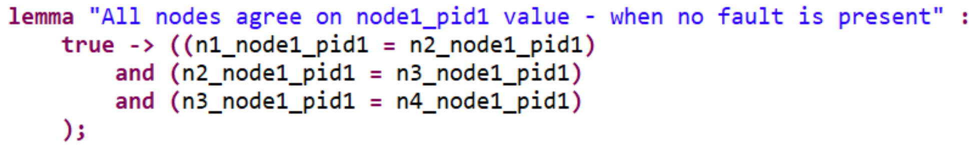


```
--allow each field of the output record to fail to random value
node fail_any_PID_to_any_value (val_in: Node_Msg.Impl, pid1_val: int, pid2_val: int,
    pid3_val: int, pid4_val: int, trigger: bool) returns (val_out: Node_Msg.Impl);
let
    val_out =
        if (trigger)
            then(val_in{Node1_PID_from_Node1 := 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 6.12: Fault Node Definition for PID Example

- 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 6.13 and Figure 6.14 show example contracts regarding Node 1 PID. There are similar contracts for each node's PID.



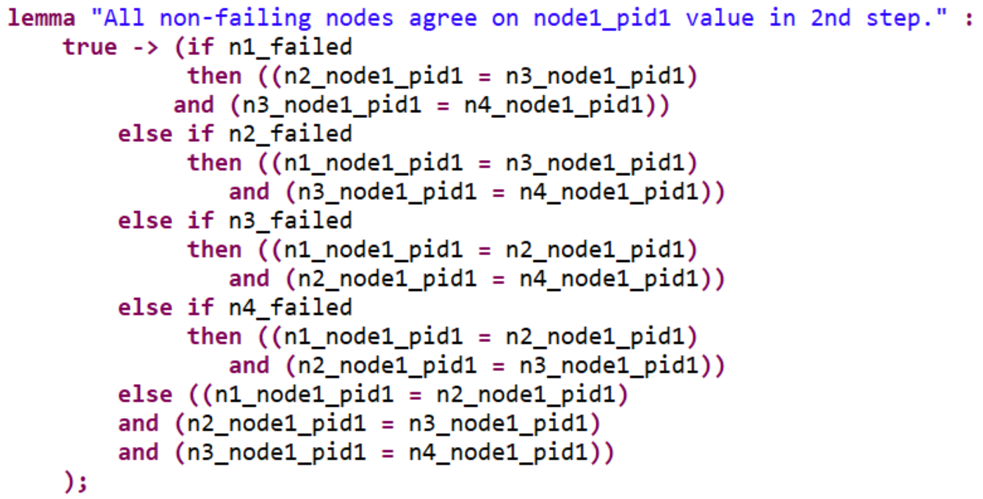
```
lemma "All nodes agree on node1_pid1 value - when no fault is present" :
    true -> ((n1_node1_pid1 = n2_node1_pid1)
        and (n2_node1_pid1 = n3_node1_pid1)
        and (n3_node1_pid1 = n4_node1_pid1)
    );
```

Figure 6.13: Agreement Protocol Contract in AGREE for No Active Faults

Referencing Fault Activation Status To fully implement the agreement protocol, it must be possible to describe whether or not a subcomponent is failed; this is performed through the use of a *fault activation statement*. The user first defines *eq* variables in AGREE that will correspond to specific faults in components. 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*). The fault activation statements for the PID example are shown in Figure 6.15.

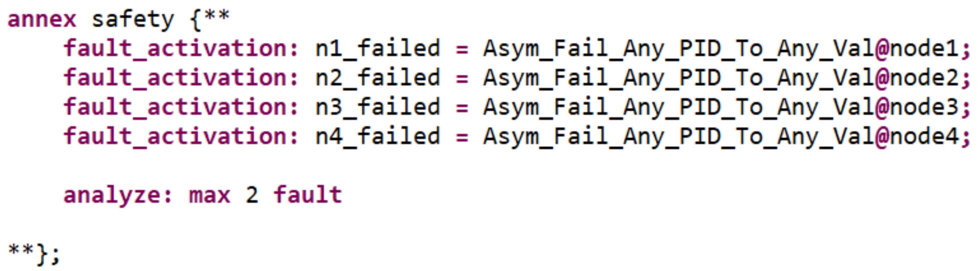
6.2.2 Nominal and Fault Model Analysis

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 6.13) fail. This is expected since this



```
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 6.14: Agreement Protocol Contract in AGREE Regarding Non-failed Nodes



```
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 6.15: Fault Activation Statement in PID Example

property is specific to the case when no faults are present in the model. The remaining 4 top level properties (Figure 6.14) 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 6.13). 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 [119].

