Architectural Modeling and Analysis for Safety Engineering

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

Danielle Stewart

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

Mats P. E. Heimdahl

May, 2019

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.

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

Contents

Chapter 1

Introduction

System safety analysis is crucial in the development life cycle of critical systems to ensure adequate safety as well as demonstrate compliance with applicable standards. A prerequisite for any safety analysis is a thorough understanding of the system architecture and the behavior of its components; safety engineers use this understanding to explore the system behavior to ensure safe operation, assess the effect of failures on the overall safety objectives, and construct the accompanying safety analysis artifacts [49, 50]. Developing adequate understanding, especially for software components, is a difficult and time consuming endeavor which has traditionally been performed manually. Given the increase in model-based development in critical systems [11, 29, 32, 36, 40], leveraging the resultant models in the safety analysis process, as well as automating the generation of safety analysis artifacts, holds great promise in terms of analysis accuracy as well as efficiency. Although some research groups have introduced automating the safety assessment process and have developed tools to support this [11, 15, 34], there is a lack that we wish to address:

- A fault model that extends the system model and is not separate.
- The ability to model behavioral propagation as well as explicit propagation of faults within this fault model.
- Compositional generation of required artifacts in the safety assessment process.
- Compositional probabilistic computations regarding the violation of the top-level safety properties (top-level events).

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

Our long range goal is to support a model-based safety assessment process backed by formal methods to help safety engineers with early detection of design issues and automation of the required artifacts used in certification. **The objective of this proposal**, which is a logical step towards our goal, is to define a safety analysis tool that allows for flexible fault modeling and fits into the MBSA approach, and furthermore to formally prove a relationship between verification results and safety analysis artifacts.

There are two main pieces of this research work. The first is the research and development of a safety analysis tool that works closely with existing verification engines. The second is to define algorithms and formulate theoretical proofs that make use of verification results in order to extract information about the fault model.

We plan to accomplish the objectives of this proposal by pursuing the following aims:

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

2. **Define compositional verification capabilities in the face of component failures.** Determine capabilities for activating a certain number of faults or a probabilistic threshold for all active faults and perform compositional verification within these thresholds.

3. **Define compositional computation of minimal cut sets.** Formally prove the relationship between verification results of compositional analysis and minimal cut sets, and implement the formal algorithm in the Safety Annex.

4. **Determine accurate compositional probabilistic computations for the top-level events.** Define what these computations would be, prove that the computations are accurate, and implement these algorithms in the Safety Annex.

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

1.1 Chapters and Organization of the Proposal

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

Chapter 2

Background

2.1 The Safety Assessment Process and Required Artifacts

2.1.1 Concise Overview of the Process and It's Artifacts

In order to approve an aircraft for flight through the FAA in the USA, one must meet federal regulations. Industry has adopted the standards published by SAE including Aerospace Information Reports (AIR) and Aerospace Recommended Practices (ARP). These standards aim to provide an organized and acceptable approach for safety analysis that serves to meet the federal regulations and provides industry with a methodology for safety assessment [1,2]. During the safety assessment process, certain artifacts showing behavior and functionality of the system in the presence of faults are used. Often, these artifacts are necessary for certification of the system Is there a citation from anywhere that I can use here?. There are a handful of important artifacts, but one that is important to this research is a fault tree.

Fault Tree Analysis

The use of fault trees are common in many safety assessment processes and the ability to generate the cut sets needed for the construction of the fault tree is a useful part of any safety analysis tool. The fault tree is a safety artifact commonly referenced in requirement protocol documents such as ARP4761, ARP4754, and AIR6110 [3,49,50].

A Fault Tree (FT) is a directed acyclic graph whose leaves model component failures and whose gates model failure propagation [48]. The system failure under examination is the root of the tree and is called the Top Level Event (TLE). The node types in a fault tree are *events* and *gates*. An event is an occurrence within the system, typically the failure of a subsystem down to an individual component. Events can be grouped into Basic Events (BEs), which occur independently, and *intermediate events* which occur dependently and are caused by one or more other events [22]. These events model the failure of the system (or subsystem) under consideration. The gates represent how failures propagate through the system and how failures in subsystems can cause system wide failures. The two main logic symbols used are the Boolean logic AND-gates and OR-gates. An AND-gate is used when the undesired top level event can only occur when all the lower conditions are true. The OR-gate is used when the undesired event can occur if any one or more of the next lower conditions is true. This is not a comprehensive list of gate types, but we focus our attention on these two common gate types.

