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

Prof. Mats P. E. Heimdahl and Dr. Michael W. Whalen

August, 2020

Acknowledgements

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.

Risk and fault analyses help to ensure that critical systems operate in an expected way, even in the presence of component failures. As critical systems become more dependent on software components, analyses of error propagation becomes crucial. These analyses should be scalable and sound in order to provide sufficient guarantee that the system is safe. This dissertation presents a behavioral approach to modeling faults and their propagations in an architecture analysis language, and uses a compositional reasoning framework to derive artifacts that encode pertinent system safety information. We also presents a compositional approach to the generation of fault forests and minimal cut sets. The analysis is performed per layer of the architecture and we compose the results. Complete formalizations are given. We implement the composition of fault forests by leveraging minimal inductive validity cores produced by an infinite state model checker. This research provides a sound and scalable alternative to a monolithic framework. This enables safety analysts to understand the behavior of the system in the presence of faults while collecting artifacts required for certification.

Contents

List of Tables

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 for example 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 began shortly after World War II. The US Air Force was having numerous aircraft accidents; over 7,700 aircraft were lost between the years of 1952 and 1966 and over 8,000 people were killed [70]. Their approach to aircraft system safety was to analyze the accident and "fix" the problem for the next flight. At the time, many of the accidents were blamed on pilots, but a number of flight engineers did not believe the causes were so simple. They posited that safety must be designed and built into the aircraft [88]. With the growth of nuclear capabilities, the defense industry complex, and the overall increase of computerization, the need to abandon a "fly-fix-fly" approach to safety was imminent [70, 88, 100]. 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, 131, 132].

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 [131, 132]. 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 [29]. An important goal of the safety assessment process is to show what kinds of failures may occur during normal use of the system. These analyses, both qualitative and quantitative, can provide information on how the system is safe (or unsafe) for use [126].

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 engineering* (MBSE) and *model-based safety assessment* (MBSA) caught the attention of researchers in the safety critical system domains [19, 68, 76, 81, 94]. In model-based engineering, 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.

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 [**?**, 16, 22, 68]. Thus there is a system model used by the system engineers and a fault model used by safety analysts. 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 of this approach is that any changes made in system development are not automatically reflected in the safety analysis process; those changes must be communicated to safety analysts and incorporated into the separate fault model. 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 [93]. Various research has attempted to address this gap by providing tools that operate over a model and provide some form of propagation analysis, (e.g., [16, 53, 79]). Other times this propagation is done explicitly (the analyst manually defines where the fault will propagate through the system) [95], but as the size and complexity of industrial sized systems grow, explicit propagation can become unwieldy [143]. To address this problem, *behavioral* propagation has been introduced [16, 141]. Behavioral propagation automates the process of propagating 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 should be possible.

While using model based safety assessment, *verification* of the model and its requirements can provide additional and crucial information about the system model. Verification, in this context, 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. Without it, verification of the model and its requirements cannot be adequately performed.

Commonly used artifacts in the safety assessment process are *minimal cut sets*, or the minimal sets of faults that can lead to a violation of a system safety property and their associated fault trees. The automatic generation of these artifacts have been studied in depth, but have often lacked in terms of scalability [13,82,99,102,145]. Some research groups have introduced automating aspects of the safety assessment process and have developed tools to support this [19, 23, 79]; nevertheless, there are gaps in current capabilities we address in this dissertation.

1.1 Objectives 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 **contributions of this dissertation**, which are logical steps towards the goal, started with the definition of 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 were incorporated into these analyses in order to fully understand the safety of the system. Domain specific case studies demonstrate the feasibility 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 [6,52]. 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 grammar with a safety annex extension and kept a few specific fault modeling needs in mind [141, 143]. 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 that performs 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 for either 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. The hypothesis statement 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 variables in the model. If any combination of faults is within allowable 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 derive sets of fault trees and associated minimal cut sets. The minimal sets of faults that when active can 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 [29, 49, 117, 118, 127]. 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. [15,24,25,27,127,133]. We have pushed forward on this front and found a way to generate these sets in a *compositional* fashion by composing sets of fault trees. Compositional verification performs the proof in a per-architectural-layer approach; this divides a very large proof over the entire system into smaller proofs over each layer of the system. These smaller proofs are then composed together to provide the system level proof. To our knowledge, composition of fault forests has not been previously performed. This research formalizes the composition of fault forests and implements the associated algorithm in the safety annex.

Explored how the formal specification of requirements can change analysis results. Splitting a complex requirement into its constituent conjuncts introduces the possibility of changing certain analysis results. Because the compositional generation of minimal cut sets relies on the requirements for each component, it is natural to question how the structure of the requirements may affect analysis results. We explored this idea by automatically decomposing requirements into smaller subexpressions and rewriting them into semantically equivalent but syntactically (structurally) different forms, and

compared analysis results between the original contracts and the rewritten contracts and discussed the findings. We found that as the requirements became more *granular*, i.e., split into more conjuncts [63], the inductive validity cores enumerated show which subformulae of an equation were necessary for proof. The specificity of requirements we refer as *granularity* and this idea ties into a broader discussion of the ideas underlying requirement engineering, behavioral modeling, minimal cut sets, and system development.

Demonstrated the objectives of this proposal by use of case studies. A large scale 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. We perform various timing experiments to provide insight into the scalability of the approach. Furthermore, numerous subsystem examples are given throughout the dissertation to illustrate specific capabilities and solutions. These examples demonstrate how the safety analysis process described in here can be applied in the domain of aerospace and other critical system domains.

In summary, this dissertation provides a modeling notation that supports a close relationship between the system development and safety assessment processes, and it defines analysis procedures that verify that the system model meets requirements in the presence of faults. This research also provides a formalism that defines compositional derivation of fault forests, and it explores the granularity of contracts and how that affects analysis results. Finally, the demonstration of these contributions are shown through use of case studies.

1.2 Structure of this Document

This dissertation is organized into 8 chapters. Chapter 2 discusses the preliminaries, related work, and an overview of formal verification. Chapter 3 provides a detailed look at fault modeling in complex critical systems, and the safety annex and its implementation. Chapter 4 describes the compositional generation of minimal cut sets and

provides the formalisms and algorithms of such generation; this is followed by a chapter on case studies. Chapter 6 provides the initial exploration of how a particular form of contract definition can change the results of the analysis. We include Chapter 7 as a discussion of this research and how it could be extended. Lastly, the conclusion in Chapter 8 summarizes the dissertation.

Chapter 2

Preliminaries and Related Work

Safety analysis plays an important role in the development of critical systems; it is through the safety assessment process that safety engineers and certification authorities are convinced that the system under consideration satisfies its requirements. In this chapter, we describe the high level safety assessment process that is used in the avionics industry, and then we introduce model-based safety assessment as an approach to the safety process. Because the main contributions of this work rely heavily on formal methods of analysis, the background and key ideas of formal methods are presented. This is followed by a brief review of closely related work.

2.1 The Safety Assessment Process

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 [1]. The development of such complex systems can benefit from a process with clearly defined design and implementation phases, and can further be subdivided into several sub-processes and phases. Analyses can be performed 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)

documents are commonly referenced. The Society of Automotive Engineers (SAE) is an association of engineers and professionals devoted to the standards that guide the development of transportation systems [131, 132].

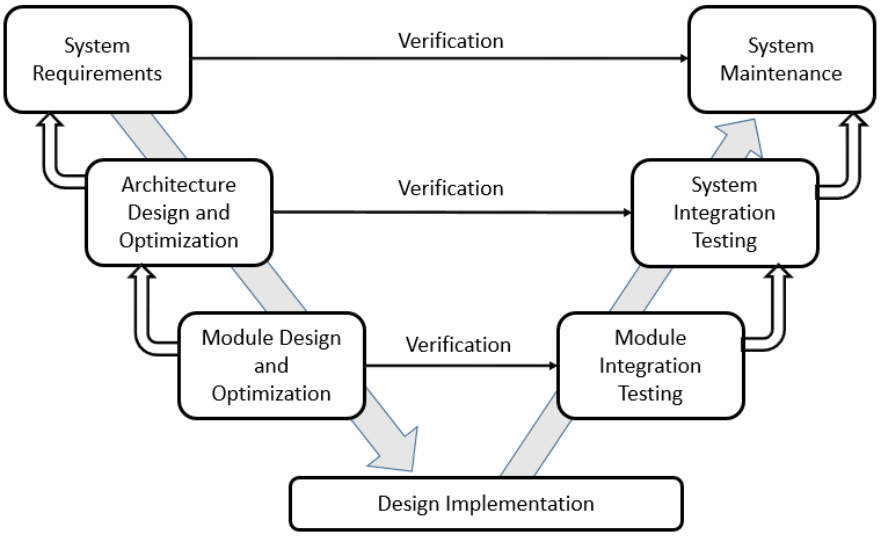


Figure 2.1: The V Model in System Development

The V model in software engineering, shown in Figure 2.1, 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 high-level design; the requirements of the system drive the design. The right side of the V describes the low-level implementation and testing of each module independently and as a whole.

ARP4754A, the Guidelines for Development of Civil Aircraft and Systems [132], 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 safety 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 [132]
.

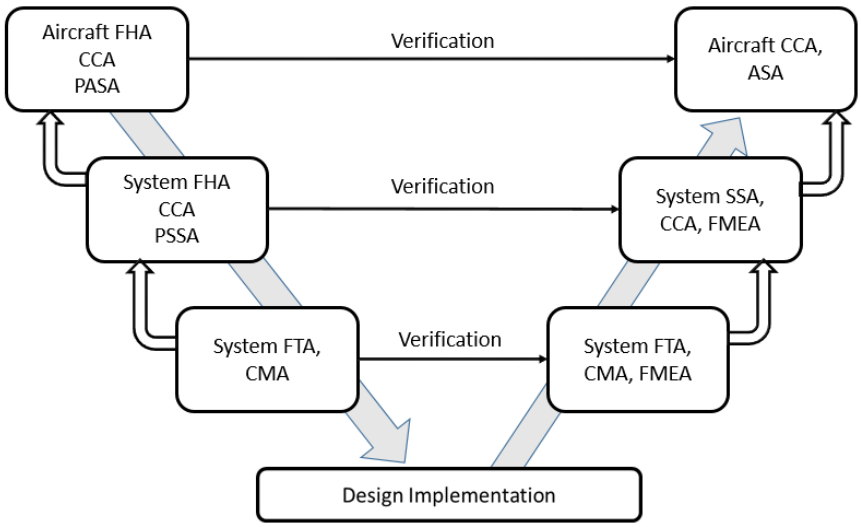


Figure 2.2: The V Model in Safety Assessment

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 process includes safety requirements identification (left side of V) and verification (right side of V) supporting the aircraft development activities. The aircraft-level Functional Hazard Analysis (FHA) is conducted at the beginning of the development cycle and is followed by system level Functional Hazard Analysis for individual subsystems. The FHA is followed by the Preliminary System Safety Assessment (PSSA), which derives safety requirements for the subsystems, primarily using Fault Tree Analysis (see Section 2.1.1 for more information on fault trees). The PSSA process iterates throughout the design evolution as potential safety problems are identified and addressed through design changes. Once design and implementation are completed, the System Safety Assessment (SSA) process verifies whether the safety requirements are met in the implemented design.

Both the preliminary safety assessment and the system safety assessment relies on safety related artifacts that describe the behavior of the system in the presence of faults.

These artifacts are used throughout the entire process of development and are ubiquitous in the field of safety analysis. The most important of these for this research are fault trees and minimal cut sets.

2.1.1 Fault Trees and Minimal Cut Sets

A *fault tree* is a directed acyclic graph whose leaves model component failures and whose gates model failure propagation. The system failure under examination is the root of the tree and is called the *top level event*. The *basic events* are the events that can occur in the system which lead to the top level event and in the graphical model, these correspond to the leaves. 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 2.3, the AND gate has three inputs and one output. The leaves of the tree represent the basic events of the system and in the case of this fault tree, these three events are also a *minimum cut set* for this top level event. The minimal cut set is the minimal set of basic events that must occur together in order to cause the top level event to occur. Finding these sets is important to fault tree analysis and has been an active area of interest in the research community since fault trees were first described in Bell Labs in 1961 [49,128].

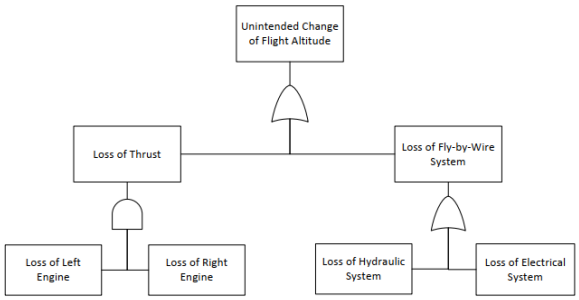


Figure 2.3: A Simple Fault Tree

Figure 2.3 shows a simple example of a fault tree. In this example, the top level event corresponds to an aircraft having an unintended change of altitude. In order for this event to occur, there must be either a loss of thrust or the loss of a Fly-by-Wire system. This is seen through the use of the OR gate below the top level event. The

malfunction of both the left and right engines will cause the loss of thrust to occur and the Fly-by-Wire system can be lost if either the hydraulic system or the electrical system were to malfunction. The MCSs for this example are {Loss of Left Engine, Loss of Right Engine}, {Loss of Hydraulic System}, and {Loss of Electrical System}.

History of Fault Trees 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 [128,145]. Fault tree analysis requires one to explore the faults of the system and their effects on system behavior to determine minimal fault configurations that may violate requirements. From the beginning of fault tree analysis in the 1960s, algorithms worked directly with the fault tree structure to produce minimal cut sets [58, 135]. In essence, these algorithms represented each AND/OR gate as a boolean expression and then performed simplification to relate the basic events to the top level event without any gates [128]. In 1993, Rauzy et al. developed a new approach that converted the fault tree structure into a binary decision diagram (BDD) [118]. This was a natural way to reduce the Boolean formula into something far more computationally efficient and reducible to even simpler forms. BDDs are still commonly used to perform quantitative and qualitative fault tree analysis [8,32,62,77,119,120,123,137]. Other forms of fault tree analysis include Monte Carlo methods [146], Markov chains [17], and zero suppressed BDDs [102].

2.1.2 Model Based Development

System safety analysis techniqes are well established and used extensively in the design of safety critical systems. These safety analysis techniques are often performed manually based on informal design models and various other documents [81,134]. Fault trees are one of the most common artifacts used by safety engineers, but different engineers may produce substantially different fault trees for the same system. It becomes clear that the analyses are highly subjective and dependent on the skill of the practitioner. Since the analyses are based on informal system documentation, researchers and practitioners have proposed a consolidation of the information into a central entity and use this to perform safety analysis [19,29,78–80,95].

One way to achieve consolidation of information spread across various informal

documents is through *Model-based Development* (MBD) [134]. In MBD, the development is centered around a formal specification or model of the system. This model can be analyzed for completeness and consistency [73], model checking [38, 67, 101], theorem proving [122], test case generation [5, 121], etc. One can also automate aspects of the implementation from the formal specification. There are several modeling and verification notations that provide these capabilities.

Model-based Development can also refer to a process that considers a non-formal model, such as SysML [57] or UML [55], as the central development artifact. In this dissertation, we consider a formal model of the system in a language with well-defined semantics as the central artifact of the MBD process.

2.1.3 Model Based Safety Assessment

The process of creating system models suitable for use in safety assessment closely parallels the model-based development process. A *Model-based Safety Analysis* (MBSA) approach has been proposed in related literature [20, 78, 81] that extends the MBD process.

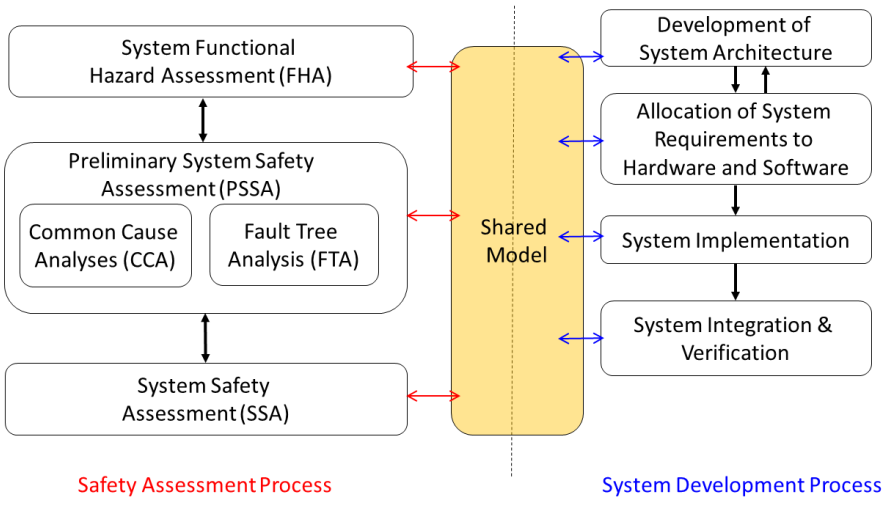


Figure 2.4: Use of the Shared System/Safety Model in the ARP4754A Safety Assessment Process

Model-based development focuses on electronic components of an embedded system. To perform safety analysis at a system level, in addition to electronic or digital components, one must also consider the environment and mechanical components. Both mechanical and electrical/digital components are necessary to model the system-level faults that are of interest in safety analysis. By combining the models containing digital components (i.e., software and hardware architectures) with models of the mechanical components (i.e., pumps, valves), we create a *nominal model* of the system. The *nominal system behavior* is a model of the system as it behaves in the absence of faults.

The nominal model can then be augmented with fault behaviors for the various electrical and mechanical components to create the *fault model* of the system. A great advantage to this approach is that the system and safety engineers work off of a *shared model* as shown in Figure 2.4 which leads to a tighter integration between the system and safety engineering processes. Figure 2.4 presents a proposed use of a single unified model to support both system design and safety analysis. It describes both system development and safety-relevant information that are kept distinguishable and yet are able to interact with each other. The shared model maintains a living model that captures the current state of the system design as it moves through the development life cycle, allowing all participants of the process to be able to communicate and review the system design. 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.

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 as it does in Figure 2.4; our proposed contribution to this process is shown in Figure 2.5 and described as follows.

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.

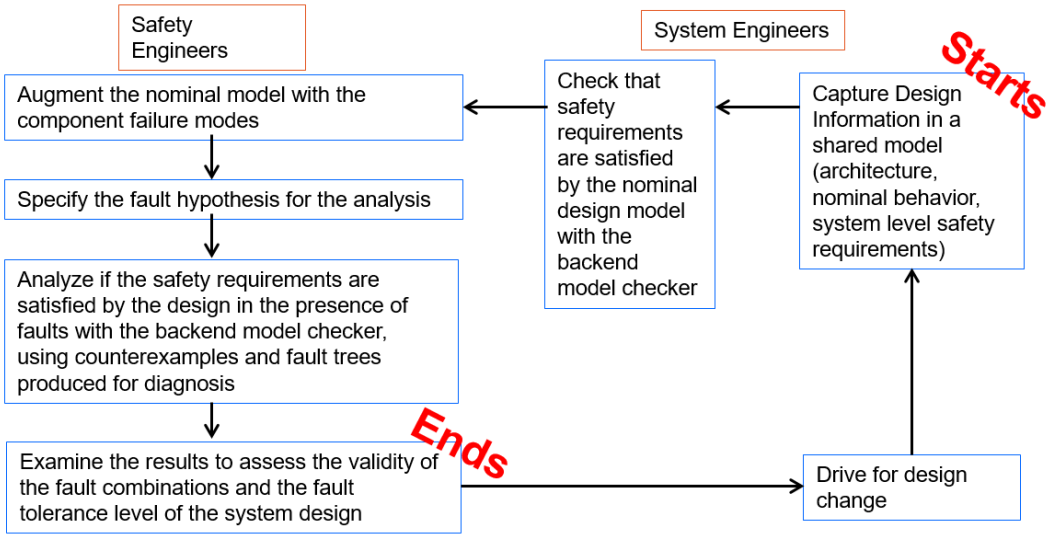


Figure 2.5: Proposed Steps of the Safety Assessment Process

2. System engineers use formal verification 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 formal verification 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 examples of system states during failure 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 2.5 shows these steps within the context of the start and end of a project. A model that supports both system design and safety analysis must describe both the system design information (e.g., system architecture and functional behavior) and safety-relevant information (e.g., failure modes, failures rates). It must do this in a way that keeps the two types of information distiguishable, yet allows them to interact with each other. The shared model in Figure 2.5 is expected to be created and maintained in sync with the software and hardware design and implementation, and guided by the hazard and probability information from the preliminary system safety assessment. This safety information is then used to drive for design changes if necessary and continues in an iterative fashion until the system safety property is satisfied with the desired fault tolerance.

2.2 Formal Methods in 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 undesirable events are sufficiently unlikely to occur [50]. Utilizing the recent advancements in automated formal verification within the validation process has become essential to the certification of critical systems [2, 87, 111] and the world of safety analysis began to see its powerful benefits [29, 30, 41, 74, 92]. 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 practical issues. Formal validation and verification is a proof-based methodology used to assess the correctness of requirements, system design, and implementation. This section provides a background of the formal method techniques that are commonly used in the system development and safety assessment processes.

2.2.1 Overview

Given that this research is focused on model-based system development and safety assessment, we focus our attention onto *model checking* as a method of formal analysis. Model checking is an automatic technique for verifying that concurrent system models meet their specified requirements [38]. Applying model checking to a system design consists of a few main tasks: *modeling*, *formal specification*, and *formal verification*. The digital and mechanical components of a system can be described in abstract form (modeling), and the requirements of the system and of each component can be specified in formal logic (formal specification). The formal verification of such models take both the architecture and the requirement specification into account when analyzing the behavior and interactions of the components comprising the system. In the sections that follow, we will outline these three major components of model checking and describe the aspects important in this research.

2.2.2 Modeling

When modeling a system, the digital and mechanical components are described in abstract form; furthermore, the requirements of the system and of each component can be specified in formal logic. The verification of such models take both the architecture and the requirement specification into account when analyzing the behavior and interactions of the components comprising the system. Throughout the past few decades, numerous modeling languages and tools have been introduced, for example Simulink from MathWorks [97], SCADE from Esterel Technologies [3], and research base languages such as Lustre [69]. Other common modeling languages include SysML [57] and AADL [52].

Often, engineers who design safety critical systems model their systems as networks of operators transforming flows of data. At a higher level, this can be represented by block diagrams that group these networks into reusable components. *Dataflow* languages allow these models to directly represent the digital control system. Dataflow programming languages have several merits, one of which is that the program is a completely functional model of the system. This feature makes the model well suited to formal verification and program transformation; it also facilitates reuse, because the

module will behave the same way in any context into which it is embedded [78]. For this dissertation, we focus our attention on Lustre [69], a synchronous[1] dataflow programming language used in the formal verification portion of this research. Lustre is described in more detail in Section 2.4.3.

