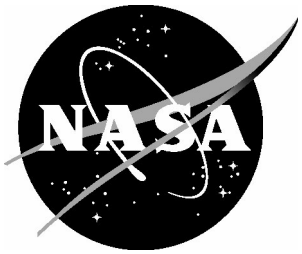
NASA/CR-20210017388



Architectural Modeling and Analysis for Safety Engineering

Danielle Stewart[1], Jing (Janet) Liu[2], Darren Cofer[2],
Mats Heimdahl[1], Michael W. Whalen[1], and Michael Peterson[3]
[1]University of Minnesota Department of Computer Science and Engineering
[2]Collins Aerospace: Trusted Systems - Enterprise Engineering
[3]Collins Aerospace: Flight Controls Safety Engineering - Avionics

June 2021

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

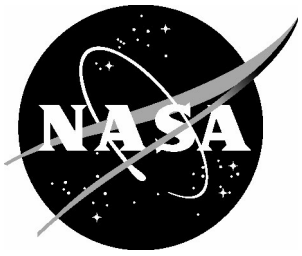
For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at http://www.sti.nasa.gov

- Help desk contact information:

https://www.sti.nasa.gov/sti-contact-form/
and select the "General" help request type

NASA/CR-20210017388



Architectural Modeling and Analysis for Safety Engineering

Danielle Stewart[1], Jing (Janet) Liu[2], Darren Cofer[2],
Mats Heimdahl[1], Michael W. Whalen[1], and Michael Peterson[3]
[1]University of Minnesota Department of Computer Science and Engineering
[2]Collins Aerospace: Trusted Systems - Enterprise Engineering
[3]Collins Aerospace: Flight Controls Safety Engineering - Avionics

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL16AA09B

June 2021

Available from:

NASA STI Program/ Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

30 September 2019

Architectural Modeling and Analysis for Safety Engineering (AMASE)
Final Report

Prepared for NASA Langley Research Center

Contract NNL16AA09B / T.O. NNL16AB07T

Item 4.14 / Task 3.3.2

Technical Point of Contact:
Dr. Darren Cofer
Rockwell Collins, Inc.
7805 Telegraph Rd. #100
Bloomington, MN 55438
Telephone: (319) 263-2571
darren.cofer@rockwellcollins.com

Business Point of Contact:
Mr. Scott Jensen
Rockwell Collins, Inc.
400 Collins Rd. NE
Cedar Rapids, IA 52498
Telephone: (319) 263-7545
scott.jensen@rockwellcollins.com

Rockwell Collins, Inc.
400 Collins Rd. NE
Cedar Rapids, Iowa 52498

Architectural Modeling and Analysis for Safety Engineering

Danielle Stewart[1], Jing (Janet) Liu[2], Darren Cofer[2], Mats Heimdahl[1], Michael W. Whalen[1], and Michael Peterson[3]

[1] *University of Minnesota Department of Computer Science and Engineering*
[2] *Collins Aerospace: Applied Research & Technology*
[3] *Collins Aerospace: Flight Controls Safety Engineering - Avionics*

June 11, 2021

Contents

List of Figures

List of Tables

List of Algorithms

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. In this report we describe an extension to the Architecture Analysis and Design Language (AADL) that supports modeling of system behavior under failure conditions. This *Safety Annex* enables the independent modeling of component failures and allows safety engineers to weave various types of fault behavior into the nominal system model. The accompanying tool support uses model checking to propagate errors from their source to their effect on safety properties without the need to add separate propagation specifications. The tool also captures all minimal set of fault combinations that can cause violation of the safety properties, that can be compared to qualitative and quantitative objectives as part of the safety assessment process. We describe the Safety Annex, illustrate its use with a representative example, and discuss and demonstrate the tool support enabling an analyst to investigate the system behavior under failure conditions.

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. Developing adequate understanding, especially for software components, is a difficult and time consuming endeavor. Given the increase in model-based development in critical systems [12, 41, 44, 47, 52], leveraging the resultant models in the safety analysis process holds great promise in terms of analysis accuracy as well as efficiency.