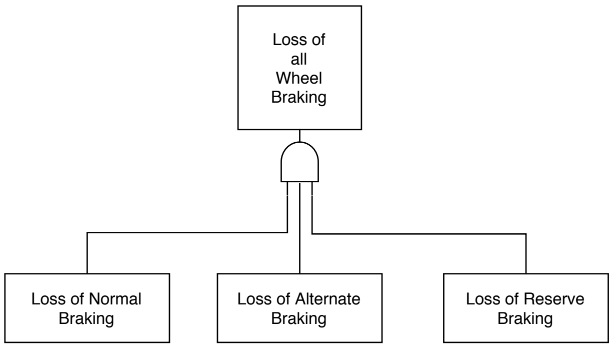


Figure 2.1: A simple fault tree

Figure 2.1 shows a simple example of a fault tree based on SAE ARP4761 [49]. In this example, the top level event corresponds to an aircraft losing all wheel braking. In order for this event to occur, all of the basic events must occur. This is seen through the use of the AND gate below the top level event. The gates in the fault tree describe how failures propagate through the system. Each gate has one output and one or more inputs. In Figure 2.1, the AND gate has three inputs and one output. The leaves of the tree represent the basic events of the system. In the case of this fault tree, these three events are also the Minimal Cut Sets (MinCutSets) for this top level event. A MinCutSet is the minimal set of basic events that must occur together in order to cause the TLE to occur. Generating and analyzing these MinCutSets is important to FTA and has been an active area of interest in the research community since fault trees were first described in Bell Labs in 1961 [22, 48].

There are two main types of fault tree analysis that we differentiate here as *qualitative* analysis and *quantitative* analysis. In qualitative analysis, the structure of the fault tree is considered and the MinCutSets are a way to indicate which combinations of component failures will cause the system to fail. On the other hand, in quantitative analysis the probability of the TLE is calculated given the probability of occurrence of the basic events. By being able to generate MinCutSets based on both cardinality and probability, this allows for either form of FTA to be created.

2.1.2 The Traditional Safety Assessment Process

The traditional safety assessment process at the system level is based on the standards ARP4754A [50] and ARP4761 [49]. It starts with the System level Functional Hazard Assessment (SFHA) examining the functions of the system to identify potential functional failures and classifies the potential hazards associated with them.

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

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

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

2.1.3 Model-Based Safety Assessment Process Supported by Formal Methods

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

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

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

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

4. Safety engineers use the backend model checker to analyze if the safety requirements and fault tolerance objectives are satisfied by the design in the presence of faults. If the design does not tolerate the specified number of faults (or probability threshold of fault occurrence), then the tool produces counterexamples leading to safety requirement violation in the presence of faults, as well as all minimal set of fault combinations that can cause the safety requirement to be violated.

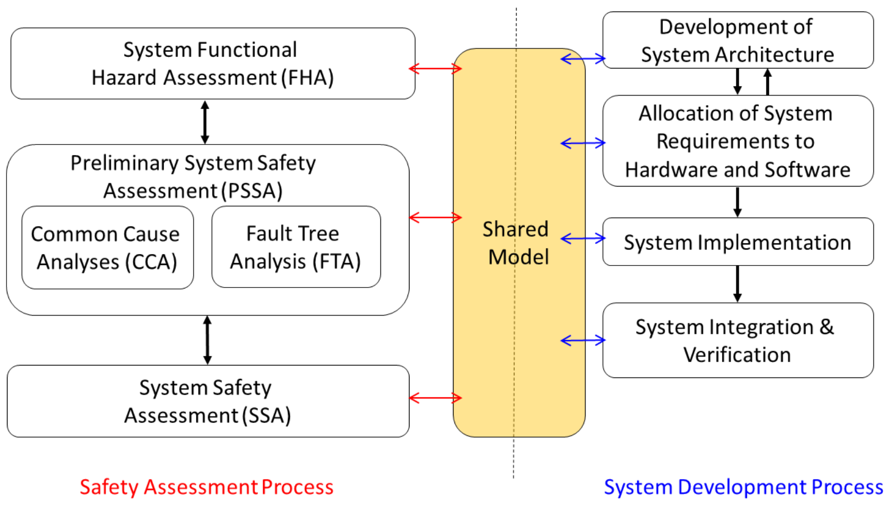

System Functional Hazard Assessment (FHA)
Preliminary System Safety Assessment (PSSA)
Common Cause Analyses (CCA)
Fault Tree Analysis (FTA)
System Safety Assessment (SSA)
Shared Model
Development of System Architecture
Allocation of System Requirements to Hardware and Software
System Implementation
System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