2.2.3 Formal Specification

Before we can verify the correctness of a system, we must first specify the properties that the system should have [38]. The formal specification process translates the informal system requirements into a mathematical logic to determine if the system design is correct [74]. This process guarantees an unambiguous description of the requirements, which is not possible when using an informal natural language. The formal definition of system requirements includes the system design and its expected behavior as well as the assumptions on the environment in which the system is expected to operate. 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 [86]. A commonly used method of specification is *temporal logic*. Temporal logics are useful for specifying complex system requirementss, because they can describe the ordering of events in time without introducing time explicitly.

Linear Temporal Logic

Temporal logic can be used to express properties of reactive systems [29]. 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."

[1] A synchronous language breaks real time into a sequence of instants in which the outputs of the model are computed.

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 [138]. Formal definitions and more information on LTL and CTL can be found in a number of research works [29, 38].

2.2.4 Formal Verification

Once we have specified the important properties (formal specification), then a formal model for the system is created; this model captures the properties that must be considered to establish correctness [38]; this process is referred in this dissertation as *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 [54].

Model checking was introduced in the early 1980s and consists of exploring the states and transitions of a model [37, 115]. By representing the system abstractly, a possibly infinite state space is reduced to a finite model. [46]. 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 formal logic, then explores the state space of the system to determine if the model violates the properties [38, 56]. 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 [46]. Nevertheless, the biggest limiting factor of model checking is scalability and much of the recent research in this area attempts to address this problem [38].

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 [46, 54]. Deductive methods of verification include automated theorem provers (e.g., Coq [44], Isabelle [107]) and satisfiability modulo theories (e.g., SMTInterpol [34], Z3 [43], Yices [47]).

2.3 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 [29, 30, 41, 74, 92]. 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 of related work and how formal methods has been applied in the domain of safety assessment research.

2.3.1 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 formal methods in the '90s and are still contributing today (e.g., [33, 35, 124, 136].

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 [46].

In the early 2000s, model based safety assessment began to make an appearance in literature [29, 79–81]. The researchers began applied model checking in model based system development to safety analysis. In this approach, a safety analysis system model 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 system model.

The contents and structure of the safety analysis system model differ significantly across different conceptions of model-based safety analysis. We can draw distinctions between approaches along several different axes. The first is whether they propagate errors explicitly through user-defined propagations, which we call *explicit propagation*, or through behavioral requirements and interactions in the model itself, which we call *implicit propagation*. The next is whether models and notations are *purpose-built* for safety analysis vs. those that extend *existing system models*.

For implicit propagation 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*.

Figure 2.6 highlights the differences between these approaches in closely related work. The left column of the figure shows the tool names and across the top row are the various ways of structuring and analyzing the safety analysis system model. The tools and their approaches are described in the following subsections.

These tools attempt to address various needs in the safety community and do so in distinct ways, but we wish to combine many of these efforts under one existing system

	Explicit Prop	Implicit Prop : Directional	Implicit Prop: Bidirectional	Mono	Comp	Purpose Built	Existing System Model
EMV2	✓						AADL
Compass	✓				✓		SLIM
SmartIFlow			✓	✓		✓	
SAML		✓		✓		✓	
AltaRica			✓	✓		✓	
HipHops	✓						East-ADL

Figure 2.6: Approaches of Related Work

model. We make it possible to extend the AADL system model with a fault model. Both nominal and fault analysis should be able to be performed monolithically or compositionally, and the fault model should allow for either explicit or implicit propagation. We attempt to address multiple needs within a single framework, unlike many of the related tools.

To summarize, we created a safety modeling framework that allows for (1) both *implicit* and *explicit* directional propagation, (2) both *monolithic* and *compositional* verification, and (3) that extends an existing system model.

The following 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 and closely related 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 [136]. The key idea was that this transition system (more specifically *constraint automata*) could be compiled into Boolean formulae and transformed into a binary decision diagram [110]. 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 [14]. For this dialect, the safety assessment, fault tree generation, and functional verification can be performed with the aid of NuSMV model checking [22].

The most recent language update (AltaRica 3.0) uses non-causal semantics [112–114]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [13]. AltaRica 3.0 has support for simulation and Markov model generation through the OpenAltaRica (www.openaltarica.fr) tool suite; it uses *implicit error propagation*, and it is a *purpose-built*, *monolithic* safety analysis language.

FSAP, xSAP, and COMPASS

The Formal Safety Analysis Platform (FSAP) was introduced in 2003 [28] and supported failure mode definitions, safety requirements in temporal logic formulae, automated fault tree construction, and counterexample traces. The platform used NuSMV, a binary decision diagram (BDD)-based model checker [36]. 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 [16]. 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 [19,98]. 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 [19].

COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [21] allows for *explicit propagation*, and is a *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 [20, 26]. 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 [36, 108], MRMC (Markov Reward Model Checker) [83, 103], and RAT (Requirements Analysis Tool) [116]. The safety analysis tool xSAP [16] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [18].

SmartIFlow

SmartIFlow [75, 76] uses *implicit propagation* and is a *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” [76].

SAML

The Safety Analysis and Modeling Language (SAML) [68] uses *implicit propagation*, and is a *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 [36], PRISM (Probabilistic Symbolic Model Checker) [85], or the MRMC model checker [83]. 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 [68].

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 [96, 130]. 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) [53]. EMV2 uses *explicit propagation* and is based on an *existing system model* 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 [51].

2.4 Modeling and Formal Methods: Important Concepts

The contributions of this research require further information regarding specific concepts of formal methods. Our approach allows safety analysts to extend an existing system model with fault information. The nominal model can then be analyzed using formal methods of verification to show that in the absence of faults the system meets its specified requirements. The extended fault model can also be formally analyzed to show how active faults in the system may violate the specified properties.

The existing system model is written in the Architecture Analysis and Design Language (AADL) [6]. This model is extended with behavioral contracts that formalize

the requirements into temporal logic using the Assume-Guarantee Reasoning Environment (AGREE) [39]. This constitutes the nominal system model. Verification through AGREE consists of a translation of the AADL model along with the AGREE contracts into a Lustre [69] model. JKind [59], an infinite state model checker, is used to perform the formal verification of the Lustre model. Verification of the program is based on *k-induction* (see Section 2.4.6) and property directed reachability using a back-end SMT solver such as Z3 [43] or SMTInterpol [34].

Given this nominal model organization, we extend this to allow for reasoning about faults. This is detailed in Chapter 3. The remainder of this section includes important concepts and definitions regarding the formalization of the nominal system model.

2.4.1 Architecture Analysis and Design Language

The Architectural 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 [6, 52]. 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.

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

2.4.2 Compositional Analysis in the Assume-Guarantee Reasoning Environment

The *compositional* analysis of systems was introduced in order to address the scalability of model checking large software systems [40,72,109]. *Monolithic* analysis flattens the hierarchical system model and use all model elements from all layers in order to find proof of a safety property. Compositional analysis, on the other hand, is performed per the architecture hierarchy such that analysis at a higher level is based on the components at the next lower level and conducted layer by layer; the components of a system are organized hierarchically and each layer of the architecture is viewed a system. The idea is to partition the formal analysis of a system architecture into verification tasks that correspond into the decomposition of the architecture.

A component contract in 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 assume-guarantee compositional reasoning uses the past-time LTL operators G (globally), U (until), H (historically), and Z (in the previous instant).

A component contract is an assume-guarantee pair (A, P) for propositions A, P. Assume-guarantee reasoning attempts to prove that if the assumptions have held in all previous instances up to the current instance, then the guarantee holds at the current time [39]; formally, this can be written as $G(H(A) \implies P)$.

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.

Monolithic verification of an assume-guarantee reasoning environment attempts to prove the statement $G(H(A_S) \implies P_S)$ directly from the system and subcomponent assumptions. Compositional analysis, on the other hand, consists of $n + 1$ verification tasks per layer for each n components. Component verification conditions establish that the assumptions of each component are implied by the system assumptions and the properties of sibling components. The $n^{th} + 1$ task is the system level verification

condition which shows that system guarantees follow from system assumptions and the properties of each subcomponent. This proof is performed one layer at a time starting from the top level of the system and composed accordingly.

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 [40]. The *Assume-Guarantee Reasoning Environment* (AGREE) [39] provides a way to perform compositional verification on models that are defined using the Architecture Analysis and Design Language (AADL) [130]. More details on AGREE are given as needed throughout this document.

2.4.3 Lustre

The AADL/AGREE model is translated into Lustre [69], a synchronous dataflow programming language that is suitable for the design of a critical system, specification of its critical properties, and verification of those properties. A Lustre variable or expression is considered to represent the sequence of values it takes during the whole execution of the program, and Lustre operators are considered to operate globally over these sequences [69]. In other words, real time is abstracted into execution steps and the variables and expressions in the program take their values at each time step.

```
node sensor(env_temp : int)
            returns (temp_high,temp_reading : bool);
var is_high : bool;
let
  is_high = (env_temp > 8);
  temp_high = is_high;
  temp_reading = env_temp;
tel;
```

Figure 2.7: Sensor Node Defined in Lustre

A simple example of a sensor node is shown in Figure 2.7. There is a single input, `env_temp`, and two outputs: `temp_high` and `temp_reading`. A local variable (`is_high`) is defined and assignments are made in the `let ... tel;` body of the node. The input comes in and the sensor outputs a high indication and a reading of the environmental temperature.

2.4.4 JKind

JKind is an open-source industrial infinite-state inductive model checker for safety properties [59]. Models and properties in JKind are specified in Lustre [69], a synchronous dataflow language, using the theories of linear real and integer arithmetic. JKind uses SMT-solvers to prove and falsify multiple properties in parallel. To understand how this analysis proceeds, some formal definitions and descriptions must be provided.

2.4.5 State Machines and Transition Systems

A state machine (or 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 Lustre model is viewed as a state machine with transitions between these states defined through the properties of the nodes.

Transition systems are directed graphs with nodes representing reachable states and edges representing transitions between them. In this research we consider *safety properties* over infinite-state machines. The states are vectors of variables that define the values of state variables. We assume there are a set of legal *initial states* and the safety property is specified as a formula over state variables. A *reachable state space* means that all states are reachable from the initial state.

Given a state space U, a transition system (I, T) consists of an initial state predicate $I : U \rightarrow \mathit{bool}$ and a transition step predicate $T : U \times U \rightarrow \mathit{bool}$. We define the notion of reachability for (I, T) as the smallest predicate $R : U \rightarrow \mathit{bool}$ which satisfies the following formulas:

$$\forall u.\ I(u) \Rightarrow R(u)$$
$$\forall u, u'.\ R(u) \wedge T(u, u') \Rightarrow R(u')$$

A safety property $P : U \rightarrow \mathit{bool}$ is a state predicate. A safety property P holds on a transition system (I, T) if it holds on all reachable states, i.e., $\forall u.\ R(u) \Rightarrow P(u)$, written as $R \Rightarrow P$ for short. When this is the case, we write $(I, T) \vdash P$.

2.4.6 Induction

For an arbitrary transition system (I, T), computing reachability can be very expensive or even impossible. Thus, we need a more effective way of checking if a safety property P is satisfied by the system. The key idea is to over-approximate reachability. If we can find an over-approximation that implies the property, then the property must hold. Otherwise, the approximation needs to be refined.

A good first approximation for reachability is the property itself. That is, we can check if the following formulas hold:

$$\forall s.\ I(s) \Rightarrow P(s) \tag{2.1}$$

$$\forall s, s'.\ P(s) \wedge T(s, s') \Rightarrow P(s') \tag{2.2}$$

If both formulas hold then P is *inductive* and holds over the system. If (4.1) fails to hold, then P is violated by an initial state of the system. If (4.2) fails to hold, then P is too much of an over-approximation and needs to be refined.

The JKind model checker [59] used in this research uses k-*induction* which unrolls the property over k steps of the transition system. For example, 1-induction consists of formulas (4.1) and (4.2) above, whereas 2-induction consists of the following formulas:

$$\forall s.\ I(s) \Rightarrow P(s)$$

$$\forall s, s'.\ I(s) \wedge T(s, s') \Rightarrow P(s')$$

$$\forall s, s', s''.\ P(s) \wedge T(s, s') \wedge P(s') \wedge T(s', s'') \Rightarrow P(s'')$$

That is, there are two base step checks and one inductive step check. In general, for an arbitrary k, k-induction consists of k base step checks and one inductive step check as shown in Figure 4.4 (the universal quantifiers on s_i have been elided for space). We say that a property is k-inductive if it satisfies the k-induction constraints for the given value of k. The hope is that the additional formulas in the antecedent of the inductive step make it provable.

In practice, inductive model checkers often use a combination of the above techniques. Thus, a typical conclusion is of the form "P with lemmas $L_1, \ldots, L_n$ is k-inductive".

$$I(s_0) \Rightarrow P(s_0)$$
$$\vdots$$
$$I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-2}, s_{k-1}) \Rightarrow P(s_{k-1})$$
$$P(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge P(s_{k-1}) \wedge T(s_{k-1}, s_k) \Rightarrow P(s_k)$$

Figure 2.8: k-induction formulas: k base cases and one inductive step

2.4.7 The SAT Problem and SMT Solvers

A k-induction model checker utilizes parallel SMT-solving engines at each induction step to glean information about the proof of a safety property. The transition formula is translated into clauses such that satisfiability is preserved [48]. 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, or true. On the other hand, the proposition $a \wedge \neg a$ is unsatisfiable; no such assignment can be found to satisfy both a and $\neg a$.

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 an SMT formula.

Thus, in a k-induction proof, the satisfiability at each step can be determined by an SMT-checker:

$$P(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge P(s_{k-1}) \wedge T(s_{k-1}, s_k) \wedge \neg P(s_k)$$

By negating the property and receiving an *unsatisfiable* result, this tells us that we can prove the property given the conjuncts defined in that step.

UNSAT Cores and Minimal Unsatisfiable Subsets

When analyzing a model, there are certain questions that may be asked about the model requirements. If a model is unsatisfiable with respect to some system level property, it is of benefit to know *why* it is not satisfiable.

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 [90]. 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. Definitions 2-4 are taken from research by Liffiton et. al. [90].

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 [89] and most of them focus on finding a single such subset [9–11]. More recently, algorithms have been introduced that can find all such minimal unsatisfiable subsets [12, 64, 66].

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 of a safety property. 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 [64]. 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`) [12, 66].

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. More details about IVCs can be found in Chapter 5.

Chapter 3

Fault Modeling and the Safety Annex

Early on in Section 2.1.3, 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.

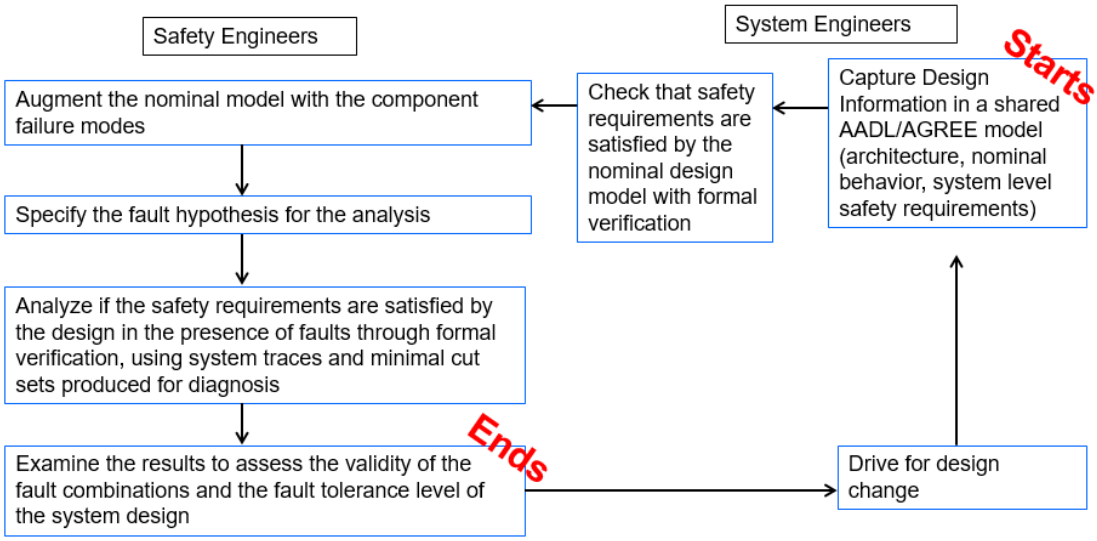


Figure 3.1: Proposed Steps of the Safety Assessment Process

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 2.3, it can be seen that this research attempts to fill certain gaps in the field of safety analysis.

Shared model with 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 this chapter, the fault modeling process using the safety annex for the Architecture Analysis and Design Language (AADL) [6] is described. The safety annex was developed with these two broad ideas in mind: (1) how this annex can support the proposed safety assessment process, and (2) what are the gaps in current related safety analysis tools that can be closed with this research.

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 [132]. 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) [53], 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 Implementation of the Safety Annex

Important features were considered before the implementation of the safety annex; these we addressed one by one and will outline below.

Shared model As described in Section 2.4, an AADL [6] model describes a system in terms of a hierarchy of components and their interconnections, where each component can either represent a logical entity (e.g., application software functions, data) or a physical entity (e.g., buses, processors). AADL is used to specify and analyze real-time embedded systems. It includes specifications specific to hardware, software, and system component abstractions. The language definition is sufficiently rigorous to support formal analysis tools that allow for early phase error/fault detection [52].

```
system TempSensor
    features
        env_temp : in data port Integer;
        high_temp_indicator : out data port Boolean;
        temp_reading : out data port Integer;

end TempSensor;
```

Figure 3.2: An AADL Component Type Definition

Central to an AADL model are component *type* and *implementation* declarations. Figure 3.2 shows an example of a simple sensor component type defined in AADL. The component has an environmental temperature as input and two outputs: a high temperature indication and a temperature reading. In the type declaration, you define the category (`system` in this example) and features such as inputs and outputs; the implementation contains definitions of the internal structure of the component, e.g., internal constituents and their interactions. Figure 3.2 shows a component *type*.

The implementation containing the sensor component type is shown in Figure 3.3. The system contains a type of its own (top of figure: `system TopLevel`) which holds any environmental inputs or subcomponent outputs. The implementation defines the subcomponents of the system and their connections (bottom half of figure).

Since AADL supports model-based system development and the language definition is sufficiently rigorous to support formal analysis tools that allow for early phase

```
system TopLevel
    features
        env_temp : in data port Integer;
        env_pressure : in data port Integer;

        temp_sensor_high : out data port Boolean;
        pressure_sensor_high : out data port Boolean;

end TopLevel;

system implementation TopLevel.impl
    subcomponents
        pressure_sensor: system Sensors::PressureSensor;
        temp_sensor: system Sensors::TempSensor;

    connections
        temp_out :
            port env_temp -> temp_sensor.env_temp;
        pressure_out :
            port env_pressure -> pressure_sensor.env_pressure;

        temp_indicator :
            port temp_sensor.high_temp_indicator -> temp_sensor_high;
        pressure_indicator :
            port pressure_sensor.high_pressure_indicator -> pressure_sensor_high;

end TopLevel.impl;
```

Figure 3.3: An AADL Component Implementation Definition

error/fault detection [52], this language was chosen for this research.

Behavioral analysis As described in Section 2.1.3, *nominal model analysis* is a part of the MBSA process. The nominal model consists of the system model architectural design as well as behavioral contracts for each component and requirement specifications. The verification at the nominal level consists of showing that the model satisfies the specified requirements in the absence of faults.

The Assume-Guarantee Reasoning Environment (AGREE) [39] is a language annex for AADL that provides a mechanism for the specification of component requirements in formal logic and utilizes a model checker to provide proofs regarding these specifications as described in Section 2.4.

An example of an AGREE contract is shown in Figure 3.4 and is placed in the context of the AADL temperature sensor component shown in Figure 3.2. An AGREE contract consists of *assumptions* on the inputs of AADL components that constrain what

```
system TempSensor
    features
        env_temp : in data port Integer;
        high_temp_indicator : out data port Boolean;
        temp_reading : out data port Integer;

    annex agree {**
        assume "Temp bounded":
            ((env_temp > 0) and (env_temp < 10));

        guarantee "If temperature is high, output high indication.":
            ((env_temp > 800) <=> high_temp_indicator);

        guarantee "Temperature reading equals env temperature.":
            (env_temp = temp_reading);
    **};
end TempSensor;
```

Figure 3.4: The AGREE Contract for an AADL Component Type

the component sees from the environment and *guarantees* on the outputs that constrain how the component behaves given its environment. In this example, the assumption restricts the environmental temperature to be within a range of values; the guarantee defines the behavior of the component given the environment.

Since our desire was to facilitate *behavioral error propagation*, AGREE was a suitable and obvious choice for the nominal verification tooling.

Model checker Through AGREE, the nominal model is translated into the dataflow programming language Lustre [69] which is then used as input to the JKind model checker [59]. JKind uses a series of backend SMT-solvers to generate proofs of the top level AGREE properties specified in the model. When there exists a trace such that a property is invalid, JKind provides a *counterexample* showing the state of the system in which the property is violated. An example of this is shown in Figure 3.5.