Chapter 7

Conclusion

System safety analysis is crucial 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.

This research leverages the compositional generation of minimal inductive validity cores in order to transform them into minimal cut sets for a fault model. We also explore how the specification of contracts can change the results of this analysis and provide algorithms that provide granular refinement of the model.

This dissertation work includes the development and implementation of the Safety Annex for AADL. The Safety Annex provides a way to capture complex relationships between faults in a model and analyze their effects behaviorally through either compositional or monolithic analysis.

References

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

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

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

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

[5] J. Backes, D. Cofer, S. Miller, and M. W. Whalen. Requirements Analysis of a Quad-Redundant Flight Control System. In *NFM*, volume 9058 of *LNCS*, pages 82–96, 2015.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

[17] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. Model-based codesign of critical embedded systems. In *ACES-MB@ MoDELS*, 2009.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] 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.

[24] 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.

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

[26] 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.

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

[28] 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.

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

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

[31] 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.

[32] 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.

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

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

[35] 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.

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

[37] D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In *NASA Formal Methods Symposium*, pages 126–140. Springer, 2012.

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

[39] 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.

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

[41] 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.

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

[43] 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.

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

[45] 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.

[46] 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.

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

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

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

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

[51] 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.

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

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

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

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

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

[57] G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. Heimdahl. The effect of program and model structure on the effectiveness of mc/dc test adequacy coverage. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(3):1–34, 2016.

[58] 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.

[59] E. Ghassabani. *Inductive validity cores*. PhD thesis, 2018.

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

[61] 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.

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

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

[64] W. Hammer. Product safety management and engineering. Prentice Hall, 1972.

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

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

[67] 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.

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

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

[70] 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.

[71] 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.

[72] 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.

[73] 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.

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

[75] 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.

[76] 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.

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

[78] N. Leveson. White paper on approaches to safety engineering. http://sunnyday.mit.edu/caib/concepts.pdf, 2003.

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

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

[81] 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.

[82] 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.

[83] 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.

[84] P. Manolios, K. Siu, M. Noorman, and H. Liao. A model-based framework for analyzing the safety of system architectures. In *2019 Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–8. IEEE, 2019.

[85] C. Mattarei. *Scalable Safety and Reliability Analysis via Symbolic Model Checking: Theory and Applications*. PhD thesis, Ph. D. thesis, University of Trento, Trento, Italy, p 2, 2016.

[86] C. Miller. Applying lessons learned in accident investigations to design through a systems safety concept. Flight Safety Foundation seminar, Santa Fe, NM, 1954.

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

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

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

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

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

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

[93] 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.

[94] 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.

[95] 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.

[96] T. Prosvirnova and A. Rauzy. Automated generation of minimal cut sets from altarica 3.0 models. *International Journal of Critical Computer-Based Systems*, 6(1):50–80, 2015.

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

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

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

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

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

[102] 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.

[103] 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.

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

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

[106] 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.

[107] 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.

[108] SAE Aerospace. SAE AS5506B: Architecture Analysis & Design Language (AADL) standard document. *SAE International*, 2012.

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

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

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

[112] 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.

[113] 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.

[114] 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.

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

[116] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985.

[117] 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.

[118] 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*, Jan 2020.

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

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

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

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