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

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

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

2.2 Related Work

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

Safety Analysis Tools and Research: A model-based approach for safety analysis was proposed by Joshi et. al in [34–36]. In this approach, a safety analysis system model (SASM) is the central artifact in the safety analysis process, and traditional safety analysis artifacts, such as fault trees, are automatically generated by tools that analyze the SASM.

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

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

Tools such as the AADL Error Model Annex, Version 2 (EMV2) [24] and HiP-HOPS for EAST-ADL [18] are *FLM*-based *ESM* approaches. As previously discussed, given many possible faults, these propagation relationships require substantial user effort and become more complex. In addition, it becomes the analyst's responsibility to determine whether faults can propagate; missing propagations lead to unsound analyses. In the Safety Annex, propagations occur through system behaviors (defined by the nominal contracts) with no additional user effort.

Closely related to our work is the model-based safety assessment toolset called COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [12]. COMPASS is a mixed *FLM/FEM*-based, *causal compositional* tool suite that uses the SLIM language, which is based on a subset of AADL, for its input models [13, 16]. 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 [19, 43], MRMC (Markov Reward Model Checker) [37, 41], and RAT (Requirements Analysis Tool) [46]. The safety analysis tool xSAP [9] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [10]. COMPASS is an impressive tool suite, but some of the features that make AADL suitable for SW/HW architecture specification: event and event-data ports, threads, and processes, appear to be missing, which means that the SLIM language may not be suitable as a general system design notation (ESM).

SmartIFlow [33] is a *FEM*-based, *purpose-built*, *monolithic non-causal* safety analysis tool that describes components and their interactions using finite state machines and events. Verification is done through an explicit state model checker which returns sets of counterexamples for safety requirements in the presence of failures. SmartIFlow allows *non-causal* models containing simultaneous (in time) bi-directional error propagations. On the other hand, the tools do not yet appear to scale to industrial-sized problems, as mentioned by the authors [33]: "As current experience is based on models with limited size, there is still a long way to go to make this approach ready for application in an industrial context".

The Safety Analysis and Modeling Language (SAML) [29] is a *FEM*-based, *purpose-built*, *monolithic causal* safety analysis language. System models constructed in SAML can be used used for both qualitative and quantitative analyses. It allows for the combination of discrete probability distributions and non-determinism. The SAML model can be automatically imported into several analysis tools like NuSMV [19], PRISM (Probabilistic Symbolic Model Checker) [38], or the MRMC probabilistic model checker [37].

AltaRica [8, 45] is a *FEM*-based, *purpose-built*, *monolithic* safety analysis language with several dialects. There is one dialect of AltaRica which use dataflow (*causal*) semantics, while the most recent language update (AltaRica 3.0) uses non-causal semantics. The dataflow dialect has substantial tool support, including the commercial Cecilia OCAS tool from Dassault [7]. For this dialect the Safety assessment, fault tree generation, and functional verification can be performed with the aid of NuSMV model checking [14]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [6]. AltaRica 3.0 has support for simulation and Markov model generation through the OpenAltaRica (www.openaltarica.fr) tool suite.

Formal verification tools based on model checking have been used to automate the generation of safety artifacts [9, 14, 17]. This approach has limitations in terms of scalability and readability of the fault trees generated. Work has been done towards mitigating these limitations by the scalable generation of readable fault trees [15].