The model checker takes an adversarial role in the proof process by trying to find paths such that the proof is violated. If none exist, then the results are valid. This adversarial role is exactly what we wished to harness for this kind of analysis. If we allow faults to be active, but leave them unconstrained, this allows the model checker to determine if certain faults could violate a proof. These counterexamples could then

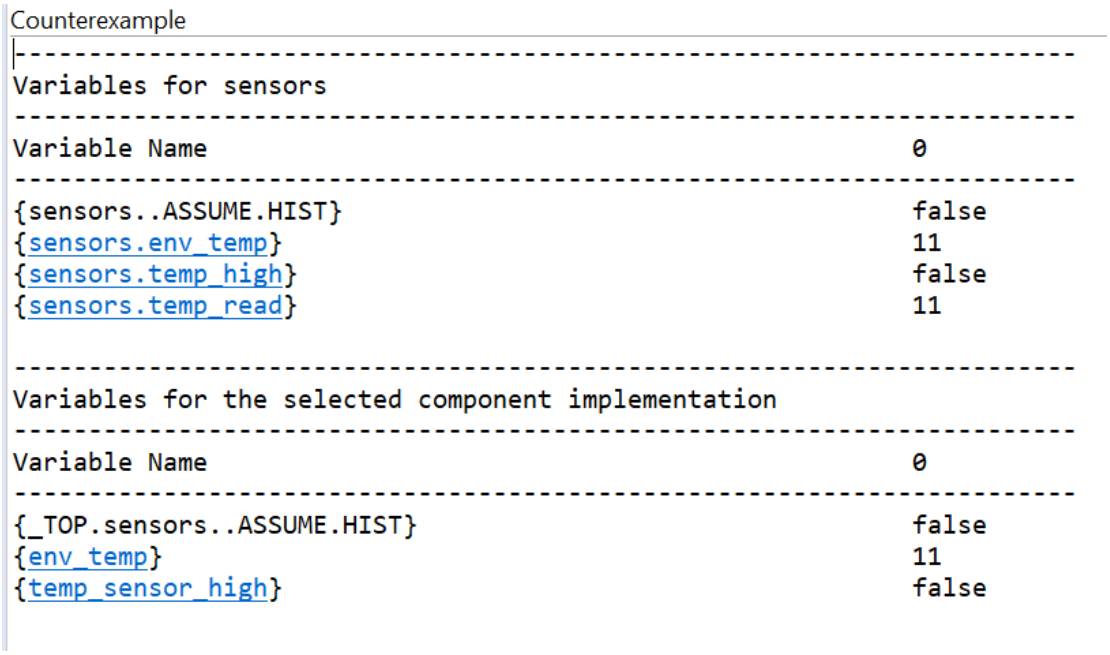


Figure 3.5: A Counterexample to an Invalid Property

contain fault information.

Behavioral error propagations Given that AGREE guarantees define the *output* behavior of components, any connected component's assumptions rely on those guarantees. If an assumption is violated, the guarantee may not hold. By associating a fault with the output of a component, this fault – when active – may violate assumptions and guarantees along the signal flow within a system. This was our goal; we wished to view the behavioral propagation of an active fault.

3.2.1 Implementation Architecture

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 [40]. The architecture of the Safety Annex is shown in Figure 3.6.

The safety language extension resides in an annex of AADL and the faults defined

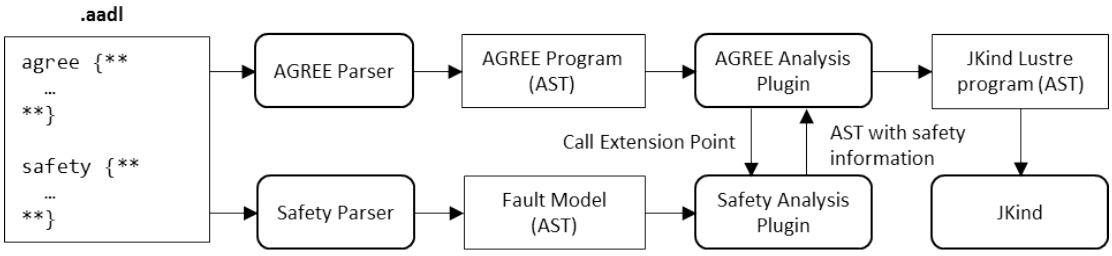


Figure 3.6: Safety Annex Plug-in Architecture

therein are translated into an abstract syntax tree and inserted into the AGREE program. The AGREE program contains the building blocks for the translation into Lustre which is the program directly analyzed by JKind.

When performing fault analysis, the fault definitions defined in the safety annex extend the AGREE contracts to allow faults to modify the behavior of component outputs. The temperature sensor subcomponent shown in Figure 3.4 encoded into Lustre is shown in Figure 3.7[1].

```
1  node temp(
2    env_temp : int;
3    high_temp_indicator : bool;
4    temp_reading : int
5  ) returns (
6    __ASSERT : bool
7  );
8  var
9    __GUARANTEE0 : bool;
10 let
11   __GUARANTEE0 = (((env_temp > 800) = high_temp_indicator)
12                   and (env_temp = temp_reading));
13   __ASSERT = (__ASSUME__HIST => __GUARANTEE0);
14 tel;
```

Figure 3.7: Temperature Component in Lustre

The inputs and outputs (lines 2-4) correspond directly to the AADL inputs and outputs of the component; likewise, the guarantee (`__GUARANTEE0`) corresponds to the guarantee on the outputs. The `__ASSERT` statement on line 13 just states that as long as the assumptions hold, the guarantee is implied.

[1]The Lustre code is slightly simplified for readability.

From the perspective of fault analysis, we want to insert a fault on the output of the component. This fault may or may not be active – it is up to the model checker. To this end, we specify three variables per potentially faulty output: `fault_nominal`, `fault_trigger`, and `fail_val`. If the trigger is true, then output failure value, else output nominal value. This can be seen in Figure 3.8: the new variables are assigned as inputs (lines 4-6) and the assert statement in line 20 shows the triggering behavior.

```
1  node temp(
2    env_temp : int;
3    time : real;
4    fault__trigger__temp__fault : bool;
5    temp__fault__fail_val : int;
6    fault__nominal__high_temp_indicator : bool;
7    high_temp_indicator : bool;
8    temp_reading : int
9
10 ) returns (
11   __ASSERT : bool
12 );
13 var
14   __GUARANTEE0 : bool;
15   temp__fault1__val_out : bool;
16 let
17   __GUARANTEE0 = (((env_temp > 800) = fault__nominal__high_temp_indicator)
18                  and (env_temp = fault__nominal__temp_reading));
19
20   __ASSERT = ((high_temp_indicator =
21                       (if fault__trigger__temp__fault
22                        then temp__fault__val_out
23                        else fault__nominal__high_temp_indicator))
24
25                  and ((temp_reading = env_temp)));
26
27   temp__fault__val_out = stuck_true(fault__nominal__high_temp_indicator,
28                                     fault__trigger__temp__fault);
29 tel;
```

Figure 3.8: Temperature Component with Fault in Lustre

This allows for the possibility of active faults, but when the faults are inactive, the nominal value is simply passed through. Line 27 of Figure 3.8 shows a call to what we call a *fault node*; this is the code that specifies the behavior of an active fault. The fault node `stuck_true` is shown in Figure 3.9. The behavior of an active fault is to output *true*. The `trigger` input to the fault node corresponds directly with the trigger defined in the temperature node of Figure 3.8 on line 4.

The model elements are translated into Lustre formulae. These are represented in JKind as a transition system, and reasoning is performed using k-induction. At each time-step of analysis, every formula in the model is given an assignment based on the constraints over that formula. If every assignment results in a provable property

```
1  node stuck_true(
2    val_in : bool;
3    trigger : bool
4  ) returns (
5    val_out : bool
6  );
7  let
8    val_out = (if trigger then true else val_in);
9
10 tel;
```

Figure 3.9: A Fault Node in Lustre

over k steps of induction, the property holds. When performing safety analysis over the model, each fault is defined as an *activation literal* and given limited constraint. If the assignment to an activation literal is *true*, this corresponds to an active fault and potentially violated guarantee. If that assignment violates a guarantee, then this violation will be reflected in the analysis results. At a system level, it can be seen if a violated guarantee will in turn violate the top level property. Hence it is seen how active faults at leaf level components violate the system level properties.

This analysis approach allows for implicit propagation of violations throughout the system. It also allows for arbitrary temporal activations of faults. There are no explicit constraints put on faults stating when an activation can occur, which allows the model checking procedure free reign to activate the faults at the worst possible times. If there are dependencies regarding fault activations, these are handled through the use of explicit error propagations.

The main constraint put on the model checker in terms of the activation of faults consist of *fault hypothesis statements*. These constrain the model by stating either the number of faults that may be active at once, or the overall probability threshold that is allowed. In the latter case, each fault has an associated probability; assuming independence, the probability of a set of faults occurring should not be less than the threshold defined.

There are two different types of fault analysis that can be performed on a fault model: verification in the presence of faults or the generation of minimal cut sets. The Safety Annex plugin intercepts the AGREE program and adds fault model information

depending on which type of fault analysis is being run. For more information on types of fault analysis, see Section 3.8.

3.3 Running Example: Sensor System

We present a running example of a simplified sensor system in a Pressurized Water Reactor (PWR). In a typical PWR, the core inside of the reactor vessel produces heat. Pressurized water in the primary coolant loop carries the heat to the steam generator. Within the steam generator, heat from the primary coolant loop vaporizes the water in a secondary loop, producing steam. The steamline directs the steam to the main turbine, causing it to turn the turbine generator, which produces electricity. There are a few important factors that must be considered during safety assessment and system design. An unsafe climb in temperature can cause high pressure and hence pipe rupture, and high levels of radiation could indicate a leak of primary coolant.

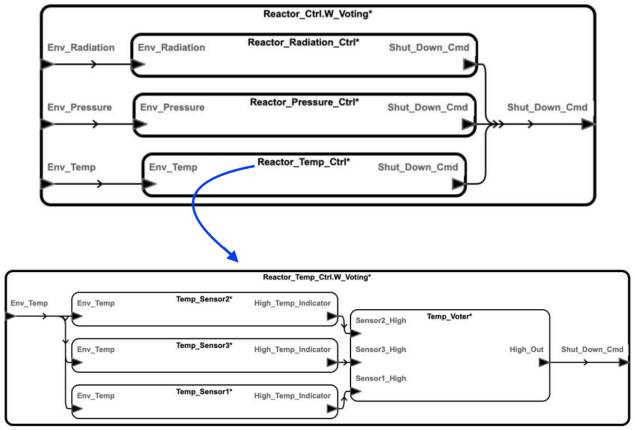


Figure 3.10: PWR Sensor System

The following sensor system can be thought of as a subsystem within a PWR that monitors these factors. A diagram of the model is shown in Figure 4.1 and represents a highly simplified version of a safety critical system. The temperature subsystem details are shown at the bottom of Figure 4.1; each of the subsystems have a similar architecture.

The subsystems each contain three sensors that monitor pressure, temperature, and radiation. Environmental inputs are fed into each sensor in the model and the redundant sensors monitor temperature, pressure, or radiation respectively. If temperature, pressure, or radiation is too high, a shut down command is sent from the sensors to the parent components.

PWR Nominal Model

The temperature, pressure, and radiation sensor subsystems use a majority voting mechanism on the sensor values and will send a shut down command based on this output. The safety property of interest in this system is: *shut down when and only when we should*; the AGREE guarantee stating this property is shown in Figure 4.2.

```
guarantee "Shut down when and only when we should":
    Shut_Down_Cmd =
        ((Env_Temp> HIGH_TEMPERATURE_THRESHOLD)  or
        (Env_Pressure > HIGH_PRESSURE_THRESHOLD) or
        (Env_Radiation > HIGH_RADIATION_THRESHOLD));
```

Figure 3.11: Sensor System Safety Property

The safety of the system requires a shut down to take place if the temperature, pressure, or radiation levels climb beyond safe levels; thus, a threshold for each subsystem is introduced. If any sensor subsystem reports passing that threshold, a shutdown command is sent. Supporting guarantees are located in each sensor subsystem and correspond to temperature, pressure, and radiation sending a shut down command if sensed inputs are above a given threshold. Each sensor has a similar guarantee.

Throughout the remainder of this chapter, we refer to the PWR example for illustrative purposes. The goal is to take the nominal system model and extend it to become a fault model using the safety annex.

3.4 Component Fault Modeling in the Safety Annex

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

When given a nominal model with which to perform safety analysis, the analyst must associate faults with each component. Examples of such faults include valves being stuck open or closed, output of a software component being nondeterministic, or power being cut off. These are often determined based on hardware specification guidelines and domain expertise. Returning to the PWR running example, each sensor for each subsystem may fail to indicate that the threshold has been surpassed. If the radiation levels are high and the radiation sensor reports no such indication, this is a fault that must be considered during analysis.

As an illustration of fault modeling using the safety annex, we look at one of the components important to the PWR system level safety property: the radiation sensor. If the radiation levels are high, the radiaton sensor reports this through an indicator to the radiation sensor subsystem. A shut down command will be sent to the top level. Figure 3.12 shows the AADL pedal sensor component with a contract for its nominal behavior. The sensor has only one input, the environmental radiation, and one output, the high radiation levels indication. The property that governs the behavior of the component is that the radiation sensor will indicate when the threshold is reached.

One possible failure for this sensor is inversion of its output value; the radiation levels are actually above threshold, but the sensor does not indicate such. This fault can be triggered with probability 1.0×10^{-52}. The safety annex definition for this fault is shown in Figure 3.13. Fault behavior is defined through the use of a fault node called *stuck_false* (shown in Figure 3.14). When the fault is triggered, the nominal output of the component (*High_Radiation_Indicator*) is replaced with its failure value (*val_out*).

Similar faults can be defined for all sensors in the PWR system. These may include stuck high (sensor reports high when threshold is not reached), stuck low (sensor does

[2]In practice, the component failure probability is collected from hardware specification sheets.

```
system Radiation_Sensor
    features
        Env_Radiation: in data port Integer;
        High_Radiation_Indicator: out data port Boolean;

    annex agree{**
        guarantee "If env radiation higher than radiation threshold,
               then indicate high radiation.":
        High_Radiation_Indicator =
            (Env_Radiation > Constants::HIGH_RADIATION_THRESHOLD);

    **};
end Radiation_Sensor;
```

Figure 3.12: PWR Radiation Sensor

```
annex safety {**

    fault Radiation_sensor_stuck_at_low "Radiation sensor stuck at low":
                                Common_Faults::stuck_false {
        inputs: val_in <- High_Radiation_Indicator;
        outputs: High_Radiation_Indicator <- val_out;
        probability: 1.0E-5 ;
        duration: permanent;
    }
**};
```

Figure 3.13: PWR Radiation Sensor Fault

not report high when threshold is reached), etc.

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

Figure 3.14: A Fault Node Definition

Given the complexity of systems, there are many types of faults that correspond to various components. The safety annex allows for complex fault behavior to be modeled through the node definitions and this can benefit safety analysts from numerous disciplines. 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. For more information on modeling asymmetric faults, see Section 3.7.

3.5 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's approach to implicit error propagation and also describes how one can model an explicit propagation by defining dependent faults.

3.5.1 Implicit Propagation

In the approach primarily used in the safety annex, 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. In this way, the safety analysis is closely tied to the behavior model of components and their requirements; the analysis is focused on the system dynamics and interactions.

In the PWR running example, all error propagations are defined implicitly.

3.5.2 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.

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 hardware fault declaration is shown in Figure 3.15 and describes a possible failure occurring between two co-located pumps.

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

Figure 3.15: 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.

Assume that there exist a green and a blue hydraulic pump; these 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 3.15. The activation of this fault triggers the activation of related faults as seen in the *propagate_to* statement shown in Figure 3.16. 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 3.16: Hardware Fault Propagation Statement

By allowing both kinds of error propagation, this allows great flexibility in modeling and allows an analyst to capture multiple kinds of faults.

3.6 Fault Hypothesis

The fault hypothesis (also referred to as the fault analysis statement) 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 cut sets to a specified maximum number of terms, and the latter is analogous to restricting the cut sets 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).

3.7 Asymmetric Fault Modeling

A *asymmetric* or *Byzantine* fault is a fault that presents different symptoms to different observers [45]. 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 5 using a process ID example.

3.7.1 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 3.17 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 3.17 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 3.17.

An asymmetric fault is defined for Component A as in Figure 3.18. This fault defines an asymmetric failure on Component A that when active, is stuck at a previous value (*prev(Output, 0)*). This can be interpreted as the following: some connected components may only see the previous value of Comp A output and others may see the correct (current) value when the fault is active. This fault definition is injected into the communication nodes and which of the connected components see an incorrect value is completely nondeterministic. Any number of the communication node faults (0...all) may be active upon activation of the main asymmetric fault.

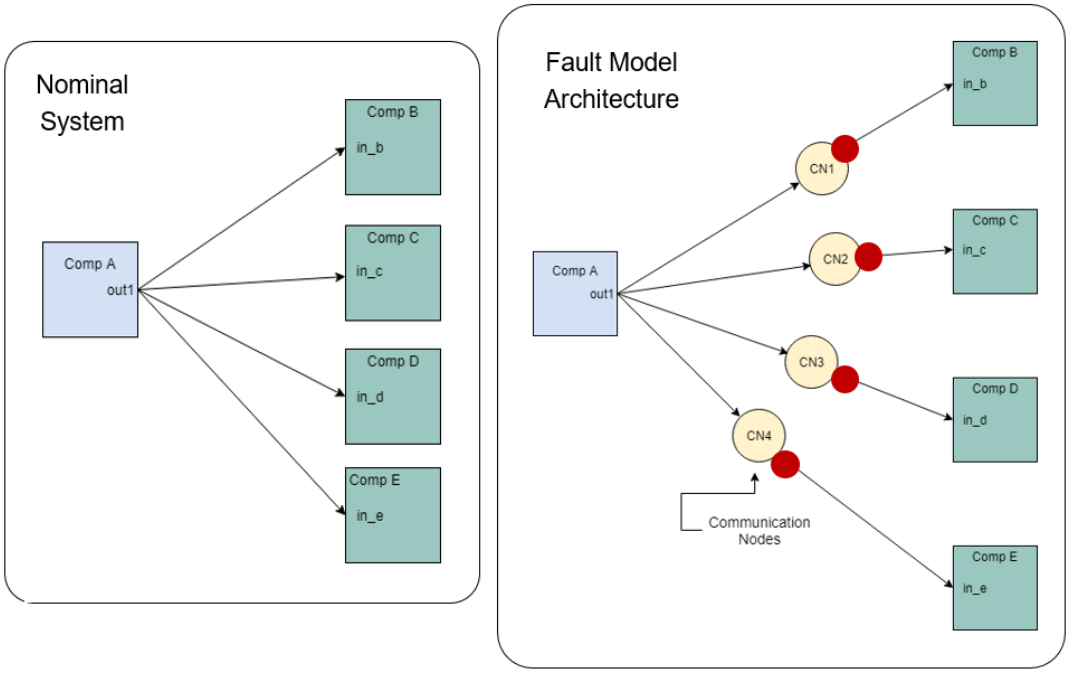


Figure 3.17: Communication Nodes in Asymmetric Fault Implementation

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

3.7.2 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`.

3.8 Verification in the Presence of Faults

There are two main options for fault model analysis using the safety annex. The first option injects faults into the Lustre program based on the restrictions placed through the fault hypothesis. The Bounded Model Checker (BMC) engine used in JKind will produce a *counterexample* to an invalid property. These counterexamples are returned to the user and include a trace of the system that causes the violation. This includes any active faults that were part of that violation. The second option is used to generate minimal cut sets for the model. The details of minimal cut set generation can be found in Chapter 4 and a full description of the analysis results for minimal cut set generation can be found there.

Counterexamples

An important feature of a bounded temporal logic model checker is the ability to find counterexamples. When the model checker determines that a formula with a universal path quantifier (e.g., for all paths...) is false, it will find a computation path which demonstrates that the negation of the formula is true [38].

In a transition system such as described in Section 2.4.5, a counterexample reveals a state of the system such that $(I, T) \nvdash P$. For a proof to hold, i.e., $(I, T) \vdash P$, it must be the case that P can be derived from (I, T) in every state and in every transition. If any valuation of the Boolean equations in T can lead to a violation of P, then P does not derive from (I, T) and a counterexample can be found.

In nominal model analysis, counterexamples provide insight into the assumptions and guarantees for each component and how they may be strengthened to support correct behavior. In fault analysis, the counterexamples include active faults and show

the effects of these faults on the proofs regarding safety properties. Assuming that the nominal model (the model in the absence of faults) proves the specified properties, if verification in the presence of faults returns a counterexample, an active fault caused the violation of the property. Insight into the state of the system when this occurs is invaluable to an analyst. This information can be used in the model-based safety assessment process to understand the effects of faults on a system and drive design change.

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 system subcomponents and a fault hypothesis statement is added to the component implementation. Given the constraint on the number of possible simultaneously active faults, the model checker attempts to prove the top level properties. If the property is violated given activation literals assigned a true value, a counterexample is provided that shows which of the faults are active and which contracts are violated. We always assume that the nominal model properties prove in the absence of faults.

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. 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.

Returning to the PWR system example, we wish to see if the faults defined for the sensors contribute to a violation of the top level safety property (shut down occurs when

and only when it should). The model has two distinct layers: top level with subsystems as subcomponents and leaf level with sensor subcomponents. If verification in the presence of faults is run compositionally with maximum 2 faults active, the results are as shown in Figure 3.19.

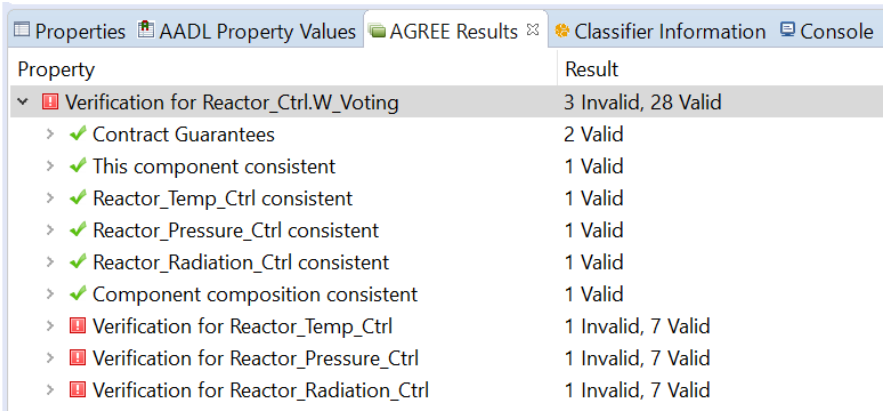


Figure 3.19: PWR Verification with Maximum Two Faults Hypothesis

The top level verification passes because faults are not defined at that level. The leaf level verification, however, does not pass; the property at the subsystem level is violated. Selecting to view the counterexample, we see that two active faults on two sensors will violate the subsystem property. The counterexample is shown in Figure 3.20.

In the state described by this counterexample, two of the radiation sensors are stuck low. The radiation level is above the given threshold, two of the three radiation sensors report low radiation values, and the voter determines the radiation to be low. This violates the safety property that we shut down when we should.

A monolithic verification in the presence of faults will show that the top level safety property is violated. In this example, the subsystem guarantee directly supports the proof of the top level safety property.