In this report we describe the *Safety Annex* for the system engineering language AADL (Architecture Analysis and Design Language), a SAE Standard modeling language for Model-Based Systems Engineering (MBSE) [2]. The Safety Annex allows an analyst to model the failure modes of components and then "weave" these failure modes together with the original models developed as part of MBSE. The safety analyst can then leverage the merged behavioral models to propagate errors through the system to investigate their effect on the safety requirements. Determining how errors propagate through software components is currently a costly and time-consuming element of the safety analysis process. The use of behavioral contracts to capture the error propagation characteristics of software component without the need to add separate propagation specifications (*implicit* error propagation) is a significant benefit for safety analysts. In addition, the annex allows modeling of dependent faults that are not captured through the behavioral models (*explicit* error propagation), for example, the effect of a single electrical failure on multiple software components or the effect

hardware failure (e.g., an explosion) on multiple behaviorally unrelated components. Furthermore, we will describe the tool support enabling engineers to investigate the correctness of the nominal system behavior (where no failures have occurred) as well as the system's resilience to component failures. We illustrate the work with a substantial example drawn from the civil aviation domain.

Our work can be viewed as a continuation of work conducted by Joshi et al. where they explored model-based safety analysis techniques defined over Simulink/Stateflow [54] models [19, 45–47]. Our current work extends and generalizes this work and provide new modeling and analysis capabilities not previously available. For example, the Safety Annex allows modeling explicit error propagation, supports compositional verification and exploration of the nominal system behavior as well as the system's behavior under failure conditions. Our work is also closely related to the existing safety analysis approaches, in particular, the AADL Error Annex (EMV2) [36], COMPASS [14], and AltaRica [9, 60]. Our approach is significantly different from previous work in that unlike EVM2 we leverage the behavioral modeling for implicit error propagation, we provide compositional analysis capabilities not available in COMPASS, and in addition, the Safety Annex is fully integrated in a model-based development process and environment unlike a stand alone language such as AltaRica.

The main contributions of the Safety Annex and this project are:

- close integration of behavioral fault analysis into the *Architecture Analysis and Design Language* AADL, which allows close connection between system and safety analysis and system generation from the model,
- support for *behavioral specification of faults* and their *implicit propagation* (both symmetric and asymmetric) through behavioral relationships in the model, in contrast to existing AADL-based annexes (HiP-HOPS [26], EMV2 [36]) and other related toolsets (COMPASS [14], Cecilia [7], etc.),
- additional support to capture binding relationships between hardware and software and logical and physical communications,
- compute all minimal set of fault combinations that can cause violation of the safety properties to be compared to qualitative and quantitative objectives as part of the safety assessment process, and
- guidance on integration into a traditional safety analysis process.

2 Preliminaries

One of our goals is to transition the tools we have developed into use by the safety engineers who perform safety assessment of avionics products. Therefore, we need to understand how the tools and the models will fit into the existing safety assessment and certification process.

2.1 Traditional Safety Assessment Process

The traditional safety assessment process at the system level is based on ARP4754A [65] and ARP4761 [66]. 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. Section 2.4 provides a comparison of our approach with it.

2.2 Modeling Language for System Design

Figure 1 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.

We are using the Architectural Analysis and Design Language (AADL) [35] 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" [2]. 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

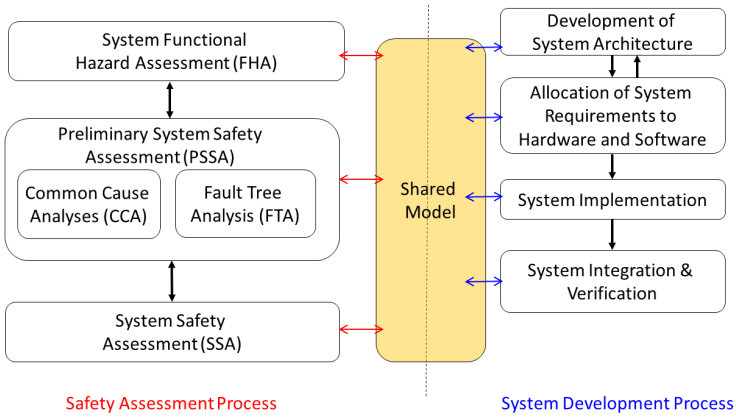


Figure 1: Use of the Shared System/Safety Model in the ARP4754A Safety Assessment Process

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.