Minimal Cut Set Generation and Probabilistic Evaluations:

2.3 Tools and Modeling Language

2.3.1 Architecture Analysis and Design Language

We are using the Architectural Analysis and Design Language (AADL) to construct system architecture models. AADL is an SAE International standard that defines a language and provides a unifying framework for describing the system architecture for "performance-critical, embedded, real-time systems" [4, 23]. From its conception, AADL has been designed for the design and construction of avionics systems. Rather than being merely descriptive, AADL models can be made specific enough to support system-level code generation. Thus, results from analyses conducted, including the new safety analysis proposed here, correspond to the system that will be built from the model.

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

2.3.2 Assume Guarantee Reasoning Environment

The Assume Guarantee Reasoning Environment (AGREE) is a tool for formal analysis of behaviors in AADL models [20]. It is implemented as an AADL annex and annotates AADL components with formal behavioral contracts. Each component's contracts can include assumptions and guarantees about the component's inputs and outputs respectively, as well as predicates describing how the state of the component evolves over time.

AGREE translates an AADL model and the behavioral contracts into Lustre [30] and then queries a user-selected model checker to conduct the back-end analysis. The analysis can be performed compositionally or monolithically.

Monolithic vs. Compositional Analysis: Compositional analysis of systems was introduced in order to address the scalability of model checking large software systems [20, 31, 44]. Monolithic verification and compositional verification are two ways that mathematical verification of component properties can be performed. In monolithic analysis, the model is flattened and the top level properties are proved using only the leaf level contracts of the components. The analysis can alternatively be performed compositionally following the architecture hierarchy such that analysis at a higher level is based on the components at the next lower level. 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 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 components of a system are organized hierarchically and each layer of the architecture is viewed a system. For any given layer, the proof consists of demonstrating that the system guarantee is provable given the guarantees of its direct subcomponents and the system assumptions. This proof is performed one layer at a time starting from the top level of the system. When compared to monolithic analysis (i.e., analysis of the flattened model composed of all components), the compositional approach allows the analysis to scale to much larger systems [20].

2.3.3 Safety Annex for AADL

The Safety Annex for AADL is a tool that provides the ability to reason about faults and faulty component behaviors in AADL models and has been developed throughout the course of this project [51–54]. In the Safety Annex approach, formal assume-guarantee contracts are used to define the nominal behavior of system components. The nominal model is verified using AGREE. The Safety Annex 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 as well as provides support to capture binding relationships between hardware and software compönents of the system.

2.4 Definitions

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 [39]. Given a constraint system, we require some method of determining, for any subset $S \subseteq C$, whether S is *satisfiable* (SAT) or *unsatisfiable* (UNSAT). When a subset S is SAT, this means that there exists an assignment allowed by all $C_i \in S$; when no such assignment exists, S is considered UNSAT. Given a constraint system C, there are certain subsets of C that are of interest in terms of satisfiability. Definitions 2-4 are taken from research by Liffiton et. al. [39].

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

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

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

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

A duality exists between MUSs of a constraint system and MCSs as established by Reiter [47]. This duality is defined in terms of *Minimal Hitting Sets* (MHS). A hitting set of a collection of sets A is a set H such that every set in A is "hit" be H; H contains at least one element from every set in A. Every MUS of a constraint system is a minimal hitting set of the system's MCSs, and likewise every MCS is a minimal hitting set of the system's MUSs [21,39,47].

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

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

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

We define a minimal cut set consistently with much of the research in this field [22,48].

Inductive Validity Cores: Given a complex model, it is useful to extract traceability information related to the proof; in other words, which elements of the model were necessary to construct the proof. An algorithm was introduced by Ghassabani, et. al. to provide Inductive Validity Cores (IVCs) as a way to determine which model elements are necessary for the inductive proofs of the safety properties for sequential systems [27]. 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 MIVCs) [5, 28].

The MIVC 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 proof the safety property.

Chapter 3

Proposed Approach

The contributions of this project can be seen as two main categories of research work. The first set was accomplished in the beginning phase of this project: behavioral and explicit error propagation through the implementation of the Safety Annex for AADL [52,54]. The remaining pieces of this research provide the bulk of the contribution and consist of the compositional generation of minimal cut sets through the transformation of inductive validity cores and using the fault tree generated by this transformation to compute the probability of a safety property violation.