Verification in the Presence of Faults: Probabilistic Analysis

A probabilistic analysis constrains the number of *true* fault activation literals based on the combination of faults whose occurrence probability is less than the probability threshold. Allowable fault combinations are ascertained and associated Lustre assertions are added to the program. If the model checker proves that the safety properties

1	**Step**	0
2		
3	**Radiation_Sensor1**	
4	Radiation_Sensor1.Env_Radiation	401
5	Radiation_Sensor1.High_Radiation_Indicator	FALSE
6		
7	**Radiation_Sensor2**	
8	Radiation_Sensor2.Env_Radiation	401
9	Radiation_Sensor2.High_Radiation_Indicator	FALSE
10		
11	**Radiation_Sensor3**	
12	Radiation_Sensor3.Env_Radiation	401
13	Radiation_Sensor3.High_Radiation_Indicator	TRUE
14		
15	**Radiation_Voter**	
16	Radiation_Voter.High_Out	FALSE
17	Radiation_Voter.Sensor1_High	FALSE
18	Radiation_Voter.Sensor2_High	FALSE
19	Radiation_Voter.Sensor3_High	TRUE
20		
21		
22	Env_Radiation	401
23	Radiation sensor stuck at low (Radiation_Sensor1__fault_1)	TRUE
24	Radiation sensor stuck at low (Radiation_Sensor2__fault_1)	TRUE
25	Radiation sensor stuck at low (Radiation_Sensor3__fault_1)	FALSE
26	Shut_Down_Cmd	FALSE
27	shut down when and only when we should	FALSE

Figure 3.20: PWR Counterexample with Maximum Two-Faults Hypothesis

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. 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. Algorithm 1 describes the algorithm used to determine

possible fault combinations given an hypothesis threshold.

Algorithm 1: Monolithic Probability Analysis

```
1  F = {} : fault combinations above threshold ;
2  Q : faults, q_i, arranged with probability high to low ;
3  R = Q , with r ∈ R;
4  while Q ≠ {} ∧ R ≠ {} do
5      q = removeTopElement(Q) ;
6      for i = 0 : |R| do
7          prob = q × r_i ;
8          if prob < threshold then
9              removeTail(R, j = i : |R|);
10         else
11             add({q, r_i}, Q);
12             add({q, r_i}, F);
```

As shown 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 $\mathcal{R}$.

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 1, each allowable combination is inserted into an assertion in disjunctive normal form.

In the PWR system, each sensor can fail with probability 1.0×10^{-5} and the probabilistic threshold for the system safety property is given as 1.0×10^{-9}. Running monolithic verification in the presence of faults shows that given the voting mechanism, no faults will violate the safety property. By lowering the probabilistic threshold to 1.0×10^{-10}, the safety property is violated with two active faults on any sensor subsystem as shown in Figure 3.21.

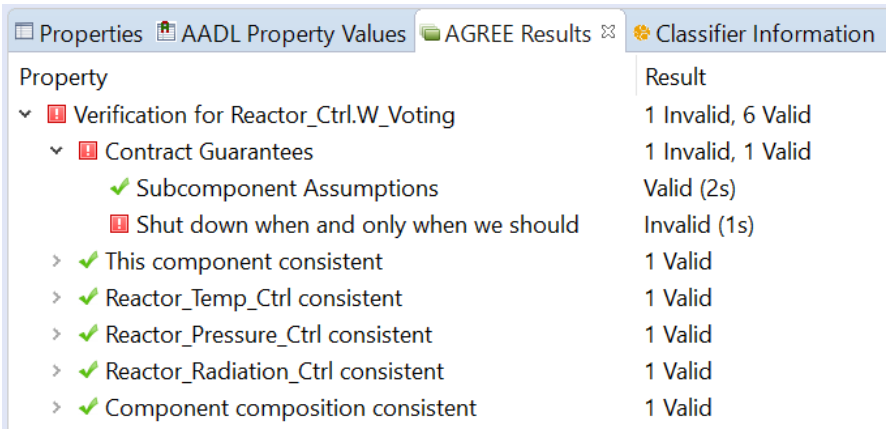


Figure 3.21: PWR Verification with Probabilistic Hypothesis

Figure 3.22 shows a portion of the counterexample provided by the model checker for the probabilistic hypothesis of 1.0×10^{-10}.

74	**Reactor_Pressure_Ctrl**	
75	Reactor_Pressure_Ctrl.Env_Pressure	0
76	Reactor_Pressure_Ctrl.Shut_Down_Cmd	FALSE
77		
78	**Reactor_Radiation_Ctrl**	
79	Reactor_Radiation_Ctrl.Env_Radiation	0
80	Reactor_Radiation_Ctrl.Shut_Down_Cmd	FALSE
81		
82		
83	Env_Pressure	0
84	Env_Radiation	0
85	Env_Temp	301
86	Pressure sensor stuck low (Pressure_Sensor1__fault_1)	FALSE
87	Pressure sensor stuck low (Pressure_Sensor2__fault_1)	FALSE
88	Pressure sensor stuck low (Pressure_Sensor3__fault_1)	FALSE
89	Radiation sensor stuck at low (Radiation_Sensor1__fault_1)	FALSE
90	Radiation sensor stuck at low (Radiation_Sensor2__fault_1)	FALSE
91	Radiation sensor stuck at low (Radiation_Sensor3__fault_1)	FALSE
92	Shut down when and only when we should	FALSE
93	Shut_Down_Cmd	FALSE
94	Temp sensor stuck at low (Temp_Sensor1__fault_1)	FALSE
95	Temp sensor stuck at low (Temp_Sensor2__fault_1)	TRUE
96	Temp sensor stuck at low (Temp_Sensor3__fault_1)	TRUE

Figure 3.22: PWR Counterexample with Probabilistic Hypothesis

The counterexample shows the active faults on the temperature subsystem: two faults are simultaneously active which violates the safety property.

A benefit of utilizing the capabilities of counterexample generation is to ability for an analyst to view the dynamic behavior of a complex system and be able to understand the signal flow between behaviorally connected components. This aids in the

design process when using model-based engineering methods and can greatly assist in design restructuring changes that may need to occur. It can be easy to see through counterexamples when a system is not resilient to a single fault, or when a combination of faults are sufficiently likely to occur together. Using these analysis capabilities support the objective to provide formal feedback to a safety engineer during the analysis of a complex critical system.

Up until this point, we have discussed modeling capabilities and counterexample generation using the safety annex for MBSA, but these counterexamples produce only a single path towards violation; in practice, safety analysts seek to find all minimal paths towards violation in terms of minimal cut sets and their associated fault trees. We turn our attention now towards an extension of the transition system such that we can reason about non-deterministic guarantees, their effect on the system, and how we can compositionally derive sets of fault trees and minimal cut sets.

Chapter 4

Compositional Minimal Cut Set Generation

Given the importance of minimal cut set computations for industrial sized systems, much research has been done on their generation (e.g., [29, 49, 117, 118, 127]). As described in preliminary sections, the methods of cut set generation have varied greatly depending on how the system is modeled – transition systems, state machines, decision diagrams, to name a few– and what kind of model checking is performed – symbolic algorithms, bounded model checking, etc. Compositional minimal cut set generation has not, to our knowledge, been previously explored. Due to the compositional nature of the verification it is difficult 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.

A 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) [130] 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 [6, 52].

(2) A compositional verification framework was required for the analysis of AADL

models. The Assume-Guarantee Reasoning Environment (AGREE) is a tool for formal analysis of behaviors in AADL models [40]. 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 [69] and then queries the model checker JKind [59] 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 the model checker provides 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 [140, 141, 143]. In the Safety Annex approach, AGREE contracts are used to define the nominal behavior of system component and the nominal model is verified using 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.

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 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 that traces a safety property to a minimal set of model elements necessary for proof; this is called the *all minimal inductive validity core* algorithm (`All_IVCs`) [12, 64, 66]. Inductive validity cores produce the minimal set of model elements necessary to prove a property. Each set contains the

behavioral contracts – the requirement specifications for components – of the model used in a proof. When the `All_IVCs` algorithm is run, this gives the minimal set of contracts required for proof of a safety property. If all of these sets are obtained, we have insight into every proof path for the property. Thus, if we violate at least one contract from every MIVC set, we have in essence "broken" every proof path. This is the information that is used to perform fault analysis using MIVCs.

Safety analysts are often concerned with faults in the system, i.e., when components or subsystems deviate from nominal behavior, and the propagation of errors through the system. To this end, the model elements included in the reasoning process of the `All_IVCs` algorithm are not only the contracts of the system, but faults as well. This will provide additional insight into how an active fault may violate contracts that directly support the proof of a safety property.

Before the specifics of the algorithm and proofs can be discussed, some background definitions are required. Throughout this chapter, a running example is referenced and we provide the description here.

4.2 Running Example

To illustrate the generation of minimal cut sets through the use of IVCs, we present a running example of a sensor system in a Pressurized Water Reactor (PWR). In a typical PWR, the core inside of the reactor vessel produces heat. Pressurized water in the primary coolant loop carries the heat to the steam generator. Within the steam generator, heat from the primary coolant loop vaporizes the water in a secondary loop, producing steam. The steamline directs the steam to the main turbine, causing it to turn the turbine generator, which produces electricity. There are a few important factors that must be considered during safety assessment and system design. An unsafe climb in temperature can cause high pressure and hence pipe rupture, and high levels of radiation could indicate a leak of primary coolant. The following sensor system can be thought of as a subsystem within a PWR that monitors these factors. A diagram of the AADL model is shown in Figure 4.1 and represents a highly simplified version of a safety critical system. May add cartoon figure demonstrating this model/process later - depending on space. Also can adjust figure placement after rewriting, adding, and cutting is done.

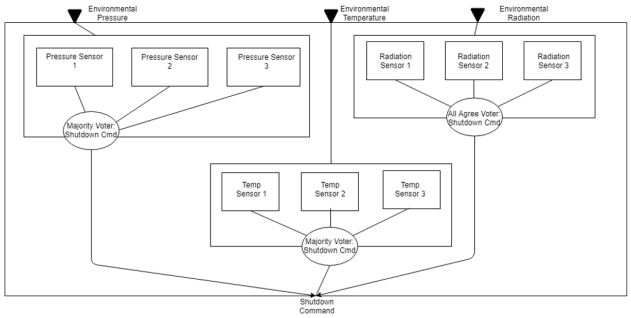


Figure 4.1: Pressurized Water Reactor Sensor System

4.2.1 PWR Nominal Model

The "top-level" system is an abstraction of the PWR and contains sensor subsystems. The subsystems contain sensors that monitor pressure, temperature, and radiation. Environmental inputs are fed into each sensor in the model and the redundant sensors monitor temperature, pressure, or radiation respectively. If temperature, pressure, or radiation is too high, a shut down command is sent from the sensors to the parent components.

The temperature, pressure, and radiation sensor subsystems use a voting mechanism on the redundant sensor values and will send a shut down command based on this output. The safety property of interest in this system is: *shut down when and only when we should*; the AGREE guarantee stating this property is shown in Figure 4.2.

```
guarantee "Shut down when and only when we should":
    Shut_Down_Cmd =
        ((Env_Temp> HIGH_TEMPERATURE_THRESHOLD)  or
        (Env_Pressure > HIGH_PRESSURE_THRESHOLD) or
        (Env_Radiation > HIGH_RADIATION_THRESHOLD));
```

Figure 4.2: Sensor System Safety Property

The safety of the system requires a shut down to take place if the temperature, pressure, or radiation levels become unsafe; thus, a threshold is introduced and if any sensor subsystem reports passing that threshold, a shutdown command is sent. But on the other hand, we do not want to shut down the system if it is not necessary. If a sensor reports high temperature erroneously and a shut down occurs, this costs time and money.

Supporting guarantees are located in each sensor subsystem and correspond to temperature, pressure, and radiation sending a shut down command if sensed inputs are above a given threshold. Each sensor has a similar guarantee.

4.2.2 PWR Fault Model

The faults that are of interest in this example system are any one of the sensors failing high or low. If sensors report high and a shut down command is sent, we shut down when we should not. On the other hand, if sensors report low when it should be high, a shut down command is not sent and we do not shut down when we should. For the remainder of this example, we focus on the failures when sensors report low when they should not.

Two faults are defined with the safety annex for each sensor in the system. An example

```
annex safety {**
    fault temp_sensor_stuck_at_high "temp sensor stuck at high": Common_Faults.stuck_true {
        inputs: val_in <- High_Temp_Indicator;
        outputs: High_Temp_Indicator <- val_out;
        probability: 1.0E-5 ;
        duration: permanent;
    }
**};
```

Figure 4.3: Fault on Temperature Sensor Defined in the Safety Annex for AADL

The Safety Annex provides a way to weave the faults into the nominal model by use of the *inputs* and *outputs* keywords. This allows users to define a fault and attach it to the output of a component. If the fault is active, the error can then in essence violate the guarantees of this component and possibly the assumptions of downstream components [141]. The activation of a fault is not up to the user, but instead left up to the backend model checker, JKind, to determine if the activation of this fault will contribute to a violation of higher level guarantees. If so, it can be activated during the analysis.

For simplicity, throughout this paper we refer only to faults that fail low (i.e., the environmental input is actually high which warrants a shut down command, but the sensor reports that it is within safe ranges). This simplification is presented to keep the example and results described concise. For ease of reference, a table is provided giving model elements of interest in the sensor example. We refer to these throughout this section. Note: the thresholds vary for pressure, temperature, and radiation. These are given as constants T_p, T_t, and T_r respectively. The shutdown command is defined notationally as S. The faults are shown as "fail low" which correspond to the temp (or

pressure or radiation) being high, but the sensor reports safe ranges. We also do not list all guarantees and assumptions that are in the model, but only the ones of interest for this analysis. Still messing around with how to display this in a way that it isn't messy, doesn't take up a ton of space, and am not currently happy with this approach. But I really hated the tables. Too much info for what is actually needed. Will keep working on this.

PWR System: $P = ((temp > T_t) \vee (pressure > T_p) \vee (radiation > T_r)) \iff S$

Temp Subsystem : $G_t = temp > T_t \iff S$

Pressure Subsystem: $G_p = pressure > T_p \iff S$

Radiation Subsystem: $G_r = radiation > T_r \iff S$

Temp Sensors (3): $g_p = pressure > T_p \iff S$, Fault f_{ti}: fails low for $i = 1, 2, 3$.

Pressure Sensors (3): $g_r = radiation > T_r \iff S$, Fault f_{pi}: fails low for $i = 1, 2, 3$.

Radiation Sensors (3): $g_r = radiation > T_r \iff S$, Fault f_{ri}: fails low for $i = 1, 2, 3$

4.3 Preliminaries for Minimal Cut Set Generation

In this research we consider *safety properties* over infinite-state machines. The states are vectors of variables that define the values of state variables. We assume there are

a set of legal *initial states* and the safety property is specified as a formula over state variables. A *reachable state space* means that all states are reachable from the initial state.

Given a state space U, a transition system (I, T) consists of an initial state predicate $I : U \to \mathit{bool}$ and a transition step predicate $T : U \times U \to \mathit{bool}$. We define the notion of reachability for (I, T) as the smallest predicate $R : U \to \mathit{bool}$ which satisfies the following formulas:

$$\forall u.\ I(u) \Rightarrow R(u)$$
$$\forall u, u'.\ R(u) \wedge T(u, u') \Rightarrow R(u')$$

A safety property $P : U \to \mathit{bool}$ is a state predicate. A safety property P holds on a transition system (I, T) if it holds on all reachable states, i.e., $\forall u.\ R(u) \Rightarrow P(u)$, written as $R \Rightarrow P$ for short. When this is the case, we write $(I, T) \vdash P$.

4.3.1 Induction

For an arbitrary transition system (I, T), computing reachability can be very expensive or even impossible. Thus, we need a more effective way of checking if a safety property P is satisfied by the system. The key idea is to over-approximate reachability. If we can find an over-approximation that implies the property, then the property must hold. Otherwise, the approximation needs to be refined.

A good first approximation for reachability is the property itself. That is, we can check if the following formulas hold:

$$\forall s.\ I(s) \Rightarrow P(s) \tag{4.1}$$
$$\forall s, s'.\ P(s) \wedge T(s, s') \Rightarrow P(s') \tag{4.2}$$

If both formulas hold then P is *inductive* and holds over the system. If (4.1) fails to hold, then P is violated by an initial state of the system. If (4.2) fails to hold, then P is too much of an over-approximation and needs to be refined.

The JKind model checker used in this research uses k*-induction* which unrolls the property over k steps of the transition system. For example, 1-induction consists of

$$
\begin{gathered}
I(s_0) \Rightarrow P(s_0) \\
\vdots \\
I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-2}, s_{k-1}) \Rightarrow P(s_{k-1}) \\
P(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge P(s_{k-1}) \wedge T(s_{k-1}, s_k) \Rightarrow P(s_k)
\end{gathered}
$$

Figure 4.4: k-induction formulas: k base cases and one inductive step

formulas (4.1) and (4.2) above, whereas 2-induction consists of the following formulas:

$$
\begin{gathered}
\forall s.\, I(s) \Rightarrow P(s) \\
\forall s, s'.\, I(s) \wedge T(s, s') \Rightarrow P(s') \\
\forall s, s', s''.\, P(s) \wedge T(s, s') \wedge P(s') \wedge T(s', s'') \Rightarrow P(s'')
\end{gathered}
$$

That is, there are two base step checks and one inductive step check. In general, for an arbitrary k, k-induction consists of k base step checks and one inductive step check as shown in Figure 4.4 (the universal quantifiers on s_i have been elided for space). We say that a property is k-inductive if it satisfies the k-induction constraints for the given value of k. The hope is that the additional formulas in the antecedent of the inductive step make it provable.

In practice, inductive model checkers often use a combination of the above techniques. Thus, a typical conclusion is of the form "P with lemmas $L_1, \ldots, L_n$ is k-inductive".

4.3.2 The SAT Problem

Boolean Satisfiability (SAT) solvers attempt 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 [90]. 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.

There are several ways of translating a propositional formula into clauses such that satisfiability is preserved [48]. By performing this translation, k-inductive model checkers are able to utilize parallel SAT-solving engines to glean information about the proof of a safety property at each inductive step. Expression of the base and induction steps of a temporal induction proof as SAT problems is straightforward. As an example, we look at an arbitrary base case from Figure 4.4.

$$I(s_0) \wedge T(s_0, s_1) \wedge \cdots \wedge T(s_{k-2}, s_{k-1}) \wedge \neg P(s_{k-1})$$

When proving correctness it is shown that the formulas are *unsatisfiable*. If an n^{th} inductive-step is unsatisfiable, that means following an n-step trace where the property holds, there exists no next state where it fails, i.e., the property P is provable.

4.4 Formalization of the Method

Section desperately needs figures of some kind to break up the text. Will try to make a few small ones of PWR example results. Compositional analysis proceeds from the top layer of the architecture down through the system model; faults are defined on leaf level components and guarantees are defined on all components. Due to the difference in analysis per layer, this section focuses on the formalism per layer type we are in.

Given an initial state I and a transition relation T consisting of conjunctive constraints as defined in section **??**. The nominal guarantees of the system, G, consist of conjunctive constraints $g \in G$. Given no faults, each g is one of the transition constraints T_i where:

$$T_n = g_1 \wedge g_2 \wedge \cdots \wedge g_n \tag{4.3}$$

We assume the property holds of the nominal relation $(I, T_n) \vdash P$. Given that our focus is on safety analysis in the presence of faults, let the set of all faults in the system be denoted as F. A fault $f \in F$ is a deviation from the normal constraint imposed by a guarantee. Any "faults" in a mid-layer are simply violated guarantees, or deviations from normal behavior.

4.4.1 Top Layer of Compositional Analysis

Since faults are defined at leaf layers of the architecture, the top (and middle) layers only contain guarantees in the analysis. The `All_IVCs` algorithm collects all *minimal unsatisfiable subsets* (MUSs) of a given transition system in terms of the *negation* of the top level property [12, 66]. Formally, an MUS of a constraint system C is a set $M \subseteq C$ such that M is unsatisfiable and $\forall c \in M : M \setminus \{c\}$ is satisfiable. 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.

Returning to our running example, this can be illustrated by the following. Given the constraint system $C = \{G_p, G_t, G_r, \neg P\}$, a minimal explanation of the infeasability of this system is the set $\{G_p, G_t, G_r, \}$. If all three guarantees hold, then P is provable.

A related set is a *minimal correction set* (MCS); a 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.

In the case of an UNSAT system, we may ask: what will correct this unsatisfiability? Returning to the PWR example, we can find the MCSs of the top level constraint system: $MCS_1 = \{G_t\}$, $MCS_2 = \{G_p\}$, $MCS_3 = \{G_r\}$. If any single guarantee is violated, a shut down from that subsystem will not get sent when it should and the safety property P will be violated.

A duality exists between the MUSs of a constraint system and the MCSs as established by Reiter [125]. 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 [42, 90, 125].

For the PWR top level constraint system, it can be seen that each of the MCSs intersected with the MUS is nonempty. And now we have the minimal set of guarantees for which, if violated, will cause P to be unprovable.

4.4.2 Leaf Layer of Compositional Analysis

The faults in the safety annex are defined on leaf level components. Thus, for the lowest analysis layer, we must take into consideration faults and the guarantees their activation may violate. A fault $f \in F$ is a deviation from the normal constraint imposed by a guarantee. For the purposes of this paper, each guarantee at the leaf layer of analysis has an associated fault. Without loss of generality, we associate a single fault and an associated fault probability with a guarantee. Each fault f_i is associated with an *activation literal*, af_i, that determines whether the fault is active or inactive.

To consider the system under the presence of faults, consider a set GF of modified guarantees in the presence of faults and let a mapping be defined from activation literals $af_i \in AF$ to these modified guarantees $gf_i \in GF$.

$$\sigma : AF \rightarrow GF$$
$$gf_i = \sigma(af_i) = \text{if } af_i \text{ then } f_i \text{ else } g_i$$

The transition system is composed of the set of modified guarantees GF and a set of conjunctions assigning each of the activation literals $af_i \in AF$ to false:

$$T = gf_1 \wedge gf_2 \wedge \cdots \wedge gf_n \wedge \neg af_1 \wedge \neg af_2 \wedge \cdots \wedge \neg af_n \tag{4.4}$$

Lemma 1. *If $(I, T_n) \vdash P$ for T_n defined in equation 4.3, then $(I, T) \vdash P$ for T defined in equation 4.4.*

Proof. By application of successive evaluations of σ on each constrained activation literal $\neg af_i$, the result is immediate. □

Consider the elements of T as a set $GF \cup AF$, where GF are the potentially faulty guarantees and AF consists of the activation literals that determine whether a guarantee is faulty. This is a set that is considered by a SAT-solver for satisfiability during the k-induction procedures. The posited problem is thus: $GF \wedge AF \wedge \neg P$ for the safety property in question. Recall, if this is an *unsatisfiable* constraint system, then $(I, T) \vdash P$. On the other hand, if it is *satisfiable*, then we know that given the constraints in GF and AF, P is not provable. These are the exact constraints we wish to find.