The Assume Guarantee Reasoning Environment (AGREE) [29] is a tool for formal analysis of behaviors in AADL models. AGREE 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 [42] and then queries a user-selected model checker to conduct the back-end analysis. The analysis can be performed compositionally following the architecture hierarchy such that analysis at a higher level is based on the components at the next lower level. 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 [29].

In our prior work [72], we added an initial failure effect modeling capability to the AADL/AGREE language and tool set. We are continuing this work so that our tools and methodology can be used to satisfy system safety objectives of ARP4754A and ARP4761.

2.3 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 use the Safety Annex to 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.
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.

There are other tools purpose-built for safety analysis, including AltaRica [60], smartIFlow [43] and xSAP [10]. These tools and their accompanying notations are separate from the system development model. Other tools extend existing system models, such as HiP-HOPS [26] and the AADL Error Model Annex, Version 2 (EMV2) [36]. EMV2 uses enumeration of faults in each component and explicit propagation of faulty behavior to perform error analysis. The required propagation relationships must be manually added to the system model and can become complex, leading to mistakes in the analysis.

In contrast, the Safety Annex supports model checking and quantitative reasoning by attaching behavioral faults to components and then using the normal behavioral propagation and proof mechanisms built into the AGREE AADL annex. This allows users to reason about the evolution of faults over time, and produce counterexamples demonstrating how component faults lead to failures. Our approach adapts the work of Joshi et al. [47] to the AADL modeling language. Stewart et al. provide more information on the approach [72], and the tool and relevant documentation can be found at: `https://github.com/loonwerks/AMASE/`.

2.4 Comparison with Proposed MBSA Appendix to ARP4761A

ARP4754A, the Guidelines for Development of Civil Aircraft and Systems [65], provides guidance on applying development assurance at each hierarchical level throughout the development life cycle of highly-integrated/complex aircraft systems. ARP4761, the Guidelines and Methods for Conducting Safety Assessment Process on Civil Airborne Systems and Equipment [66], identifies a systematic means to show compliance. A Model Based Safety Analysis (MBSA) appendix has been drafted to the upcoming revision of ARP4761 to provide concepts and processes with Model Based Safety Analysis.

We have reviewed the draft appendix and found that our approach is consistent with the MBSA appendix in the following ways:

- The common goal is to use MBSA for an equivalent analysis to the traditional safety analysis methods (e.g., Fault Trees) to support safety assessment processes.
- Both use an analytical model of the system to capture failure propagation. In the model, system architecture, nominal and faulty functional behaviors are captured. The model evolves as the system design evolves.
- Both use software application/tools to perform analysis on the model and generate outputs (e.g., failure sequences, minimal cut sets that result in the failure condition under analysis). The MBSA appendix also mentioned that model checking can be used to perform an exhaustive exploration of the state space of the model.
- Outputs generated from the analysis are to be compared to qualitative and/or quantitative objectives and requirements as part of the safety assessment process. Furthermore, the outputs drive evolution of system design.

Our approach goes beyond what is envisioned in the MBSA appendix in the following ways:

- The MBSA Appendix is not advocating a single unified model used by both system development and safety assessment activities. The model is safety specific and driven by the types of safety assessment to be conducted. However, the initial safety model may be derived from the system design model, and may be closer to the design at the lower levels of the design process.
- In the MBSA Appendix, the failure propagation modeling focuses on the inside internal flows in the components, which is similar to the bottom-up method in Failure Modes and Effects Analysis. Different components are connected by inputs and outputs, and no behavioral constraints are specified on data entering and exiting components. This leaves inter-component propagation to be explored by the analysis.

In summary, our approach provides a new way to do safety analysis. It uses an unified model that is shared by system development and safety assessment. The model

captures architecture and behavioral information for propagation within components and between components. It is a property driven approach that is consistent between system verification and safety analysis.

3 Fault Modeling with the Safety Annex

To demonstrate the fault modeling capabilities of the Safety Annex we will use the Wheel Brake System (WBS) described in AIR6110 [1]. This system is a well-known example that has been used as a case study for safety analysis, formal verification, and contract based design [12, 18, 19, 45]. The preliminary work for the safety annex was based on a simple model of the WBS [72]. 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 [19].