The usage of the terms error, failure, and fault are defined in ARP4754A and are described here for ease of understanding [50]. An *error* is a mistake made in implementation, design, or requirements. A *fault* is the manifestation of an error and a *failure* is an event that occurs when the delivered service of a system deviates from correct behavior. If a fault is activated under the right circumstances, that fault can lead to a failure. The term *error propagation* is used to refer to the propagation of the corrupted state caused by an active fault.

3.1 The Safety Annex and Fault Modeling

3.1.1 Implementation

The Safety Annex is written in Java as a plug-in for the OSATE AADL toolset, which is built on Eclipse. It is not designed as a stand-alone extension of the language, but works with behavioral contracts specified using the AGREE AADL annex [20]. The architecture of the Safety Annex is shown in Figure 3.1.

AGREE contracts are used to define the nominal behaviors of system components as *guarantees* that hold when *assumptions* about the values the component's environment are met. When an AADL model is annotated with AGREE contracts and the fault model is created using the Safety Annex, the model is transformed through AGREE into a Lustre model [30] containing the behavioral extensions defined in the AGREE contracts for each system component.

When performing fault analysis, the Safety Annex extends the AGREE contracts to allow

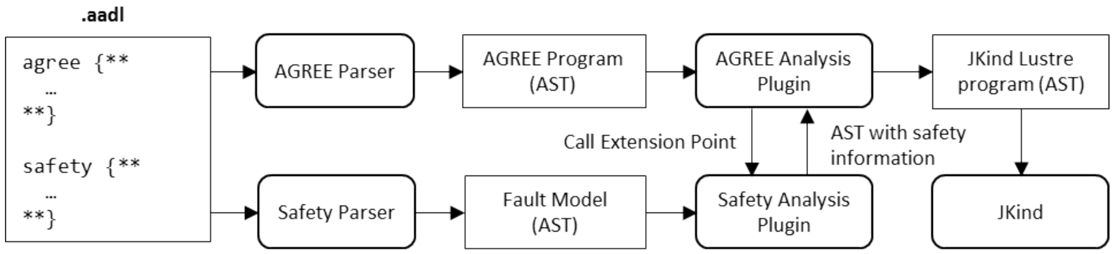
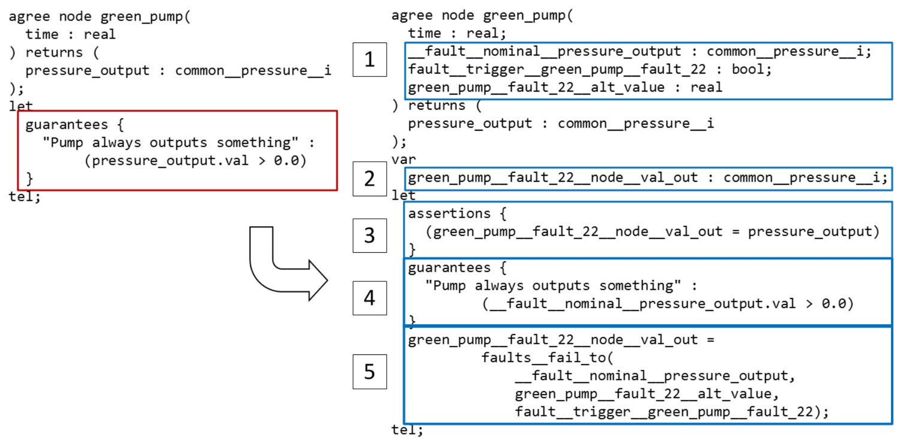


Figure 3.1: Safety Annex Plug-in Architecture

faults to modify the behavior of component inputs and outputs. An example of a portion of an initial AGREE node and its extended contract is shown in Figure 3.2. The left column of the figure shows the nominal Lustre pump definition is shown with an AGREE contract on the output; and the right column shows the additional local variables for the fault (boxes 1 and 2), the assertion binding the fault value to the nominal value (boxes 3 and 4), and the fault node definition (box 5). Once augmented with fault information, the AGREE model (translated into the Lustre dataflow language [30]) follows the standard translation path to the model checker JKind [25], an infinite-state model checker for safety properties.