Let us view this in terms of the PWR system example and focus on the temperature sensor subsystem. The safety property to be proved is G_t, the supporting guarantees are found in each of the three temperature sensors, g_{ti}. Faults f_{ti} are defined for each sensor. The transition system is:

$$T = gf_{t1} \wedge gf_{t2} \wedge gf_{t3} \wedge \neg af_{t1} \wedge \neg af_{t2} \wedge \neg af_{t3}$$

The MIVCs for this subsystem layer correspond to all pairwise combinations of constrained activation literals. Intuitively, if any two sensor faults do *not* occur, then two of the three sensor guarantees are not violated and the system responds appropriately to high temperature; therefore, G_t is provable.

The MCSs for this subsystem layer happen to also correspond to all pairwise combinations of constrained activation literals. If any two sensor faults *do* occur, then two of the three sensor guarantees will be violated and the system does not respond to high temperature as required. This would result in the inability to prove G_t. (Note: it is not always the case that the MCSs are the same as the MIVCs – in this case it is due to majority voting on three sensors.)

4.4.3 Transforming MCS into Minimal Cut Set

The MCSs contain the information needed to find minimal cut sets, but their elements consist of constrained activation literals and/or guarantees. The link between the activation literals, faults, and guarantees is defined through σ mapping (equation 4.4.2). At the leaf layer, only activation literals are found in MCSs and σ must be applied to each element in an MCS to map back to the associated fault. Without loss of generality, let $MCS = \{af_1, \cdots, af_m\}$. Let $\sigma(MCS) = \{\sigma(\neg af_1), \cdots, \sigma(af_m)\}$ be a

mapping where MCS is a minimal correction set with regard to some property G and $MCS \subseteq AF$. Question: Does minimality need its own proof?

Lemma 2. *$\sigma(MCS)$ is a minimal cut set of G.*

Proof. Assume towards contradiction that $\sigma(MCS)$ is not a cut set of G. Then $gf_1 \wedge \cdots \wedge gf_n \wedge af_1 \cdots \wedge af_m \wedge \neg af_{k+1} \wedge \neg af_n \wedge \neg G$ is unsatisfiable. Thus, the *true* activation literals do not affect the provability of G. This contradicts $C \setminus MCS$ is satisfiable. Minimality follows directly from the definition of MCS. □

In terms of the PWR example, the minimal cut sets for the temperature subsystem property G_t consist of all pairwise faults on the temperature sensors; if any two faults occur on the sensors at the same time, we violate the temperature subsystem guarantee.

Once these lower level minimal cut sets are generated, it is a matter of simple set replacement to find the higher level minimal cut sets. This can be easily seen in our example. An MCS at the top level has the element G_t. We systematically replace the contract with the faults that cause their violation. This results in three distinct minimal cut sets for P from the temperature subsystem: $\{f_{t1}, f_{t2}\}, \{f_{t1}, f_{t3}, \{f_{t2}, f_{t3}$. All minimal cut sets for P are given as similar pairwise combinations from each subsystem and total 9 for the entire system.

Seems I need a theorem to round it out: that replacement will give min cut sets of safety property. Will think about how to formulate this.

4.5 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.

Implementation in Lustre

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.

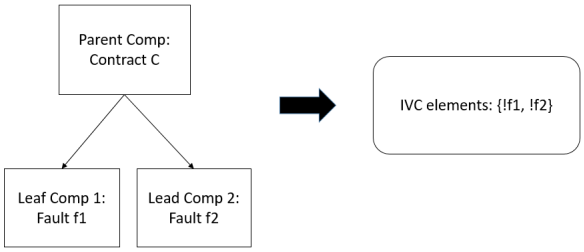


Figure 4.5: IVC Elements used for Consideration in a Leaf Layer of a System

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 4.5. In the non-leaf layers of the program, both contracts and constrained faults are considered as shown in Figure 4.6. 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

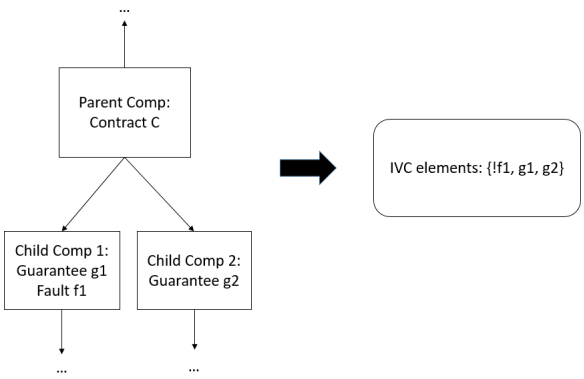


Figure 4.6: IVC Elements used for Consideration in a Middle Layer of a System

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.

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 [60, 90, 104]. 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 2 describes this process.

Algorithm 2: MinCutSets Generation Algorithm

```
1  Function replace(P):
2      List(I):= List(MCS) for P with all constraints removed ;
3      for all I ∈ List(I) do
4          if there exists contracts g ∈ I then
5              for all constrained contracts g ∈ I do
6                  if there exists MinCutSets for g in lookup table then
7                      for all minCut(g) do
8                          I_repl = I ;
9                          I_repl := replace g with minCut(g) ;
10                         add I_repl to List(I) ;
11                 else
12                     replace(g) ;
13         else
14             add I as minCut(g) for P ;
```

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.

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:

$$R \leq \sum_{i=1}^{\alpha}(\prod_{j=1}^{i} |Cut(g_j)|) \tag{4.5}$$

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 1. *Algorithm 2 terminates*

Proof. No infinite sets are generated by the `All_IVCs` or minimal hitting set algorithms [66, 104]; 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 4.5). □

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 2 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

Case Studies

This chapter serves to illustrate the safety annex in terms of both modeling and analysis of various systems. The first section outlines a large case study example called the Wheel Brake System for an aircraft; the nominal system is modeled and extended by the safety annex. We discuss the results in terms of scalability and present the timing and minimal cut set results. An example showing the flexibility of fault modeling with the safety annex is given in the following section. The chapter concludes with a discussion on analysis timing results for a set of models.

5.1 Wheel Brake System

To demonstrate the fault modeling capabilities of the Safety Annex we will use the Wheel Brake System (WBS) described in AIR6110 [4]. This system is a well-known example that has been used as a case study for safety analysis, formal verification, and contract based design [19, 23, 25, 79]. The preliminary work for the Safety Annex was based on a simple model of the WBS [143]. 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 [25].

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

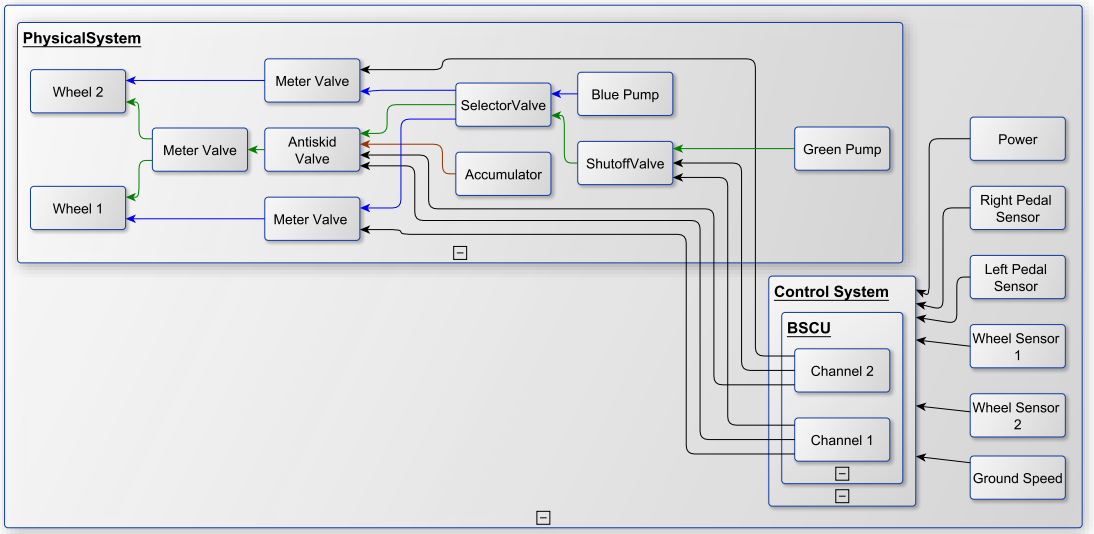


Figure 5.1: Simplified Two-Wheel WBS

antiskid braking. The physical system consists of the hydraulic circuits running from hydraulic pumps to wheel brakes as well as valves that control the hydraulic fluid flow. This system provides braking force to each of the eight wheels of the aircraft. The wheels are all mechanically braked in pairs (one pair per landing gear). For simplicity, Figure 5.1 displays only two of the eight wheels.

There are three operating modes in the WBS model:

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

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

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

Figure 5.2: Safety Property: Inadvertent Braking

5.1.1 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 [7, 39] and we refer you to Section 2.4.

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 5.1. 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 over 440 assumptions and guarantees after instantiation.

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 5.1: Safety Properties of the WBS

5.1.2 Fault Model Analysis

There are two main options for fault model analysis using the Safety Annex. The first option injects faults into the Lustre program based on the restrictions placed through the fault hypothesis. The bounded model checker engine used in JKind will find counterexamples to an invalid property. These counterexamples are returned to the user and include a trace of the system state that causes the violation. This includes any active faults that were part of that violation. The second option is used to generate minimal cut sets for the model. The fault activation literals and supporting guarantees are added

to the `--%IVC` elements as described in Sections 4.4 and 4.5, the algorithms generating the cut sets are run (Section 4.5), and the results are displayed to the user.

We outline here the results of the fault analysis for the WBS. All analyses were run on an Intel Core i7 with a 2.80GHz CPU and 16 GB RAM.

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. More detail on verification of fault models can be found in Section 3.8.

The WBS was verified in the presence of faults given a `max 1 fault` hypothesis using compositional analysis. The time for complete model analysis was approximately 9 minutes, but a counterexample for certain top level properties took only around 20 seconds. (Recall that when using compositional verification in the presence of faults, that hypothesis applies to each layer separately – the results are not rolled up as in the compositional generation of minimal cut sets. The counterexample given in this analysis pertains only to faults and contracts *in a given layer*, and the timing of the counterexample reflects this single layer analysis result.)

The verification in the presence of faults with `max 1 fault` hypothesis statement provided a counterexample to the property *S18-WBS-0325: never inadvertent braking of wheel i*, for $i = 1, \ldots 8$. This property is given in Figure 5.2. The intuition behind the failure is that the pedal was not pressed, the sensor stated that it was pressed, braking was commanded through the digital components, and brake pressure was supplied at the wheel. This sensor fault was a single point of failure with regard to all of the inadvertent braking properties. Later in this section we look at the sensor on the pedal position in closer detail from the lens of architectural design changes led by the analysis results.

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 Chapter 4 of this dissertation and for more details on the probabilistic hypothesis algorithm and analysis results, see Section 3.8.

Recall that when using the `max 1 fault` hypothesis statement on the WBS, we found that the sensor was a single point of failure for multiple properties. The probability of this particular sensor failing is given in AIR6110 [4] as 1.0×10^{-2}. The probabilistic hypothesis was set according to the thresholds given per property (see Table 5.1) and the analysis was run monolithically on the WBS model. The total time *to generate a counterexample* for violated properties using a probabilistic hypothesis with monolithic analysis varied depending on the safety property; the range of times was between 15 seconds and 9 minutes. The property that took the longest to generate a counterexample for was *S18-WBS-0323: never inadvertent braking with all wheels locked shall be less than* 1.0×10^{-9} *per takeoff.* The formula for this property references all 8 wheels and numerous subcomponents, most of which have faults associated with them. The low probability threshold combined with the formula complexity and the exponential increase in fault combinations likely served to make this the longest time in counterexample generation. For a full discussion on probabilistic compositional analysis, see Section 5.1.2.

Generate Minimal Cut Sets: Max *n* Analysis

As described in Chapter 4, the generation of minimal cut sets uses the `All_IVCs` 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 than or equal to the number n specified in the fault hypothesis and displayed to the user.

Generate minimal cut set analysis was performed on the Wheel Brake System and results are shown in Table 5.6. Notice in Table 5.6, the label across the top row refers to the cardinality (n) 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. Table 5.6 shows the total number of cut sets of cardinality n. The total number of cut sets computed at the given threshold is the sum across a row. (For the full text of the properties, see Table 5.1.)

Property	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0321	7	0	0	256	57,600
0322-R	75	0	0	0	0
0322-L	75	0	0	0	0
0323	182	0	0	0	0
0324	8	3,665	28,694	883,981	-
0325-WX	33	0	0	0	0

Table 5.2: WBS Minimal Cut Set Results for Max n Hypothesis

As can be seen in Table 5.6, the number of cut sets increases proportional to the cardinality of the cut sets. Intuitively, this can be understood as simple combinations of faults that can violate the hazard; if more things go wrong in a system at the same time, the more likely a property will be violated. Property S18-WBS-0324 with a max fault hypothesis of 5 was unable to finish due to an out of memory error. At the time that the error was thrown, the number of cut sets exceeded 1.5 million. In practice, it is not likely that an analyst will manually sift through a million or more cut sets, but rather will filter out the combinations that are sufficiently unlikely to occur. A probabilistic approach would be warranted in these situations.

The next analysis shows the difference between the time it takes to generate all MIVCs and the time it takes to transform those MIVCs into minimal cut sets. Each column of Table 5.7 labeled with the value of n gives the fault hypothesis threshold for that analysis run. For comparison with the number of minimal cut sets generated per property, we refer to Table 5.6[1]. The overall time of the algorithms used for minimal

[1]The property S18-WBS-0325-WX is symmetric for all eight wheels. For readability, we only include wheel one verification timing results in Table 5.7. Likewise for property 0322-L/R.

Property	MIVC Gen	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0321	396.417	5.913	5.468	5.61	5.636	11.925
0322-L	407.078	5.931	5.435	5.302	5.268	5.243
0323	412.926	6.397	6.452	6.420	5.309	5.459
0324	446.610	41.334	41.744	44.062	69.142	-
0325-W1	391.137	5.632	5.388	5.359	5.301	5.236

Table 5.3: WBS Analysis Time in Seconds

cut set generation is quite small compared to the nominal analysis and extended model analysis time. This is not altogether surprising; the nominal and extended analysis is contending with an infinite state model checking problem whereas the algorithms presented to generate minimal cut sets deal with set element algorithms and boolean formula manipulation. The greatest time recorded was for property S18-WBS-0324 at $n = 4$; the reason is clear when comparing this with Table 5.6: the total number of minimal cut sets computed for this threshold is 916,349.

Generate Minimal Cut Sets: Probabilistic Analysis

Both probabilistic analysis and max n analysis use the same underlying minimal cut set generation algorithm (see Section 4.5), but 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 hypothesis.

The probabilistic analysis for the WBS was given a top level threshold per property as stated in AIR6110 and shown in Table 5.1. The faults associated with various components were all given probability of occurrence according to the AIR6110 document [4]. The table shows the property name and associated probability. The generation of minimal cut sets provided all sets that violate that property whose combined probabilities (assuming independence) are greater than the threshold. The number of sets per cardinality are listed in the table.

As shown in Table 5.4, 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

Property	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0321: 5.0×10^{-7}	7	0	0	256	0
0322-R: 5.0×10^{-7}	75	0	0	0	0
0322-L: 5.0×10^{-7}	75	0	0	0	0
0323: 1.0×10^{-9}	182	0	0	0	0
0324: 1.0×10^{-9}	8	3665	0	0	0
0325-W1: 1.0×10^{-9}	33	0	0	0	0

Table 5.4: WBS Minimal Cut Set Results for Probabilistic Hypotheses

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 on Table 5.4 that the likely contributors to a hazard are minimal cut sets of cardinality one. The probabilistic analysis eliminated many thousands of cut sets from consideration. The total computation time for these runs is given in Table 5.5.

Property	Total Analysis Time
R-0321: 5.0×10^{-7}	433.144 sec
R-0322: 5.0×10^{-7}	431.954 sec
L-0322: 5.0×10^{-7}	429.081 sec
0323: 1.0×10^{-9}	571.216 sec
0324: 1.0×10^{-9}	589.07 sec
0325-W1: 1.0×10^{-9}	430.021 sec

Table 5.5: WBS Minimal Cut Set Time for Probabilistic Hypothesis

It is clear that the lower probabilistic threshold for properties 0323-0325-WX allowed more fault combinations to be possible which increased computation time. One must also recall property 0324 from the max *n* analysis whose threshold of $n = 4$ produced close to a million cut sets of various cardinalities. The pruning according to probabilities cut out many of these sets from consideration; they are sufficiently unlikely to occur together.

Display Results from Generate Minimal Cut Sets

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

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 5.3.

```
Minimal Cut Sets for property violation:
property lustre name: safety___GUARANTEE0
property description: (WheelBrake) braking force is positive if and only if
                          either normal or alternate pressure is positive.
Total 8 Minimal Cut Sets
Minimal Cut Set # 1
Cardinality 1
original fault name, description: HydraulicPiston_StuckAtLastPosition,
        "(HydraulicPiston) Stuck at last value fault."
lustre component, fault name: safety,
        fault__independently__active__alt_hyd_piston__alt_hyd_piston__fault_3
failure rate, default exposure time: 3.3E-5, 1.0

Minimal Cut Set # 2
Cardinality 1
original fault name, description: BrakeActuator_FailedFullOff,
        "(BrakeActuator) Stuck closed fault."
lustre component, fault name: safety,
        fault__independently__active__brake_actuator__brake_actuator__fault_1
failure rate, default exposure time: 3.3E-6, 1.0

Minimal Cut Set # 3
Cardinality 1
original fault name, description: BrakeActuator_StuckAtLastPosition,
        "(BrakeActuator) Stuck at last value fault."
lustre component, fault name: safety,
        fault__independently__active__brake_actuator__brake_actuator__fault_3
failure rate, default exposure time: 3.3E-6, 1.0
```

Figure 5.3: Detailed Output of Minimal Cut Sets

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 5.4.

Use of Analysis Results to Drive Design Change

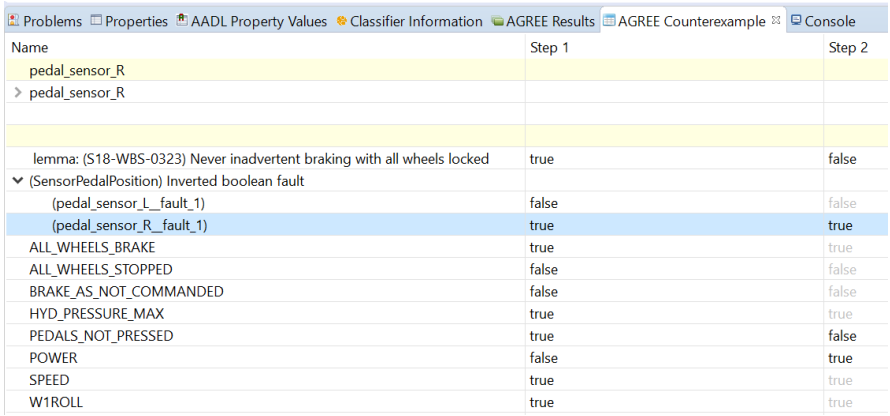
We use a single top level requirement of the WBS: *S18-WBS-0323: Never inadvertent braking with all wheels locked* to illustrate how Safety Annex can be used to detect

```
Minimal Cut Sets for property violation:
property lustre name: wBS_inst___GUARANTEE1
property description:  lemma:
        (S18-WBS-R-0322-left) Asymmetrical left braking.
Total 32 Minimal Cut Sets
Cardinality 1 number: 32

Minimal Cut Sets for property violation:
property lustre name: wBS_inst___GUARANTEE2
property description:  lemma:
        (S18-WBS-R-0322-right) Asymmetrical right braking
Total 32 Minimal Cut Sets
Cardinality 1 number: 32

Minimal Cut Sets for property violation:
property lustre name: wBS_inst___GUARANTEE3
property description:  lemma:
        (S18-WBS-0323) Never inadvertent braking with all wheels locked.
Total 90 Minimal Cut Sets
Cardinality 1 number: 90
```

Figure 5.4: Tally Output of Minimal Cut Sets

design flaws and how faults can affect the behavior of the system.

Upon running max n compositional fault analysis with $n = 1$, a pedal sensor fault was shown to be a single point of failure for the inadvertent braking properties. A



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 5.5: Counterexample for Inadvertent Braking

counterexample is shown in Figure 5.5 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 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 $valid \wedge pedal\_sensor\_value$.

- 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.

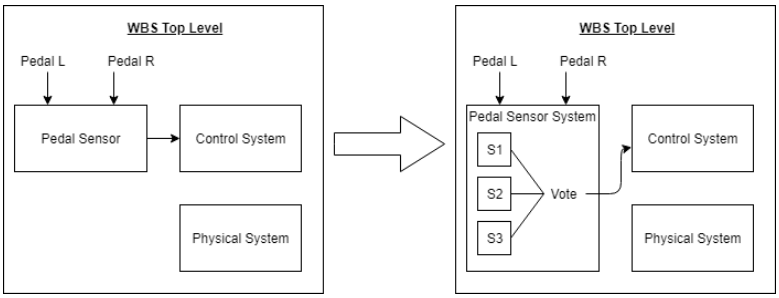


Figure 5.6: Architectural Changes for Fault Mitigation

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 5.6 outlines the 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 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 [142]. This repository also contains all models used in this project.

5.2 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.

5.2.1 The Agreement Protocol Behavioral Implementation

In order to mitigate this problem, special attention must be given to the behavioral model. Using the strategies outlined in previous research [31, 45], 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 5.7.