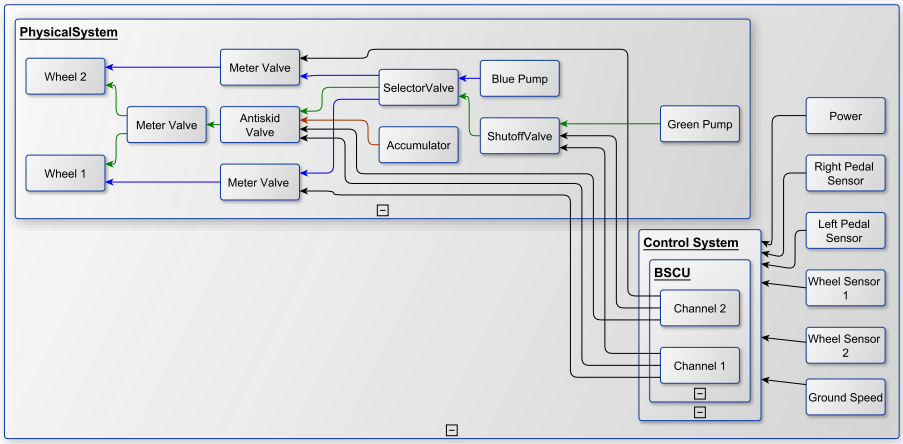


Figure 2: Wheel Brake System

The WBS is composed of two main parts: the Line Replaceable Unit control system and the electro-mechanical physical system. The control system electronically controls the physical system and contains a redundant channel of the Braking System Control Unit (BSCU) in case a detectable fault occurs in the active channel. It also commands antiskid braking. The physical system consists of the hydraulic circuits running from hydraulic pumps to wheel brakes as well as valves that control the hydraulic fluid flow. This system provides braking force to each of the eight wheels of the aircraft. The wheels are all mechanically braked in pairs (one pair per landing gear). For simplicity, Figure 2 displays only two of the eight wheels.

There are three operating modes in the WBS model:

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

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

- In *alternate* mode, the system is composed of a *blue* hydraulic pump, four meter valves, and four antiskid shutoff valves, one for each landing gear. The meter valves are mechanically commanded through the pilot pedal corresponding to each landing gear. If the selector detects lack of pressure in the green circuit, it switches to the blue circuit.
- In *emergency* mode, the system mode is entered if the *blue* hydraulic pump fails. The accumulator pump has a reserve of pressurized hydraulic fluid and will supply this to the blue circuit in emergency mode.

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

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

An example system safety property is to ensure that there is no inadvertent braking of any of the wheels. This is based on a failure condition described in AIR6110 is *Inadvertent wheel braking on one wheel during takeoff shall be less than 1E-9 per takeoff*. Inadvertent braking means that braking force is applied at the wheel but the pilot has not pressed the brake pedal. In addition, the inadvertent braking requires that power and hydraulic pressure are both present, the plane is not stopped, and the wheel is rolling (not skidding). The property is stated in AGREE such that inadvertent braking does *not* occur, as shown in Figure 3.

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

3.1 Component Fault Modeling

The usage of the terms error, failure, and fault are defined in ARP4754A and are described here for ease of understanding [65]. An *error* is a mistake made in implementation, design, or requirements. A *fault* is the manifestation of an error and a *failure* is an event that occurs when the delivered service of a system deviates from correct behavior. If a fault is activated under the right circumstances, that fault can lead to a failure. The

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

The Safety Annex is used to add possible faulty behaviors to a component model. Within the AADL component instance model, an annex is added which contain the fault definitions for the given component. The flexibility of the fault definitions allows the user to define numerous types of fault *nodes* by utilizing the AGREE node syntax. A library of common fault nodes has been written and is available in the project GitHub repository [69]. Examples of such faults include valves being stuck open or closed, output of a software component being nondeterministic, or power being cut off. When the fault analysis requires fault definitions that are more complex, these nodes can easily be written and used in the model.

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

The majority of faults that are connected to outputs of components are known as *symmetric*. That is, whatever components receive this faulty output will receive the same faulty output value. Thus, this output is seen symmetrically. An alternative fault type is *asymmetric*. This pertains to a component with a 1-n output: one output which is sent to many receiving components. This fault can present itself differently to the receiving components. For instance, in a boolean setting, one component might see a true value and the rest may see false. This is also possible to model using the keyword *asymmetric*. For more information on fault definitions and modeling possibilities, we refer readers to the Safety Annex Users Guide [69].