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

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

Figure 3.2: Nominal AGREE Node and Extension with Faults

3.1.2 The Sensor System

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

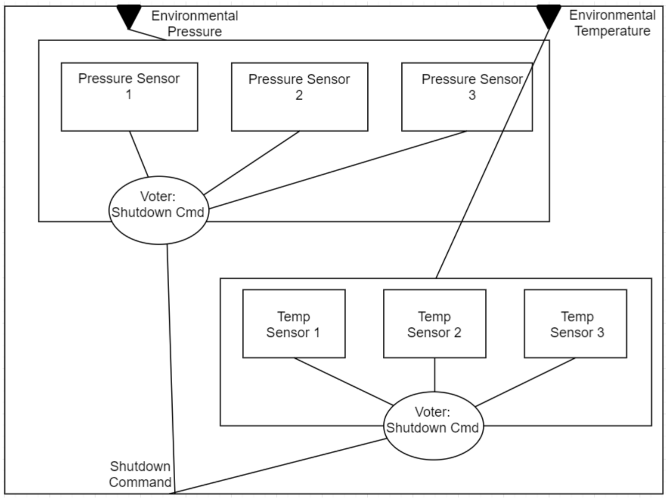


Figure 3.3: A Simple Sensor System

3.1.3 The Safety Annex for the Sensor System

The grammar of the Safety Annex is an extension of AADL and AGREE. Additional grammar elements are added to allow for fault information to be connected to each AADL component. The Safety Annex is used to add possible faulty behaviors to a component model. Within the AADL component instance model, an annex is added which contain the fault definitions for the given component. The flexibility of the fault definitions allows the user to define numerous types of fault *nodes* by utilizing the AGREE node syntax. An example of the AGREE and Safety Annexes defined on a pressure sensor in the sensor system is shown in Figure 3.4. The fault definition connects to a *fault node* that is defined in Figure 3.5 (this is seen in the example syntax in the fault defintion: *Common_Faults.stuck_false*. This node, *stuck_false*, defines the behavior to be always outputting false, even when the pressure is higher than the threshold.

When a fault is activated by its specified triggering conditions, it modifies the output of the component. This faulty behavior may violate the contracts of other components in the system, including assumptions of downstream components. The impact of a fault is computed by the AGREE model checker when the safety analysis is run on the fault model. The mechanism of this can be seen in the Lustre code generated by the safety analysis shown in Figure 3.2. If the trigger is active, the node outputs a faulty value which overrides the component's nominal output value.

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

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

    annex agree{**

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

    **};

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

end Pressure_Sensor;
```

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

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

Figure 3.5: A Fault Node Definition

- **Fault node statement**: This is seen as *Common_Faults.stuck_false* in the figure and refers to a file of fault node definitions containing commonly used faults such as stuck false, stuck true, fail to, and so on. This provides the node definition for behind the scenes linking in the Lustre code.

- **Input**: The inputs state which values from the component are passed into the node definition. In this case, the nominal component output *High_Pressure_Indicator* is passed into the node parameter *val_in*. If a node contains multiple parameters, the input list connects them to AADL component features.

- **Output**: The output from the fault node is designated to an AADL component feature. In this case, the node output (fail-to value if active) will be the component's output *High_Pressure_Indicator*.

- **Probability**: This field specifies the probability of occurrance for this fault. This is used only in probabilistic analysis.

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

3.2 Fault Modeling

Within the AADL component instance model, the Safety Annex is added which contain the fault definitions for the given component. When a fault is activated by its specified triggering conditions, it modifies the output of the component. This faulty behavior may violate the contracts of other components in the system, including assumptions of downstream components. The impact of a fault is computed by the AGREE model checker when the safety analysis is run on the fault model and thus determined implicitly; the error propagation is not explicitly defined as in other closely related tools [10, 24].

On the other hand, failures in hardware (HW) components can trigger behavioral faults in the system components that depend on them. This makes it beneficial to allow for explicit error propagation and the definition of dependencies in the fault model. For example, a CPU failure may trigger faulty behavior in the threads bound to that CPU or a failure in one HW component may trigger failure in other HW components located nearby, such as overheating, fire, or explosion in the containment location. The Safety Annex provides the capability to explicitly model the impact of hardware failures on other faults, behavioral or non behavioral.

Users specify dependencies between the HW component faults and faults that are defined in other components, either HW or SW. The hardware fault then acts as a trigger for 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. Within the implementation, this corresponds to a statement linking the active fault to all dependent faults. Thus, if the triggering fault is active, so are all dependencies.

3.3 Verification Capabilities in the Face of Component Failures

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

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

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

Figure 3.6: Max N Faults Analysis Statement

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

Figure 3.7: Probability Analysis Statement

Tying back to the fault tree analysis in traditional safety analysis, the former is analogous to restricting the cutsets to a specified maximum number of terms, and the latter is analogous to restricting the cutsets to only those whose probability is above some set value. In the former case, we assert that the sum of the true *fault__trigger* variables is at or below some integer threshold. In the latter, we determine all combinations of faults whose probabilities are above the specified probability threshold, and describe this as a proposition over *fault__trigger* variables. 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).

3.3.1 Maximum Fault Analysis

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

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

assert (__fault__global_count <= 3);
```