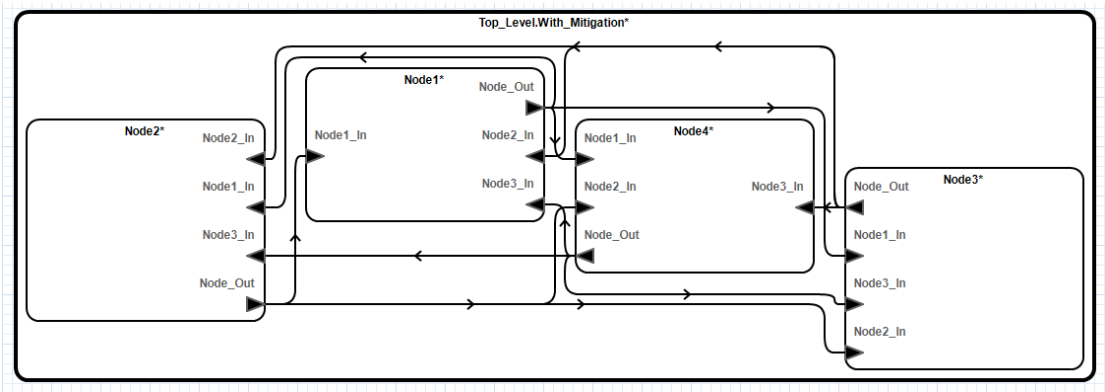


Figure 5.7: Updated PID Example Architecture

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 5.8. These outputs

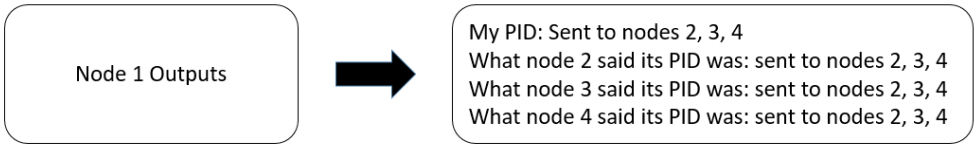


Figure 5.8: Description of the Outputs of Each Node in the PID Example

are modeled as a nested data 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 5.9.

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

```
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 5.9: Data Implementation in AADL for Node Outputs

affected by this fault. This can be accomplished through the fault definition shown in Figure 5.10 and the fault node definition in Figure 5.11.

```
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 5.10: Fault Definition on Node Outputs for PID Example

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

Figure 5.11: Fault Node Definition for PID Example

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

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

These requirements are encoded in AGREE through the use of the following contracts. Figure 5.12 and Figure 5.13 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 5.12: Agreement Protocol Contract in AGREE for No Active Faults

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

Figure 5.13: Agreement Protocol Contract in AGREE Regarding Non-failed Nodes

Referencing Fault Activation Status To fully implement the agreement protocol, it must be possible to describe whether or not a component has 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 5.14.

```
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 5.14: Fault Activation Statement in PID Example

5.2.2 Nominal and Fault Model Analysis

The nominal model verification shows that all properties are valid. Upon running verification in the presence of faults with maximum one active fault, the first four properties stating that all nodes agree on the correct value (Figure 5.12) fail. This is expected since this property is specific to the case when no faults are present in the model. The remaining 4 top level properties (Figure 5.13) 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 5.12). 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 example illustrates the correct handling of such faults and how to utilize the capabilities of the safety annex to model asymmetric failures and test that a system is resilient to such faults.

5.3 Discussion on Timing Results

Given that the safety annex has been developed within the course of this research, there was no pre-existing benchmark models with which to conduct experiments and collect data for comparison with existing tools. As described in Section 2.3, there are a few related research tools that perform similar analyses, e.g., xSAP [16], AltaRica [136], SAML [68], but all of these perform analysis over different modeling languages, use varying analysis methodology, and some even separate fault models from the system modeling language.

Throughout the course of this research, a small number of system models have been developed that illustrate various aspects of modeling and analysis capabilities. These range in size from quite simple two component single layer systems up to the large WBS example outlined in Section 5.1. But notice that the size of the model in terms of the architecture does not completely capture the analysis timing results. For example, a single layer architecture containing two components is small in terms of AADL models, but the nominal AGREE model may contain numerous assumptions and guarantees, and likewise a large number of faults, both of which increase computation time for proofs and for minimal cut set generation compared to a small architectural model with fewer contracts and faults. Likewise, the total number of contracts do not tell the whole story; that number does not give insight into the complexity of the formulas in the contracts.

When discussing results of the fault model analysis, one must not only take into account the number of faults defined in a model, but also the probabilistic threshold, possible fault combinations that are allowed to be active at once, and the complexity of the contracts within the model. Given all of this, it is not so straightforward to run a single analysis and make comparisons along these axes, so we provide enough comparison to glean interesting information regarding the feasibility of the approach. All following analyses were run on an Intel Core i7 with a 2.80GHz CPU and 16 GB RAM.

Nominal Model: As a baseline to the analysis comparisons we provide, we run compositional nominal analysis on the 18 AADL models extended with AGREE contracts. The analysis includes property verification and component consistency checks; this is the analysis that any user would run on a nominal model with no interwoven faults.

Extended model, no faults active: Since a safety analyst extends the nominal model with fault definitions and constraints, we want to see the results of that extension in terms of additional verification time. The model grows in size with each fault that is defined in multiple ways. There is a node definition in the Lustre model corresponding to the fault behavior, there are five variables for each fault added to the model, and each of those variables has constraints written in Lustre. There are also constraints on the new fault related variables that correspond to the fault hypothesis statement. We wish to see how these extensions increases verification time when no faults are active within the model; this will give insight into the feasibility of verifying the extension alone and how costly that extension may be.

Extended model, one fault active: The reason we wish to activate a single fault in the model is to see how removing the constraint that *no* faults are active (as in the extended case above) changes the analysis time. By giving a single active fault constraint, the model checker is required to determine which of the faults may violate a property and thus must iterate through the fault possibilities. If one or more of those faults violate a property, a counterexample is generated which may also add to fault model verification time.

To this end, we performed the above three types of analysis on the 18 models and graphed the results together as shown in Figure 5.15.

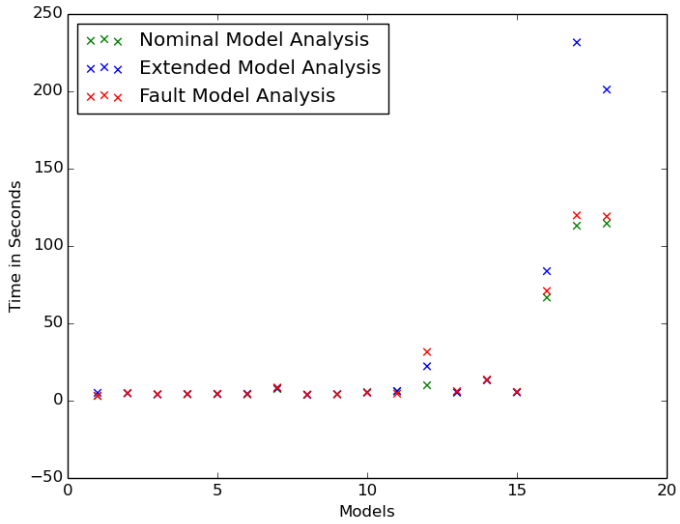


Figure 5.15: Nominal, Extended, and One Fault Verification on 18 Models

The first 15 models are similar in architectural and contractual size and the time differences between the three forms of analysis are negligible. Model 12 shows a greater irregularity between the nominal, extended, and fault analysis times and this is most likely due to the complexity of both the faults and contracts in the model. Models 16-18 show the greatest differences in timing results and are the largest of the set. Interestingly enough, the extended model analysis time was far greater than both the nominal and fault model analysis time. We believe the reason is due to... Why the heck is that happening?? Weird and unexpected results...

Extended model, probabilistic threshold: In the implementation of the safety annex, a preprocessing step to probabilistic analysis is to compute all possible combinations of faults whose probability exceeds the threshold. This is inserted into the Lustre model as an assertion that states only these combinations can be active at once. If there is such a combination that violates a property, it will be shown. Compositional probabilistic analysis can only be performed through the generation of minimal cut sets; therefore, this analysis was first performed using a monolithic probabilistic approach and then a compositional minimal cut set approach to show the comparison. To illustrate how the probabilistic threshold changes analysis results and timing, we perform

this analysis on a larger model: the wheel brake system with 4 wheels. This is a pared down version of what was described in Section 5.1, but still contains over 250 contracts among 66 components with faults defined for all leaf level components of the system.

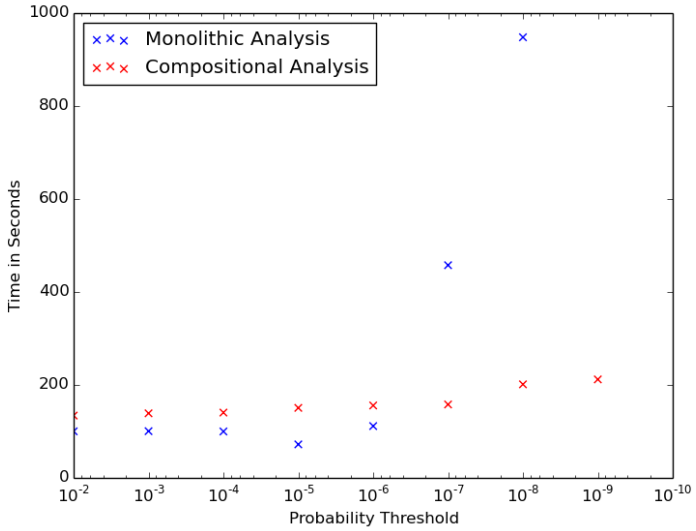


Figure 5.16: Monolithic and Compositional Probabilistic Fault Analysis for the 4 Wheel Braking System

The monolithic analysis shows that the time of analysis increases as the probabilistic threshold is lowered; by 1.0×10^{-7}, the time increase is noticeable. The reason for increased analysis time is that more faults must be considered in the analysis – there are more allowable combinations that could fail together. Theoretically, the time of analysis would increase as the threshold decreased until all possible combinations were always considered in the analysis. At that point, the analysis time would level off. But what we find for monolithic analysis is that when setting the threshold at 1.0×10^{-9}, the verification returns unknown results after 55 minutes. The monolithic approach seems quite suitable for smaller models, but when many faults are introduced by lowering the threshold, the analysis cannot proceed.

When running the same models using a compositional probabilistic approach, we see that the time increases but not nearly as quickly. As expected, a compositional approach may provide a more feasible method of verification for large scale models.

Minimal cut set generation: To analyze the results of the algorithms described in Chapter 4, we split the verification time into two parts; the first part includes only the generation of all MIVCs. This is a required step in the process. The second step is to show the additional time needed to transform those MIVCs into minimal cut sets. There are a number of parameters that can be used to show such results. The computation time may change if a probabilistic threshold is set that is very small, a max *n* analysis threshold that is very large, or a model with many faults defined or numerous contracts. To adjust for these factors, we ran the analysis on the large wheel brake system model (WBS) for each of the safety properties and performed max *n* analysis. The number of minimal cut sets generated per property at a given threshold is shown in Table 5.6.

Property	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0321	7	0	0	256	57,600
0322-R	75	0	0	0	0
0322-L	75	0	0	0	0
0323	182	0	0	0	0
0324	8	3,665	28,694	883,981	-
0325-WX	33	0	0	0	0

Table 5.6: WBS Minimal Cut Set Results for Max *n* Hypothesis

Each column of Table 5.7 labeled with the value of *n* gives the fault hypothesis threshold for that analysis run. For comparison with the number of minimal cut sets generated per property, we refer to Table 5.6[2].

The overall time of the algorithms used for minimal cut set generation is quite small compared to the nominal analysis and extended model analysis time. This is not altogether surprising; the nominal and extended analysis is contending with an infinite state model checking problem whereas the algorithms presented to generate minimal cut sets deal with set element algorithms and boolean formula manipulation. The greatest time recorded was for property S18-WBS-0324 at $n = 4$; the reason is clear when comparing this with Table 5.6: the total number of minimal cut sets computed for this threshold is 916,349.

[2]The property S18-WBS-0325-WX is symmetric for all eight wheels. For readability, we only include wheel one verification timing results in Table 5.7. Likewise for property 0322-L/R.

Property	MIVC Gen	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0321	396.417	5.913	5.468	5.61	5.636	11.925
0322-L	407.078	5.931	5.435	5.302	5.268	5.243
0323	412.926	6.397	6.452	6.420	5.309	5.459
0324	446.610	41.334	41.744	44.062	69.142	-
0325-W1	391.137	5.632	5.388	5.359	5.301	5.236

Table 5.7: WBS Analysis Time in Seconds

In summary, there are multiple ways of comparing analysis timing results and we tried to account for these by providing meaningful comparisons between some important factors. Practically, the method of using compositional verification in safety analysis is a feasible approach in both large and small models. These comparisons support the goal of the research which is to provide usable and automated safety analysis methods to analysts.

Chapter 6

Granularity of Specificationss

The specification of requirements is an important part the development of critical systems, and the analysis results are greatly dependent upon these specifications. Formal specification, as described in Section 2.2.3, is the translation of the informal system requirements into a mathematical logic to determine if the system design is correct [74]. The informal system requirements are often initially developed and written in natural language which is often ambiguous and always mathematically informal. The formal semantics of specified properties, such as LTL, is fully defined and thus is tractable to mathematical reasoning; this does not imply that formalization is fool-proof or straightforward to do [84].

In the approach of system development and safety assessment of models that we describe in this research, we define these specifications in the form of *guarantees* of the behavior of components and *assumptions* regarding the environment. The verification task is to show that a system guarantee P_s is provable given the behavior of its subcomponents C_s and the system assumption A_s. A systems engineer must translate natural language requirements into these formal contracts for each component. If there are dependencies between contracts or irrelevant subexpressions within the statements, this may change the analysis results.

The reason this is of interest to us is due to the use of MIVCs in the generation of minimal cut sets. Previous work has shown that the specification and formulation of the guarantees can affect the IVC analysis results [63]. The algorithm used to generate the cut sets based on MIVC results may lead to the loss of minimality with regard to the cut

set. Assume that there is a single critical guarantee g that is required to prove a safety property. This will be in the MIVC generation and the associated fault activation literal will be in the cut set. Assume the structure of the guarantee is $g = A \wedge B$ for Boolean formulae A and B. There are also two leaf level faults, one that affects the truth value of A and one that affects B. Of course either will violate the guarantee – and hence be in the cut set, but if only A is needed for the proof of the property, there will be more faults in the cut set than minimality suggests.

To address this problem, we begin the exploration of granularity and automated methods to solve this problem. This chapter outlines our exploration into this topic.

6.1 Background Research and Foundation

As described in Chapter 2, a transition relation is considered to be a conjunction of Boolean formulas. 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*. Simply put, coverage is a metric that determines how well properties cover the design of a model. 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 can be seen in the IVCs generated for a given safety property. The intuition can be illustrated with a small example. If the safety property is: $P : A$ for some complex formula A and an equation in the model is $g : A \wedge (B \vee C)$, the supporting equation g will be an IVC. But g also contains other statements that do not necessarily support the proof of P – the only portion of g that matters to the proof is A. 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. A more granular model in this small example could be $g_1 : A$ and $g_2 : B \vee C$. Then we would see only g_1 in the IVC elements for P.

Interestingly, similar work from a varied perspective 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 [61]. An 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 (syntax). 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 [61].

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. The IVC algorithm performs no deeper traces than those defined in those model elements (guarantees/assumptions). For our purposes, it is beneficial at times to see which *parts* of the contract are necessary for the proof. In this case, instead of making the contracts more complex (inlining), we wish to simplify the contracts (un-inlining). In this way, the IVCs are more specific with regard to which parts of the contract are vital to the proof. This will theoretically decompose the contracts and eliminate property dependencies within the model.

Granularity of contracts for IVCs has been briefly discussed by Ghassabani [63], 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 performed a preliminary investigation of granularity within the context of the Lustre language. Lustre provides an adequate formalism for this discussion because it is top-level conjunctive, equational, and *referentially transparent* [69]. 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[1] 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.

6.2 Illustrative Example

To see how different representations of the system requirements will alter the MIVC and minimal cut set 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 indication. It also gives the temperature (pressure) reading seen at the sensor. A diagram showing the two levels of the AADL implementation is shown in Figure 6.1.

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[2]: *temp* = *temp_reading*

These can be seen in the model as two distinct guarantees over the output of the

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

[2] We ignore noise in the temperature reading for simplicity's sake.

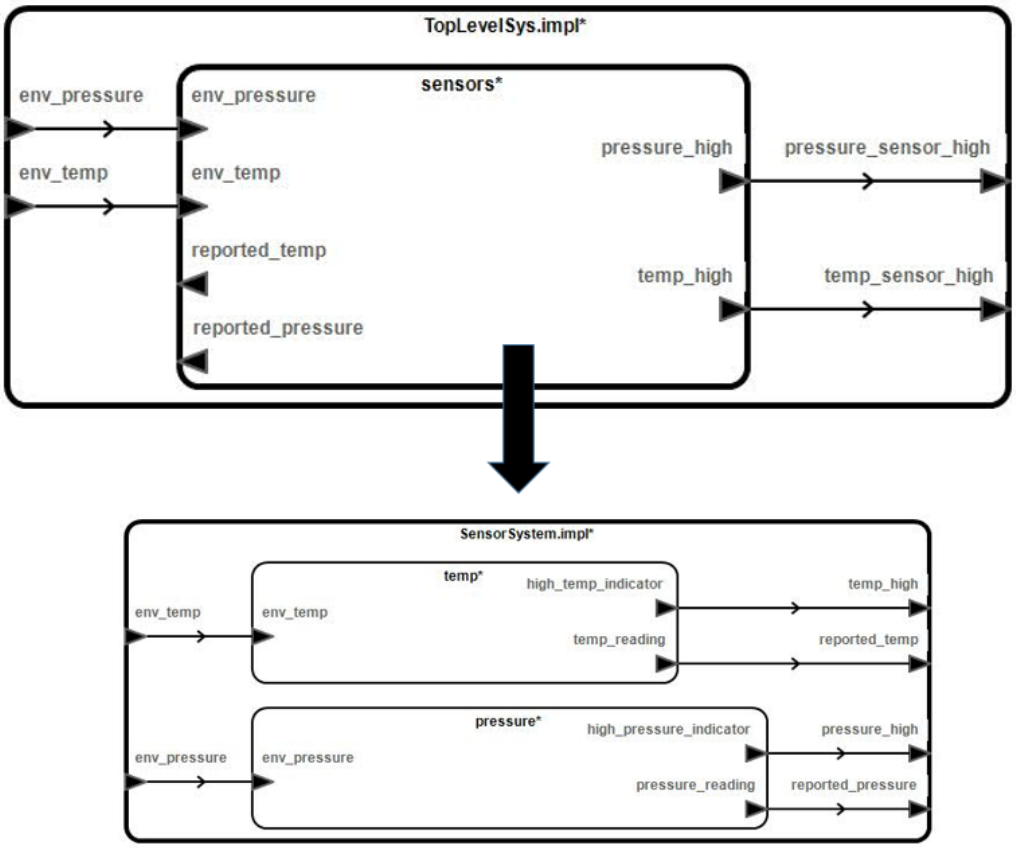


Figure 6.1: Tempurature Sensor System

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 6.2.

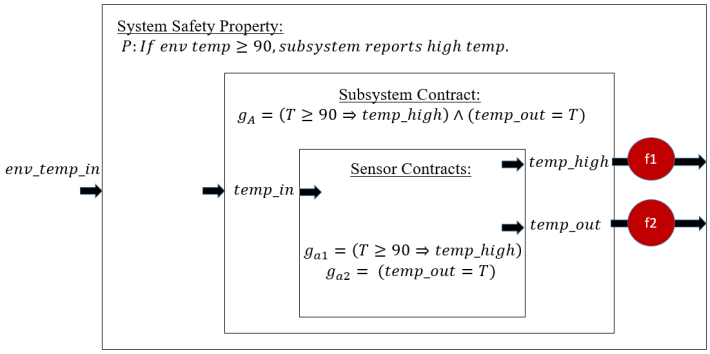


Figure 6.2: Temp Sensor System Contract Part I

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 MIVC.

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 will be found in the minimal cut set for G_A.

Now assume that Figure 6.3 was the system contract representation built by engineer B.

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 either the system safety property nor the subsystem level property, only g_A is found in the *MIVC*s and thus only f_1 will be seen in the minimal cut set 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. Structurally (syntactically), the contracts written

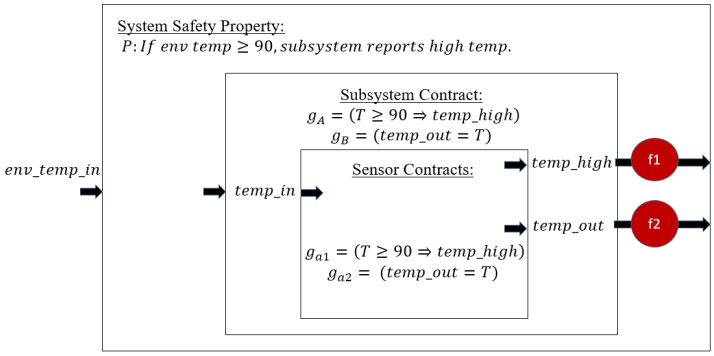


Figure 6.3: Temp Sensor System Contract Part II

by engineer A and engineer B vary; yet logically they are equivalent. The proof of the nominal system holds in both cases. Yet based on the architecture of the model and the formalization of the contracts, analysis results may be different.

In the remainder of this chapter, we attempt to explore this problem through automation of contractual refinement and comparison of analysis results for both inductive validity cores and minimal cut sets.

6.3 Algorithms and Results

The exploration of contractual granularity and its effect on analysis results was performed through the automation of contractual refinement. The algorithms and descriptions of results can be found in Sections 6.3.1 and 6.3.3.

6.3.1 Initial Contractual Refinement

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 formula and create additional guarantees from this refinement. The analysis performed on the model treats all guarantees as a large conjunction. This enables one to automatically split guarantees with expressions containing at the top level a conjunctive operator into two separate expressions. For instance, if $g_0 = A \wedge B$, then split this into $g_1 = A$, and $g_2 = B$ and

insert g_1 and g_2 into the model. This was an initial probing of a deeper issue and gave a preliminary look into our assumption regarding minimal cut set generation in a model similar to that described in Section 6.2. Algorithm 3 shows this algorithm used in this investigation.

Algorithm 3: Split guarantees on logical $\wedge$ operator

```
1 Function splitOnAnd (expression) :
2     Program P ;
3     Guarantees list_G ;
4     for all g ∈ list_G do
5         if binary statement with operator ∧ then
6             insert into P → new guarantee (left) ;
7             insert into P → new guarantee (right) ;
8             splitOnAnd (left) ;
9             splitOnAnd (right) ;
```

The sensor described in Section 6.2 is encoded into Lustre originally had a single guarantee as shown in Figure 6.4 that contained two subexpressions – one of which was unrelated to the proof of the safety property. Given the property `((env_temp >`

```
node temp_sensor(
  env_temp : int;
  high_temp_indicator : bool;
  temp_reading : int
) returns (
  __ASSERT : bool
);
var
  __GUARANTEE0 : bool;
let
  __GUARANTEE0 = (((env_temp > 8) = high_temp_indicator) and (env_temp = temp_reading));
tel;
```

Figure 6.4: Temp Sensor With Original Guarantee

`8) = sensor_high)`, in the first case, `GUARANTEE0` is an MIVC for the safety property. Since both the fault defined on the temperature indicator and the fault defined on the temperature reading output affect `GUARANTEE0`, the minimal cut sets contain both faults.

We then ran JKind on the model using Algorithm 3. The results of the algorithm on the Lustre encoding of the temperature sensor can be seen in Figure 6.5.