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

Figure 4 shows the AADL pedal sensor component with a contract for its nominal behavior. The sensor has only one input, the mechanical pedal position, and one output, the electrical pedal position. A property that governs the behavior of the component is that the mechanical position should always equal the electronic position. (The expression *true* $\rightarrow$ *property* in AGREE is true in the initial state and then afterwards it is only true if property holds.)

One possible failure for this sensor is inversion of its output value. This fault can be triggered with probability 5.0×10^{-6} as described in AIR6110 (in reality, the component failure probability is collected from hardware specification sheets). The Safety Annex definition for this fault is shown in Figure 5. Fault behavior is defined through the use of a fault node called *inverted_fail*. When the fault is triggered, the nominal output of the component (*elec_pedal_position*) is replaced with its failure value (*val_out*).

The WBS fault model expressed in the Safety Annex contains a total of 33 different

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

  -- Behavioral contracts for subcomponent
  annex agree {**

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

Figure 4: An AADL System Type: The Pedal Sensor

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

Figure 5: The Safety Annex for the Pedal Sensor

fault types and 141 fault instances. The large number of fault instances is due to the redundancy in the system design and its replication to control 8 wheels.

3.2 Implicit Error Propagation

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

On the contrary, in the AADL Error Model Annex, Version 2 (EMV2) [36] approach, all errors must be explicitly propagated through each component (by applying fault types on each of the output ports) in order for a component to have an impact on the rest of the system. To illustrate the key differences between implicit error propagation provided in the Safety Annex and the explicit error propagation provided in EMV2, we use a simplified behavioral flow from the WBS example using code fragments from EMV2, AGREE, and the Safety Annex.

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

In the EMV2 approach (top half of Figure 6), the “NoService” fault is explicitly propagated through all of the components. These fault types are essentially tokens that do not capture any analyzable behavior. At the system level, analysis tools supporting

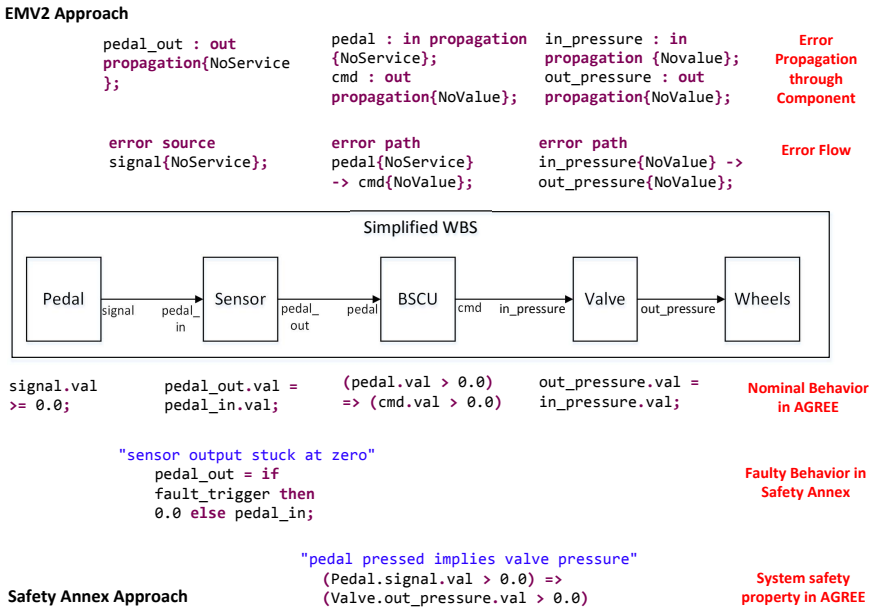


Figure 6: Differences between Safety Annex and EMV2

the EMV2 annex can aggregate the propagation information from different components to compose an overall fault flow diagram or fault tree.

When a fault is triggered in the Safety Annex (bottom half of Figure 6), the output behavior of the Sensor component is modified. In this case the result is a "stuck at zero" error. The behavior of the BSCU receives a zero input and proceeds as if the pedal has not been pressed. This will cause the top level system contract to fail: *pedal pressed implies brake pressure output is positive*.