Figure 3.8: Lustre Statement for Fault Count

iterate through the possible combinations given this restriction and if a counterexample is found to a contract, one of these combinations are displayed to the user in a counterexample pane.

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

3.3.2 Probabilistic Threshold Analysis

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

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

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, the triggered dependent faults are added to the combination as appropriate.

Algorithm 1: Monolithic Probability Analysis

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

3.4 Generation of Minimal Cut Sets from MIVCs

3.4.1 The General Idea

As was previously explained, the MIVCs (Minimal Inductive Validity Cores) are MUSs (Minimal Unsatisfiable Subsets) of a constraint system. The MCSs (Minimal Correction Sets) can be obtained by all MUSs by generating the hitting sets of all MUSs. The main idea is to utilize the MUSs that are collected through the use of the All-MIVC algorithm in order to generate all MCSs. If the constraint system is defined to take into account faults, these MCSs can be transformed into MinCutSets.

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

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

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

The All-MIVC algorithm collects all minimal unsatisfiable subsets of a given transition system in terms of the *negation* of the top level property [5, 28]. Assuming that the nominal model proves (no faults are active), it is not surprising that the guarantees (constrained to *true*) and the negation of the safety property is UNSAT. The MUSs are the minimal explanation of the infeasibility of this constraint system; equivalently, these are the minimal sets of model elements necessary for proof of the safety property.

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

Because of the duality between MUSs and MCSs, using a hitting set algorithm all MCSs can be obtained from All-MIVCs. The MCS can be seen to correct the infeasibility of the constraint system and provides the minimal such correction. By removing the constraints from C that are found in any MCS, C becomes satisfiable. In terms of our constraint system with fault activation literals, by *activating* the faults in the MCS and *violating* the contracts in the MCS, we can prove the *negation* of the property P. If the contracts in the MCS are replaced with the faults that cause its violation, the MCS is transformed into a MinCutSet.

The Steps of Transformation from MUS to MinCutSet

1. Redefine constraint system by adding fault activation literals to the constraint system used by the All-MIVC algorithm.

2. Transform the MUSs (MIVCs) into MCSs by use of a hitting algorithm [26, 42].
3. Replace all contracts in the MCSs by the faults that cause their violation (MinCutSets for that contract).

3.4.2 Illustrative Example

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

and can be also written as:

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

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

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

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

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

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

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

3.5 Compositional Probabilistic Computations

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

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

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

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

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

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

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

Due to the independence assumption, $\mathcal{P}(S_1 \cap S_2)$ is computed as $\mathcal{P}(S_1) \cdot \mathcal{P}(S_2)$. If the set of MinCutSets are represented using a Binary Decision Diagram, there have are efficient ways of computing the probability cite these.

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

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

3.6 Evaluation of Results

Chapter 4

Conclusion

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

Furthermore, we propose the compositional generation of minimal cut sets to be used in the development of various artifacts used in system certification, such as FTA, FMEA, and single point of failure examinations. This generation is done through the collection of proof elements called MIVCs and their transformation.

Lastly, we propose to use the minimal cut sets and resulting fault trees generated through the transformation algorithms to calculate the probability of the top level event, or violation of the safety property.

References

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

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

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

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

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

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

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

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

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

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

[16] 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.

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

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

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

[20] 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.

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

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

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

[24] 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.

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

[26] 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.

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

[28] 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.

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

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

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

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

[42] 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.

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

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

[45] 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.

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

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

[48] 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.

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

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

[51] 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.

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

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

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