```
node temp_sensor(
  env_temp : int;
  high_temp_indicator : bool;
  temp_reading : int
) returns (
  __ASSERT : bool
);
var
  __GUARANTEE1 : bool;
  __GUARANTEE2 : bool;
let
  __GUARANTEE1 = (env_temp = temp_reading);
  __GUARANTEE2 = ((env_temp > 8) = high_temp_indicator);
tel;
```

Figure 6.5: Temp Sensor With Modified Guarantees

In the second case, only `GUARANTEE2` is the IVC for the property. The minimal cut sets produced reflected what we expected and show only the high temperature indicator fault for the analysis in Figure 6.5.

While this algorithm is efficient and quite simple, it only catches the low hanging fruit, so to speak. Due to the logical nature of how the Lustre model is analyzed, all guarantees are viewed as a conjunction; therefore, splitting guarantees into new statements only works on the $\wedge$ operator. This is sufficient for illustration and an initial test into the problem, but cannot provide much decomposition in a general model.

6.3.2 Results of Initial Contractual Refinement

The initial contractual refinement was implemented in the safety annex and performed over the AGREE nominal guarantees. We tested the 18 AADL models used in Section 5.3 and compared the timing results of compositional verification without granular refinement and with this algorithm. The results can be seen in Figure 6.6.

In most cases, the time of analysis is not increased. This is partially due to the size of the model in terms of the number of contracts, but also due to the specifications themselves; if a model does not contain conjunctive contracts, no refinement occurs and

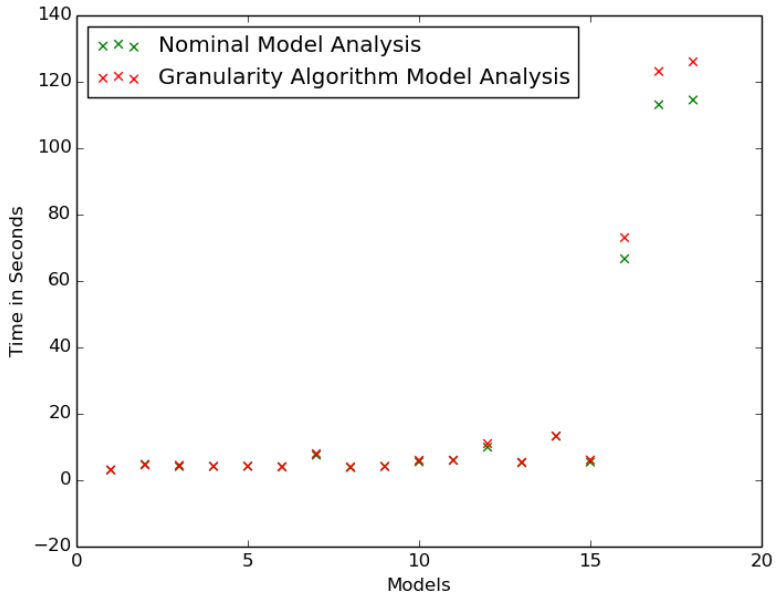


Figure 6.6: Nominal Analysis and Initial Granularity Refinement

the number of contracts in the model remains constant between both forms of analysis. The last three models of the set are all variations on the wheel brake system described in Section 5.1. The complexity of the contracts and number of conjunctions used is much higher than the other models and can be seen in the total analysis time using this refinement approach.

The MIVCs generated using this refinement varied also depending on the model in question. In most cases, there were no differences in MIVC output; the single difference was in the sensor example illustrated in Section 6.2. The results of the refinement provided the expected outcome and showed only the guarantee that directly contributes to the safety property.

Because the results of this algorithm is highly dependent on the model and due to the simplistic nature of this initial refinement, we chose to implement a deeper refinement in JKind and this way could view more closely the results of MIVC generation.

6.3.3 Deeper Granular Refinement

As previously described, we explored granularity within the context of the Lustre language; this programming language provides a good formalism for discussion because it is top-level conjunctive, equational, and *referentially transparent*: the behavior or 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 to a fresh variable which is substituted into the original equation without changing the meaning of a program [63, 69]. In this context, we can define a *granular refinement* as an extraction of a subexpression into a new equation assigning a new variable.

The maximal factorization of the model can be obtained by assigning each instance of a subexpression and each use of an input to its own variable. This results in a *totally decomposed* Lustre model: (1) each computed (non-input) variable is used at most once in the right side of an equation, (2) each equation is either a single operator or a constant expression, and (3) each model input is directly assigned to one or more fresh variables and is not used elsewhere in the model [63].

Ghassabani performed a preliminary analysis on maximally factored models for IVC coverage and found that analysis performed significantly slower for proofs and the `IVC_MUST` algorithm. For our purposes in safety analysis, our concern is the guarantees in the Lustre model; therefore, we are able to weaken the factorization performed. To this end, a *partially decomposed* Lustre model has the properties that (1) each computed variable is used at most once in the right side of an equation, and (2) each guarantee and associated equation has either a single operator or a constant expression.

We were interested in three main research questions regarding the exploration of granularity refinement:

RQ1: Do the MIVCs generated reflect a more accurate view of which portions of the model support the proof of the safety property? We expect that the additional granularity would provide more exact coverage of the model in terms of the MIVCs.

RQ2: What is the analysis time difference between the nominal model MIVC generation and the decomposed model MIVC generation? Given smaller models with fewer or less complex guarantees, the timing results should not differ greatly, but in a large model with potentially many complex guarantees, the computation time for

MIVC generation could increase quickly.

RQ3: Given the results in RQ2, could more exact minimal cut sets be produced from the MIVCs? If changes are reflected in the MIVC sets, then we will see corresponding changes in the minimal cut sets. This could provide more exact safety analysis artifacts and additional insight into the specifications and how they impact the analysis results.

To address these questions, we developed an algorithm that performs a partial decomposition of the Lustre model – a total decomposition of the equations in each node of the Lustre model. For our purposes in safety analysis and minimal cut set generation, the portions of the model that will be of interest are the guarantees (equations) that govern the output of a node and any equations these guarantees may reference; therefore, we perform the refactorization on equations alone.

The logical structure of a formula can be viewed as a tree where nodes are arranged in terms of operator precedence. Clearly, it is a requirement to preserve equivalence during refactoring and we do not want to change the semantics of the formula, only the structure. To this end, we isolated branches and subtrees of the structure so that the `All_IVCs` algorithm can view each subtree of the formula separately. The algorithm we use recursively travels down a formula tree, assigning each subtree as a fresh variable. Figure 6.7 shows the following formula in tree structure where *lit* refers to a literal and *const* refers to a constant:

$$((((const \implies lit) \vee const) \iff lit) \wedge (lit \vee lit))$$

As seen in Figure 6.7, the leaf nodes of the tree are constants and literals; these are the base cases in the recursion. The formula provided as an example has only binary operators; in the algorithm used for refactoring the Lustre equations, unary, binary, and tertiary (if-then-else) operators are handled in a similar fashion.

The refactored equation is shown in Figure 6.8. Each subformula is assigned a fresh variable and added to the equations for the Lustre node. The fresh variables are also assigned to be IVC elements and considered during the `All_IVCs` algorithm.

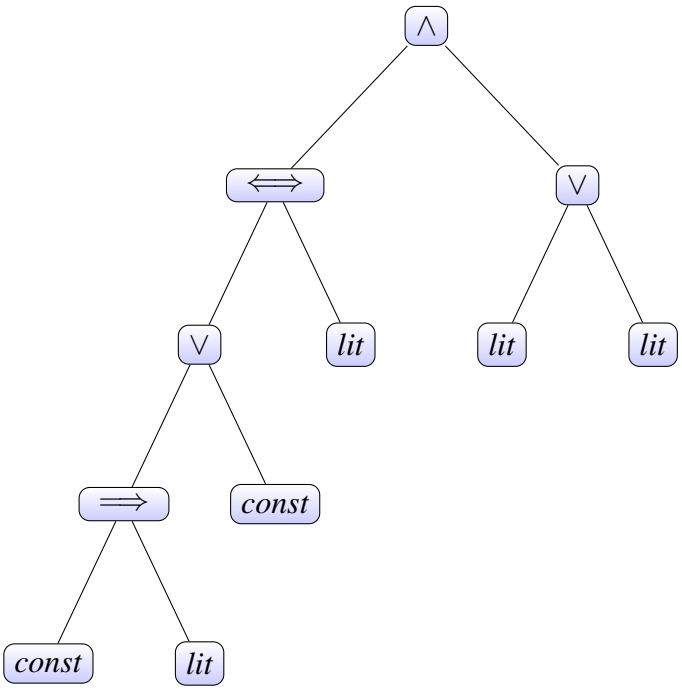


Figure 6.7: Graphical Representation of a Boolean Formula

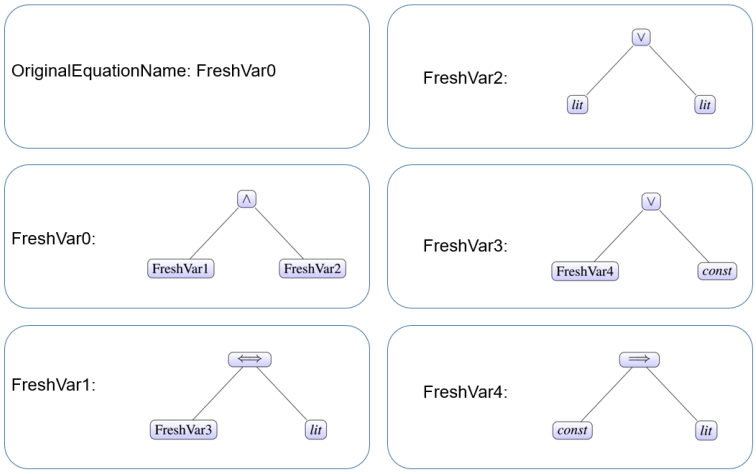


Figure 6.8: Refactored Equation with Fresh Variables

RQ1

To address RQ1, let us present a fictitious monitor component of a system as shown in Figure 6.9. There are two components being monitored for validity (component A and B), each of which sends an error indication to the monitor, `errorCompA` and `errorCompB`. The monitor calculates a `valid` indication if neither error indication is true and when `output` should be sent. There are two conditions in which the monitor is disabled: when a disable command is explicitly sent or when the system is not in auto mode. The property we wish to show is that the monitor does not calculate both a `valid` and `invalid` value simultaneously.

The MIVC generated for this property consists of the equations {`valid`, `invalid`}. Due to the simple nature of this model, it is clear to see that only the error indications from components A and B referenced in equation `valid` are directly used for the proof of the property, and the branch of `valid` contianing reference to `output` is not necessary. Since this equation is not sufficiently granular, the MIVC contains the entire `valid` equation. After performing the refactorization using fresh variables for the model equations, the Lustre code is transformed into what is shown in Figure 6.10.

The equations are broken down into their respective subformulae and refactored such that the equations are totally decomposed and semantically equivalent. While this

```
1  node monitor(
2    errorCompA : bool;
3    errorCompB : bool;
4    disableCmd : bool;
5    autoMode : bool
6  ) returns (
7    OK : bool
8  );
9  var
10   valid : bool;
11   invalid : bool;
12   output : bool;
13 let
14   OK = (not(valid and invalid));
15
16   valid = (not(errorCompA and errorCompB) and output);
17   invalid = (errorCompA and errorCompB);
18   output = (not(disableCmd) and not(autoMode));
19
20   --%PROPERTY OK;
21   --%IVC valid, invalid, output;
22
23 tel;
```

Figure 6.9: The Lustre Model of a Monitor

adds a significant number of new variables and equations to the model – even in this simple example – the MIVCs generated give more information on the subformulae of the equations necessary for proof. The MIVCs of the refactored model are shown in Figure 6.11.

To preserve semantics of the equations during the decomposition, the original equation must be assigned a fresh variable. The MIVC algorithm captures the original equation in `FRESHVAR0`, but also provides a trace down the branch of the equation that is required for the proof. In this case, it traces down the left side of the original binary equation (`valid`) and chooses the fresh variable associated with: `not(errorCompA and errorCompB)`. This correlates to `FRESHVAR1`. Instead of only providing the equation `valid` as an MIVC, this provides a trace through the equation such that the necessary subformula for the proof are found. The third MIVC in this case simply maps back to `invalid`.

The MIVCs now provide information on the necessary subformulae of an equation and not only the equation itself. If all branches are required, then all associated fresh

```
node monitor(
  errorCompA : bool;
  errorCompB : bool;
  disableCmd : bool;
  autoMode : bool
) returns (
  OK : bool
);
var
  valid : bool;
  invalid : bool;
  output : bool;
  FRESHVAR0 : bool;
  FRESHVAR1 : bool;
  FRESHVAR2 : bool;
  FRESHVAR3 : bool;
  FRESHVAR4 : bool;
  FRESHVAR5 : bool;
  FRESHVAR6 : bool;
let
  OK = (not(valid and invalid));

  valid = FRESHVAR0;
  invalid = FRESHVAR3;
  output = FRESHVAR4;
  FRESHVAR0 = (FRESHVAR1 and output);
  FRESHVAR1 = (not FRESHVAR2);
  FRESHVAR2 = (errorCompA and errorCompB);
  FRESHVAR3 = (errorCompA and errorCompB);
  FRESHVAR4 = (FRESHVAR5 and FRESHVAR6);
  FRESHVAR5 = (not disableCmd);
  FRESHVAR6 = (not autoMode);

  --%PROPERTY OK;
  --%IVC FRESHVAR0, FRESHVAR1, FRESHVAR2, FRESHVAR3, FRESHVAR4, FRESHVAR5, FRESHVAR6;

tel;
```

Figure 6.10: The Lustre Model of a Monitor After Refactorization

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
VALID PROPERTIES: [OK] || k-induction || K = 0 || Time = 0.703s
INVARIANTS:
  OK
INDUCTIVE VALIDITY CORE:
  FRESHVAR0
  FRESHVAR1
  FRESHVAR3
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 6.11: The MIVC of the Refactored Monitor Model

variables would be found in those MIVCs.

RQ2

The refactorization described in the previous section introduces multiple new variables and equations into the Lustre model. While a more exact trace of each equation is possible using this method, the size of the model could grow substantially making the time of analysis unacceptable. To this end, we ran a set of Lustre benchmark models used in previous MIVC enumeration work [63] and compared the time difference between the MIVC enumeration of the partially decomposed models and the MIVC enumeration of

the original benchmark models without any decomposition. The time of the MIVC enumeration of decomposed models includes both the decomposition itself and the MIVC enumeration.

In previous work, it was found that all minimal inductive validity core enumeration on this set of benchmarks performed well using the Z3 solver [43, 63] as compared to other solvers available for JKind. In the results presented here, we also used Z3 as a solver and the only IVC elements flagged for consideration were the fresh variables.

RQ3

6.3.4 Discussion

Chapter 7

Discussions and Future Work

The overarching goal of this dissertation is to utilize the capability of a model checker to provide information that can be used during the safety analysis process. We leveraged the model checker to provide behavioral propagation of errors throughout nominal model contracts, we leveraged MIVC generation to provide compositional minimal cut set generation, and we use the counterexample generation capability to gain insight into the state of a system when a property is violated. All of this can be used within the iterative process of complex system design and development to show safety of a system and to drive for design change (see Section 2.1.3). We discuss in this chapter other ways of providing the analyst with safety critical information regarding model elements that may be of interest.

7.1 Graphical Fault Trees

A typical fault tree generated by many of the current research tools available shows very little hierarchical information. An example of a flat fault tree is shown in Figure 7.1 and shows only the minimal cut sets that contribute to the top level event (TLE).

The result of computing the minimal cut sets and presenting them in a tree-like structure was similar to this: a very short tree (one layer) with many branches. In essence, this provides no further information than that of a textual representation of the minimal cut sets. The nature of our minimal cut set generation approach is that the proofs are performed compositionally; this gives information regarding each level of

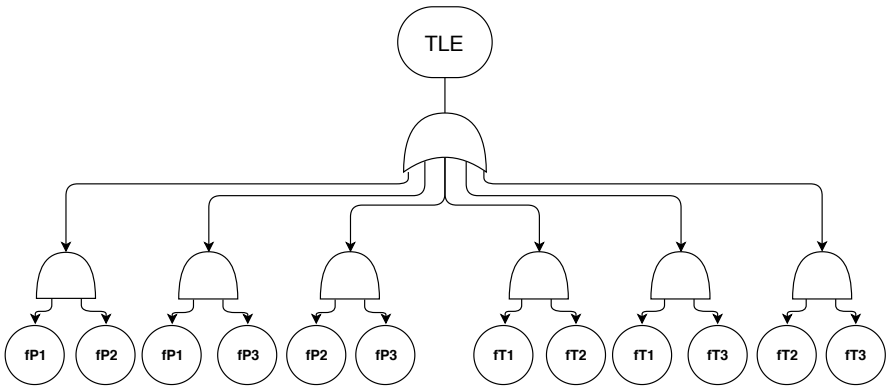


Figure 7.1: Flat Fault Tree for a Sensor Example

analysis *per* each layer of architecture. This information can be used and printed out in a fault tree-like structure and, depending on the model, may provide the links between an active fault and the supporting guarantees it violates. An example of a hierarchical fault tree that could be produced by the information gathered through the minimal cut set algorithms defined in Chapter 4 is shown in Figure 7.2.

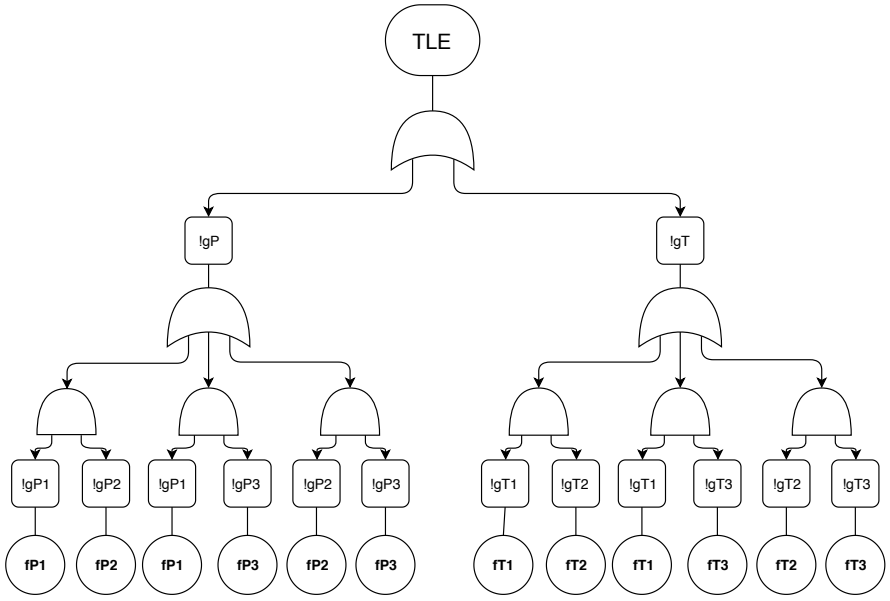


Figure 7.2: Hierarchical Fault Tree for a Sensor Example

Each layer of the architecture is shown in the fault tree and represented by the

violated contracts of that layer. The OR-gate between the lower level contracts and the top level event (violation of the safety property) reflects the property nicely: if either of the subsystems fail, the safety property is violated. The second layer of the tree also reflects the relationship between the sensors and the sensor subsystem: if any pairwise combination of sensors fail, the mid-level contract is violated. This process of using MIVCs to generate minimal cut sets will provide layer by layer the necessary information to not only collect the minimal cut sets, but also reflect the hierarchical nature of the system within the fault trees generated.

Upon initial exploration of the idea, two problems became apparent: (1) this relies on architectural depth of the model and provides no additional information if the architecture is a single layer, and (2) safety analysts wish to see fault trees as a kind of signal flow diagram and require a trace in signal from the error to the property violation. While our initial idea could partially address both issues, we began to look into a different method of analysis that may be able to address both issues.

It became quickly apparent that to include in the analysis results information regarding the roles that faults, components, and node inputs play within a proof would provide great feedback to an analyst. They could view minimal cut sets from the perspective of the entire model and not just the faults that are active. If, for instance, certain components play direct roles in the proof of a safety property, those components would be seen as critical and managed or analyzed in a more comprehensive way. Likewise if specific inputs were known to be crucial to a proof, they would also be critical within the safety analysis. Preliminary investigations showed that related work may be used and extended to perform this kind of analysis [139].

7.2 Compositional Probabilistic Analysis

Safety analysis techniques aim at demonstrating that the system meets the requirements necessary for certification and use in the presence of faults. In many domains, there are two main steps to this process: (1) the generation of all minimal cut sets, and (2) the computation of the corresponding fault probability, i.e., the probability of violating the safety property given probabilities for all faults in the system.

The probability of the Top Level Event (TLE), or violation of the safety property,

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

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

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

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

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

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

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

As previously discussed, it is standard practice to consider cut sets only up to a given cardinality. As the cardinality of the cut sets increase, the likelihood of their occurrence decreases and as the system increases in size, the possible combinations of problematic faults will inevitably increase, at times exponentially. In order to simplify these calculations and address the problem of scalability, minimal cut sets up to a certain cardinality are considered. Everything above that is "safely" ignored, and then specific criteria is

used to over-approximate the error. The end result of these computations is above the actual probability (i.e., a safe approximation), but close enough to be significant.

Another drawback to computing an exact probability of the TLE is the problem of model reliability and exactness; this corresponds exactly to the issue of granularity discussed in Chapter 6. For instance, if two groups of engineers each built a model of the same system, the models may not be equivalent; especially in terms of behavioral properties. Since our approach of computing minimal cut sets depends on the properties over system components, the calculated top-level probability will change. Different representations of the system will alter the computations.

As an example, assume that Figure 7.3 is a snapshot of a given layer in a system designed by the first group of engineers. Component A has a contract, G_A, which is determined by the `All_IVCs` algorithm to depend on a lower level contract, $g_A = a \wedge b$. g_A will be in the set of *IVC*s for the contract G_A. Assume that only a is required for the proof of G_A.

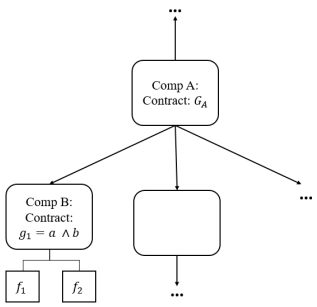


Figure 7.3: Sample System Contract Part I

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

Now assume that Figure 7.4 was the system representation built by the second group of engineers. The basic system structure is the same, but this time there are two contracts for component B: $g_1 = a$ and $g_2 = b$. Since b is not required for the proof of G_A, only g_1 is found in the *IVC*s for G_A and thus only f_1 will be seen in the minimal cut sets for this particular contract.

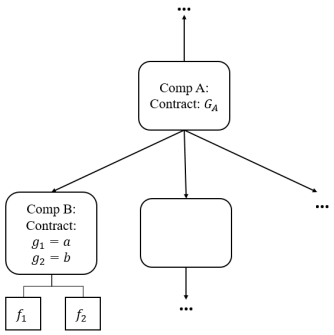


Figure 7.4: Sample System Contract Part II

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

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

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

7.3 Mutations and Equation Removers

For the development of model-based critical systems, it has been argued that formal proof should be applied to gain higher confidence in the model than with testing alone, e.g. [28, 71, 101, 129]. This has proven to be an active area of research, but has also shown some missing pieces – one of which will be of benefit to us in this granularity exploration. When a property is proved valid, no further information is provided about the coverage of the model. One does not know whether the model contains features that are not covered by the properties [144]. Furthermore, we do not know if features *within an IVC element in an MIVC set* are completely covered by the property. To this end, we began to look into mutation coverage to provide an answer.

A mutation approach described by Todorov et al. [144] consists in mutating a model for which safety properties were proved valid, and trying to prove the same properties on the mutated models (*mutants*). If the mutant is proved to be valid (i.e., it *survived*), the mutant reveals part of the model that is not covered by the properties. We know that portion of the model is not necessary to find a proof. The approach described in this section attempts to use this type of mutant analysis on the contracts of a Lustre model, and thus compare to a granular refinement MIVC approach.

A brief review of transition systems and some definitions are provided for convenience.

Given a state space U, a transition system (I, T) consists of an initial state predicate $I : U \to \mathit{bool}$ and a transition step predicate $T : U \times U \to \mathit{bool}$. We define the notion of reachability for (I, T) as the smallest predicate $R : U \to \mathit{bool}$ which satisfies the following formulas:

$$\forall u.\ I(u) \Rightarrow R(u)$$
$$\forall u, u'.\ R(u) \wedge T(u, u') \Rightarrow R(u')$$

A safety property $P : U \to \mathit{bool}$ is a state predicate. A safety property P holds on a transition system (I, T) if it holds on all reachable states, i.e., $\forall u.\ R(u) \Rightarrow P(u)$, written as $R \Rightarrow P$ for short. When this is the case, we write $(I, T) \vdash P$.

The Lustre model is a set of equations$\{eq_1, \ldots, eq_n\}$ and the transition relation T has the structure of being a top level conjunction $T = t_1 \wedge \cdots \wedge t_n$ where each t_i is

an equality corresponding to eq_i. By further abuse of notation, T is identified with the set of its top-level equalities. When an equation is removed from the Lustre model, an equality t_i is removed from T and the transition relation becomes $T \setminus \{t_i\}$.

Definition 3. *Minimal Inductive Validity Core (MIVC) [66]:* $S \subseteq T$ *is a minimal Inductive Validity Core, denoted by* $MIVC(P, S)$*, iff* $IVC(P, S) \wedge \forall T_i \in S.\ (I, S \setminus \{T_i\}) \not\vdash P$.

In this research, we are only interested in minimal sets that satisfy a property P; if $(I, T) \vdash P$, then we know P always has at least one MIVC which is not necessarily unique. By computing all MIVCs, we have a complete mapping from the requirements to the design elements; this is called *complete traceability* [105].

Ghassabani defines two metrics of coverage [65].

Definition 4. MAY-COV *:* $t \in T$ *is covered by* P *iff* $t_i \in$MAY-COV(P)*, where* MAY-COV$(P) = \{t_i | \exists S \in AIVC(P) \cdot t_i \in S\}$.

Definition 5. MUST-COV *:* $t \in T$ *is covered by* P *iff* $t_i \in$MAY-COV(P)*, where* MAY-COV$(P) = \{t_i | \forall S \in AIVC(P) \cdot t_i \in S\}$.

The MAY-COV elements are relevant to the proof, but may be modified without affecting the satisfaction of P, whereas the MUST-COV elements are absolutely necessary for the proof of P. One can view the MUST-COV set of elements as the intersection of all MIVCs; if a single MUST-COV element is removed, it "breaks" all proofs of P.

A mutator is formally a function that mutates any transition predicate T to a set of mutants $\{T^1_{mut}, \ldots, T^m_{mut}\}$, where each mutant T^i_{mut} is obtained by applying a change to T. A very simple mutator is one that simply removes an equality t_i from T, which amounts to removing the corresponding line of code from the Lustre model [144]. Todorov et al. [144] implemented an *equation remover* in JKind which removes equations one by one and replays the proof process in an incremental way. If after removing an equation, the properties are still proved (the mutant survives), it means that the removed equation has no impact on the proof. If the properties do not hold any longer (the mutant is killed), then we know the removed equation is essential for the proof. This mutator computes the minimum MUST-COV core.

7.3.1 Mutations and Guarantees

In the early stages of safety analysis on a complex critical system, it is beneficial to see what model elements may contribute to a property violation. While it is true that analysts will define faults based on their knowledge of the domain, at times in complex systems not all of these faults and their consequences are clear. Using the idea of mutations, we wish to see what the critical inputs to a system may be.

As an example, we look once again at the sensor subsystem of a PWR as outlined in Chapter 4. Given a nominal system model containing the sensor subsystem and a single temperature sensor, we wish to see what model elements – specifically guarantees – are the MUST-COV elements for the program. This tells the analyst that if these guarantees are violated, there are no paths to a proof of the property.

To this end, we modified the equation remover implemented by Todorov, et al. [144] in JKind in order to collect killed guarantees from the program in Lustre. The analysis was run on a version of the sensor system with two subsystem guarantees:

$$(env\_temp > 800) = high\_temp\_indicator$$
$$env\_temp = temp\_reading$$

and one top level property:

$$(env\_temp > 800) = temp\_sensor\_high$$

It is easy to see that a single subsystem level guarantee is sufficient to prove the property. The results from the modified equation remover shows the following guarantee that is critical to any proof of the safety property (Figure 7.5). The location referred to in the figure corresponds with the Lustre program line and column number for user reference.

After finding these results, we then defined a fault on the output governed by this guarantee and ran the minimal cut set generation on the fault model. As expected, this fault was in the minimal cut sets. While this example is sufficiently simple to

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
KILLED: Node input: __GUARANTEE0 = ((env_temp > 800) = temp_high); at location: 30:3
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 7.5: Temperature Subsystem Guarantee Killed by Equation Remover

illustrate the point, in complex models there can be multiple guarantees for a single component, many different components in a subsystem all of which are connected in various ways. It is obvious to anyone looking at the sensor subsystem model that this particular fault will violate the property. To this end, we turned our attention to a larger model: the Wheel Brake System as described in Section **??**. At the time of this analysis, there were 33 fault nodes and 141 fault instances defined for 30 component types and 169 component instances throughout the extended system model. The total number of supporting guarantees within the nominal model was 246. We ran this analysis to see if there were other faults that may have been overlooked during the development of the WBS model.

A guarantee on the hydraulic fuse component of the wheel brake subsystem was presented in a single layer mutation analysis as shown in Figure 7.6.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
KILLED: Node input: __GUARANTEE0 = (true -> (hyd_pressure_in = hyd_pressure_out)); at location: 27:3
KILLED: Node input: __GUARANTEE0 = (true -> (hyd_pressure_in = force_out)); at location: 46:3
KILLED: Node input: __GUARANTEE0 = (true -> (hyd_pressure_in = force_out)); at location: 84:3
KILLED: Node input: __GUARANTEE1 = (true -> ((braking_force > 0) = ((normal_force_in > 0) or
                                            (alternate_force_in > 0)))); at location: 108:3
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 7.6: Hydraulic Fuse Guarantee Killed by Equation Remover

The guarantee presented governs the output of a hydraulic fuse attached to the wheel brake subsystem in the WBS. The guarantee states that the hydraulic pressure in is equal to the output. A stuck closed fault was defined on this fuse and minimal cut sets were generated for the WBS with cardinality restriction at one. Since this guarantee is a MUST-COV element of the program, it should be the case that the fault is a single point of failure. The expected results are seen in the minimal cut sets as shown in Figure 7.7; the violation of the property occurs when this fault is present.

Performing a kind of mutation analysis could be beneficial during the fault modeling process to catch things that may be missed during the fault definition process. The