3.3 Explicit Error Propagation

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

To better model faults at the system level dependent on HW failures, a fault model element is introduced called a *hardware fault*. Users are not required to specify behav-

ioral effects for the HW faults, nor are data ports necessary on which to apply the fault definition. An example of a model component fault declaration is shown below:

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

Users specify dependencies between the HW component faults and faults that are defined in other components, either HW or SW. The hardware fault then acts as a trigger for dependent faults. This allows a simple propagation from the faulty HW component to the SW components that rely on it, affecting the behavior on the outputs of the affected SW components.

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

```
annex safety{**

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

**};
```

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

3.4 Fault Analysis Statements

The fault analysis statement (also referred to as the fault hypothesis) resides in the AADL system implementation that is selected for verification. This may specify either a maximum number of faults that can be active at any point in execution:

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

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

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

Tying back to the fault tree analysis in traditional safety analysis, the former is analogous to restricting the cutsets to a specified maximum number of terms, and the latter is analogous to restricting the cutsets to only those whose probability is above some set value. In the former case, we assert that the sum of the true *fault_trigger* variables

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

4 Byzantine Fault Modeling

A *Byzantine* or *asymmetric* fault is a fault that presents different symptoms to different observers [31]. 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.

4.1 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 7 under "Nominal System." If a symmetric fault was present on this output, all four connected components would see the same faulty behavior. An asymmetric fault should be able to present arbitrarily different values to the connected components.

To this end, "communication nodes" are inserted on each connection from component A to components B, C, D, and E (shown in Figure 7 under "Fault Model Architecture." From the users perspective, the asymmetric fault definition is associated with component A's output and the architecture of the model is unchanged from the nominal model architecture. Behind the scenes, these communication nodes are created to facilitate potentially different fault activations on each of these connections. The fault definition used on the output of component A will be inserted into each of these communication nodes as shown by the red circles at the communication node output in Figure 7.

An asymmetric fault is defined for Component A as in Figure 8. This fault defines an asymmetric failure on Component A that when active, is stuck at a previous value (*prev(Output, 0)*). This can be interpreted as the following: some connected components may only see the previous value of Comp A output and others may see the correct (current) value when the fault is active. This fault definition is injected into the communication nodes and which of the 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.

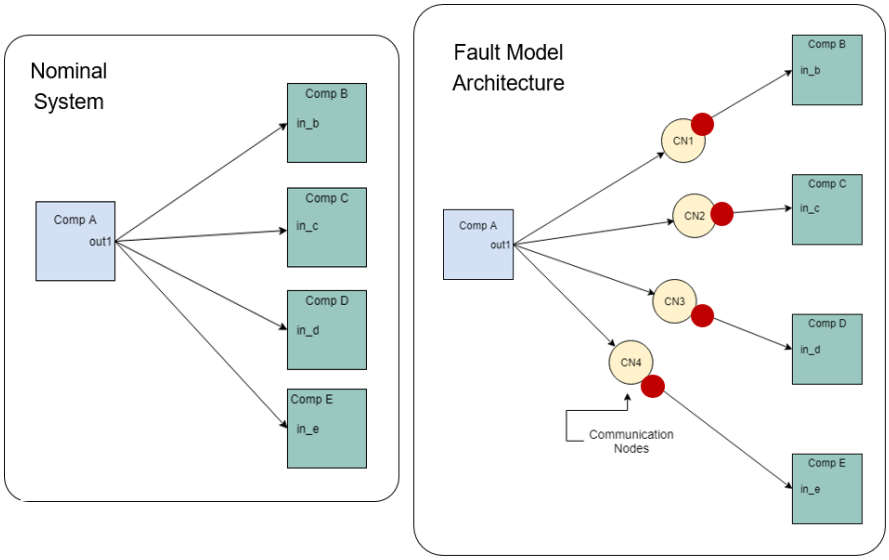


Figure 7: Communication Nodes in Asymmetric Fault Implementation

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

Figure 8: Asymmetric Fault Definition in the Safety Annex

4.2 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.

4.3 The Agreement Protocol Implementation in AGREE

In order to mitigate this problem, special attention must be given to the behavioral model. Using the strategies outlined in previous research [24, 31], 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 9.