```
Minimal Cut Sets for property violation:
property lustre name: wBS_inst___GUARANTEE8
property description:  lemma: (S18-WBS-0325) Never inadvertent braking of wheel 4
Total 19 Minimal Cut Sets
Minimal Cut Set # 1
Cardinality 1
original fault name, description: HydraulicFuse_FailedClosed, "(HydraulicFuse) Stuck closed fault."
lustre component, fault name: wheel_brake4,
    wheel_brake4_fault__independently__active__normal_hyd_fuse__normal_hyd_fuse__fault_1
failure rate, default exposure time: 1.0E-5, 1.0
```

Figure 7.7: Hydraulic Fuse Fault in Minimal Cut Sets

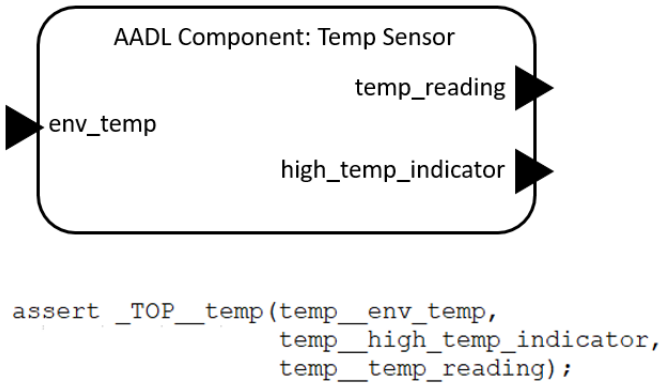
initial foray into mutation testing results show that this has potential for integration into the safety annex. It was also clear that a better presentation of the results was needed for large models. In the case of the WBS mutation analysis, over 240 guarantees were given as candidates for faults – many of which already had faults associated with them. By integrating this feature into the fault analysis, these guarantees could be pruned from the output and save the user from pouring through potentially hundreds of guarantees as well as improve mutation engine time by eliminating the fault-associated guarantees from the analysis. Likewise, in the safety annex it is possible to generate the Lustre program if desired, but the common user will not reference the location of a guarantee this way. The location feature would need to be integrated into the safety annex and provide a link to the guarantee within the component AADL file.

7.3.2 Node Inputs and Mutation Testing

The equation remover mutation also iterates through Lustre node inputs and performs the mutation one input at a time. The node inputs in the Lustre model correspond to the inputs and outputs of an AADL component as shown in Figure 7.8.

Given that the equation remover algorithm can perform this operation over node inputs, we can also gather information about critical outputs of components. Performing a similar extention to the equation remover algorithm, we collected all node input mutations that are killed and present them to the user. As an example, we show in Figure 7.9 the analysis results on the nominal temperature subsystem example.

Given that there exists an assumption on the temperature sensor input, we focus on the output. The only killed output was that of the high temperature indication, which tells us to prove the guarantee $env\_temp > 8 \iff temp\_high$, the output of this node




```
assert _TOP__temp(temp__env_temp,
                  temp__high_temp_indicator,
                  temp__temp_reading);
```

Figure 7.8: An AADL Component and Lustre Node Inputs

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
KILLED: Node input: temp__env_temp at location: 67:60
KILLED: Node input: temp__high_temp_indicator at location: 67:88
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 7.9: Temperature Node Inputs Killed by Equation Remover

is of utmost importance. Not surprisingly, when attaching a fault to this output, we get this fault in the minimal cut set for the top level property.

As in the case with application of mutation testing on guarantees, node inputs also will require integration into the safety annex in order to filter out node inputs that have already been accounted for with faults in the extended system model. In large systems, this analysis is scalable and shows itself to be informative, but the outputs are unwieldy and large. Further work to rectify this would be required before use in large systems.

The investigations into mutation testing applied to fault analysis were implemented in JKind and can be found at `https://github.com/dkstewart/jkind` on the *fault_analysis_mutations* branch.

7.4 Preprocessing Improvements

The preprocessing that must be done to generate minimal cut sets is not trivial. All minimal inductive validity cores must be produced which is as hard as model checking [64]. If all MIVCs are found, then these sets must be transformed into minimal correction sets through a minimal hitting set algorithm. Finding the hitting sets, or *set cover* is an NP-Complete problem itself in terms of cardinality alone [60, 104].We performed extensive research into hitting set algorithms and implemented an open source option that performed well on benchmark testing [], but nevertheless, these preprocessing steps could be fraught with peril when models become quite large and complex. We believe that a more direct approach to computing the minimal correction sets (MCSs) could be beneficial and is possible to implement as an additional JKind engine. This would eliminate the need to compute MIVCs and avoid the hitting set algorithm altogether.

Liffiton et al. [90] considered the enumeration of MCSs and MUSs as an exploration of the power set of all subclauses in a formula. The subsets form a lattice through subset relations and this lattice can be explored in clever ways in order to enumerate these related sets. A `map` is a boolean function used to encode the explored portions of the lattice. The entire lattice can be split into two regions based on the feasibility of the subset: these are the *maximally satisfiable subsets* (MSS) and the *minimally unsatisfiable subsets* (MUS). A maximally satisfiable subset is the complement of a MCS with respect to the constraint system; more formally, an MSS M of a constraint system C is a subset $M \subseteq C$ such that M is satisfiable and $\forall c \in C \setminus M : M \cup \{c\}$ is unsatisfiable. A full enumeration of the MSSs (also called the Max-SAT problem) will easily provide the full enumeration of the MCSs by taking the complement in C.

The lattice structure ensures that any subset of a maximally satisfiable set is still satisfiable; thus, as subsets are explored, large portions of the map are eliminated from consideration. The CAMUS algorithm presented by Liffiton et al. [91] uses the hitting set duality between MCS and MUS to produce MUSs by first computing MSSs, finding the complements (MCSs), and then producing the hitting sets (MUSs). It computes MCSs with what can be considered a top-down search through the power set, searching a level (subsets of a particular size) for satisfiable subsets that are not subsumed by some larger satisfiable subset found in a higher level. Whenever an MSS is found, all subsets are blocked from the `map` and the search moves on to the next level of the

power set. The algorithms presented for this SAT-solver based enumeration of MUSs are (1) find all MCSs, and (2) compute MUSs. Related research has recently expanded upon the former algorithm used to enumerate MCSs by employing a correction set rotation method and simultaneously implementing the strengthening and relaxing methods described in the CAMUS algorithm [106].

We believe that these algorithms can be adjusted for SMT solvers and infinite state transition systems appropriately and then implemented in JKind. This will allow for a direct computation of MCSs and avoid the two preprocessing steps (MIVC generation and MHS generation) as outlined in this dissertation.

Chapter 8

Conclusion

System safety analysis is crucial in the development of critical systems and the generation of accurate and useful 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 development process. A model-based approach was proposed that allows for a tighter integration between system development and safety analysis. The existing system model is extended and safety specific definitions and information can be defined at all levels of the model architecture (e.g., software, hardware, system level, module level). This extended model, or fault model, can be analyzed using a model checker and various safety related artifacts can be generated. These include snapshots of the state of the system when a fault is active (automatically generated counterexamples), maximum active fault thresholds, and minimal cut sets.

We also provided a formalization of the compositional generation of minimal cut sets through the use of inductive validity cores. This will open the door for future research work into more scalable options of minimal cut set generation through the use of SMT-solvers and possibly other verification engines. An introductory exploration into the concepts of granularity and mutation testing provides a framework for the application of other kinds of formal methods techniques on a fault model and what kinds of important information can be gleaned from such analyses.

There are multiple ways to view an extended system model and many ways to incorporate model checking theory into the information provided to a safety analyst; future

work in this area holds promise. The key to this is clear communication with those in the practical field of safety analysis. It has been our goal to provide to safety analysts what they need and will use. Each step of this research has been discussed with analysts working in the aerospace domain and adjusted according to requirements of the field and needs of the analysts. We believe this is the key to moving from a research oriented methodology into the realm of applicable results in a critical system domain.

All of this contributes to the long range goal of the research: to increase system safety through the support of MBSA process backed by formal methods to help safety engineers with early detection of design issues and automation of the artifacts required for certification.

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] P. A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing safe, reliable systems using scade. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 115–129. Springer, 2004.

[4] AIR 6110. Contiguous aircraft/system development process example, Dec. 2011.

[5] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[6] AS5506C. Architecture analysis & design language (AADL), Jan. 2017.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] H. Boudali, P. Crouzen, and M. Stoelinga. Dynamic fault tree analysis using input/output interactive markov chains. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 708–717. IEEE, 2007.

[18] 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.

[19] M. Bozzano, A. Cimatti, A. Griggio, and C. Mattarei. Efficient anytime techniques for model-based safety analysis. In *Computer Aided Verification*, 2015.

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

[26] 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.

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

[28] 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.

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

[30] 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.

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

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

[33] 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.

[34] 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.

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

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

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.

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

[43] 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.

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

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

[46] 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.

[47] 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.

[48] N. Eén and N. Sörensson. Temporal induction by incremental sat solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

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

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

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

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

[53] 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.

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

[55] M. Fowler and U. Distilled. A brief guide to the standard object modeling language, 2003.

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

[57] S. Friedenthal, A. Moore, and R. Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

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

[59] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The jkind model checker. *CAV 2018*, 10982, 2018.

[60] 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.

[61] 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.

[62] 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.

[63] E. Ghassabani. *Inductive validity cores*. PhD thesis, University of Minnesota, 2018.

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

[65] E. Ghassabani, A. Gacek, M. W. Whalen, M. P. Heimdahl, and L. Wagner. Proof-based coverage metrics for formal verification. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 194–199. IEEE, 2017.

[66] 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.

[67] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):843–871, 1994.

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

[69] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *IEEE*, volume 79(9), pages 1305–1320, 1991.

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

[71] D. Hardin, T. D. Hiratzka, D. R. Johnson, L. Wagner, and M. Whalen. Development of security software: A high assurance methodology. In *International Conference on Formal Engineering Methods*, pages 266–285. Springer, 2009.

[72] 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.

[73] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE transactions on Software Engineering*, 22(6):363–377, 1996.

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

[75] 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.

[76] P. Hönig, R. Lunde, and F. Holzapfel. Model based safety analysis with smartIflow. *Information*, 8(1), 2017.

[77] 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.

[78] A. Joshi. *Behavioral fault modeling and model composition for model-based safety analysis*. PhD thesis, University of Minnesota, 2008.

[79] 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.

[80] 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.

[81] 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.

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

[83] 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.

[84] G. Kotonya and I. Sommerville. *Requirements engineering: processes and techniques*. Wiley Publishing, 1998.

[85] 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.

[86] A. v. Lamsweerde. Formal specification: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 147–159, 2000.

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

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

[89] M. H. Liffiton, Z. Andraus, and K. Sakallah. From Max-SAT to Min-UNSAT: insights and applications. *Ann Arbor*, 1001:48109–2122, 2005.

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

[91] M. H. Liffiton and K. A. Sakallah. On finding all minimally unsatisfiable subformulas. In *International conference on theory and applications of satisfiability testing*, pages 173–186. Springer, 2005.

[92] 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.

[93] O. Lisagor. *Failure logic modelling: a pragmatic approach*. PhD thesis, University of York, 2010.

[94] 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.

[95] 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*, pages 625–632. IEEE, 2011.

[96] 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.

[97] MathWorks. The MathWorks Inc. Simulink Product Web Site. http://www.mathworks.com/products/simulink, 2004.

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

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

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

[101] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Communications of the ACM*, 53(2):58–64, 2010.

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

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

[104] 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.

[105] A. Murugesan, M. W. Whalen, E. Ghassabani, and M. P. Heimdahl. Complete traceability for requirements in satisfaction arguments. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 359–364. IEEE, 2016.

[106] N. Narodytska, N. Bjørner, M.-C. V. Marinescu, and M. Sagiv. Core-guided minimal correction set and core enumeration. In *IJCAI*, pages 1353–1361, 2018.

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

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

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

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

[111] 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.

[112] 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.

[113] 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.

[114] 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.

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

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

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

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

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

[120] 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.

[121] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings. Eighth Annual IEEE International Conference and Workshop On the Engineering of Computer-Based Systems-ECBS 2001*, pages 83–91. IEEE, 2001.

[122] S. Rayadurgam, A. Joshi, and M. P. Heimdahl. Using PVS to prove properties of systems modelled in a synchronous dataflow language. In *International conference on formal engineering methods*, pages 167–186. Springer, 2003.

[123] 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.

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

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

[126] H. E. Roland and B. Moriarty. *System safety engineering and management*. John Wiley & Sons, 1990.

[127] 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.

[128] 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.

[129] J. Rushby. Software verification and system assurance. In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 3–10. IEEE, 2009.

[130] SAE Aerospace. SAE AS5506B: Architecture analysis & design language (AADL) standard document. *SAE International*, 2012.

[131] SAE ARP 4761. Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment, December 1996.

[132] SAE ARP4754A. Guidelines for development of civil aircraft and systems, December 2010.

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

[134] B. Schätz, A. Pretschner, F. Huber, and J. Philipps. Model-based development of embedded systems. In *International Conference on Object-Oriented Information Systems*, pages 298–311. Springer, 2002.

[135] 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.

[136] 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.

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

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

[139] K. Siu, A. Moitra, M. Li, M. Durling, H. Herencia-Zapana, J. Interrante, B. Meng, C. Tinelli, O. Chowdhury, D. Larraz, M. Yahyazadeh, M. F. Arif, and D. Prince. Architectural and behavioral analysis for cyber security. In *2019 IEEE/AIAA 38th Digital Avionics Systems Conference (DASC)*, pages 1–10, 2019.

[140] 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.

[141] 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.

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

[143] D. Stewart, M. Whalen, D. Cofer, and M. P. Heimdahl. Architectural modeling and analysis for safety engineering. In *IMBSA 2017*, pages 97–111, 2017.

[144] V. Todorov, S. Taha, and F. Boulanger. Specification quality metrics based on mutation and inductive incremental model checking. In *Proceedings of the 10th NASA Formal Methods Symposium (NFM 2020)*, May 2020.

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

[146] W. Vesely and R. Narum. Prep and kitt: Computer codes for the automatic evaluation of a fault tree. Technical report, Idaho Nuclear Corp., Idaho Falls, 1970.