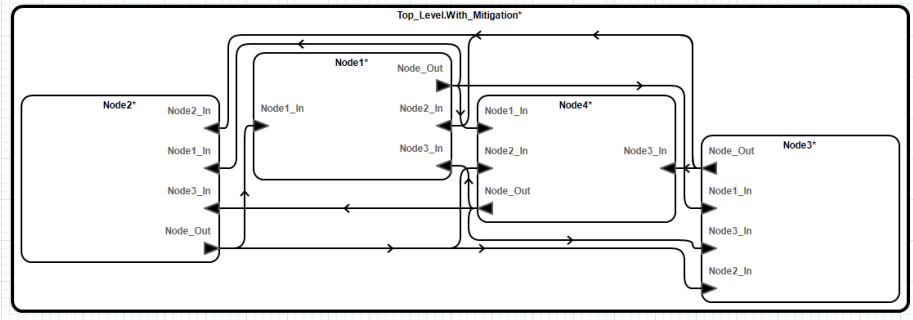


Figure 9: Updated PID Example Architecture

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 10. These outputs

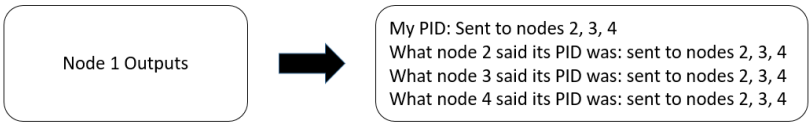


Figure 10: Description of the Outputs of Each Node in the PID Example

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 11.

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

The fault definition for each node's output can arbitrarily affect the data fields. This is a nondeterministic fault in two ways. It is nondeterministic how many receiving nodes see incorrect values and it is nondeterministic how many of the data fields are affected by this fault. This can be accomplished through the fault definition shown in Figure 12 and the fault node definition in Figure 13.

```
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 12: 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 13: 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 14 and Figure 15 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 14: 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 15: Agreement Protocol Contract in AGREE Regarding Non-failed Nodes

Referencing Fault Activation Status To fully implement the agreement protocol, it must be possible to describe whether or not a subcomponent is failed by specifying if any faults defined for the subcomponents is activated. In the Safety Annex, this is made possible through the use of a *fault activation* statement. Users can declare boolean *eq* variables in the AGREE annex of the AADL system where the AGREE verification applies to that system's implementation. Users can then assign the activation status of specific faults to those *eq* variables in Safety Annex of the AADL system implementation (the same place where the fault analysis statement resides). This assignment links each specified AGREE boolean variable with the activation status of the specified fault

activation literal. The AGREE boolean variable is true when and only when the fault is active. An example of this for the PID example is shown in Figure 16. Each of the *eq* variables declared in AGREE (i.e., *n1_failed, n2_failed, n3_failed, n4_failed*) is linked to the fault activation status of the *Asym_Fail_Any_PID_To_Any_Value* fault defined in a node subcomponent instance of the AADL system implementation (i.e., *node1*, *node2*, *node3*, *node4*).

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

    analyze: max 2 fault

**};
```

Figure 16: Fault Activation Statement in PID Example

4.4 PID Example Analysis Results

The nominal model verification shows that all properties are valid. Upon running verification of the fault model (*Verify in the Presence of Faults*) with one active fault, the first four properties stating that all nodes agree on the correct value (Figure 14) 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 15) 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 14). This is expected if faults are present. The last four guarantees (all non-failed nodes agree) are verified as true with one active fault.
- Fault model with two active faults: All 8 guarantees fail. This is expected since in order to be resilient up to two active faults ($f = 2$), we would need $3f+1 = 7$ nodes and $f + 1 = 3$ rounds of communication.

This model is in Github and is called PIDByzantineAgreement [69].

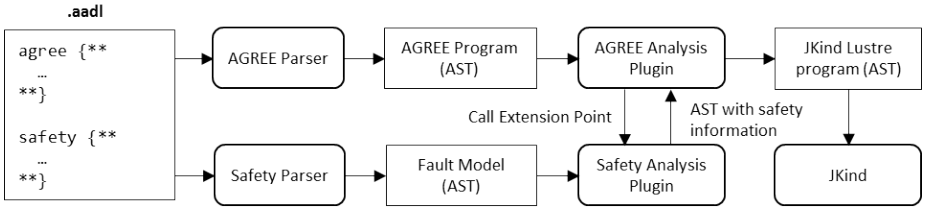


Figure 17: Safety Annex Plug-in Architecture

5 Tool Architecture and 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 [29]. The architecture of the Safety Annex is shown in Figure 17.

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 [42] containing the behavioral extensions defined in the AGREE contracts for each system component.

When performing fault analysis, the Safety Annex extends the AGREE contracts to allow faults to modify the behavior of component inputs and outputs. An example of a portion of an initial AGREE node and its extended contract is shown in Figure 18. 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 [42]) follows the standard translation path to the model checker JKind [37], 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
3 assertions {
    (green_pump__fault_22__node__val_out = pressure_output)
  }
4 guarantees {
    "Pump always outputs something" :
        (__fault__nominal__pressure_output.val > 0.0)
  }
5 green_pump__fault_22__node__val_out =
        faults__fail_to(
            __fault__nominal__pressure_output,
            green_pump__fault_22__alt_value,
            fault__trigger__green_pump__fault_22);
tel;
```

Figure 18: Nominal AGREE Node and Extension with Faults

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

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

Generate Minimal Cut Sets: This analysis collects all minimal set of fault combinations that can cause violation of a property. Given a complex model, it is often useful to extract traceability information related to the proof, in other words, which portions 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 [39]. Given a safety property of the system, a model checker can be invoked in order 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 Inductive Validity Cores (All-MIVCs) to provide a full enumeration of all minimal set of model elements necessary for the inductive proofs of a safety property [40].

In this approach, we use the all MIVCs algorithm to consider 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 what are called 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. In Section 7, we show the formal definitions in detail. The leaf nodes contribute only constrained faults to the IVC elements as shown in Figure 19.

In the non-leaf layers of the program, both contracts and constrained faults are considered as shown in Figure 20. 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 all MIVCs algorithm returns the minimal set of these elements necessary to prove the properties. This equates to any contracts or inactive faults that must be present in order for the verification of properties in the model. From here, we perform a number of algorithms to transform all MIVCs into minimal cut sets.

6 Analysis of the Model

In this section we describe results from the nominal model analysis and the fault analysis.

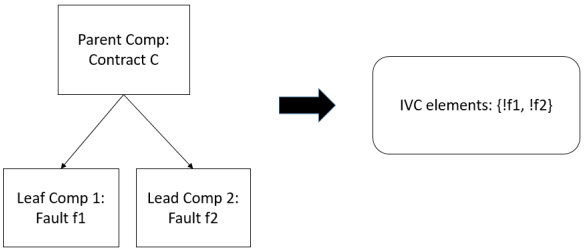


Figure 19: IVC Elements used for Consideration in a Leaf Layer of a System

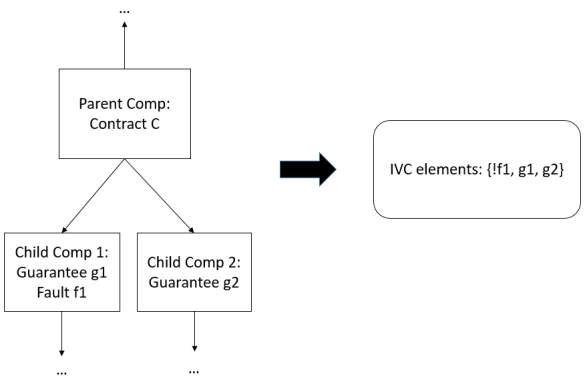


Figure 20: IVC Elements used for Consideration in a Middle Layer of a System

6.1 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 [3, 28]

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 1. Given that there are 8 wheels, contract S18-WBS-0325-wheelX is repeated 8 times, one for each wheel. The behavioral model in total consists of 36 assumptions and 246 supporting guarantees.

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 1: Safety Properties of WBS

6.2 Fault Model Analysis

There are two main options for fault model analysis using the Safety Annex. The first option injects faulty behavior allowed by faulty hypothesis into the AGREE model and returns this model to JKind for analysis. This allows for the activity of faults within the model and traceability information provides a way for users to view a counterexample to a violated contract in the presence of faults. The second option is used to generate minimal cut sets for the model. The model is annotated with fault activation that are constrained to false as well as intermediate level guarantees as model elements for consideration for the all Minimal Inductive Validity Cores (All-MIVCs) algorithm. The All-MIVCs traces the minimal set of model elements used to produce minimal cut sets and is described in Section 7. This subsection presents these options and discusses the analytical results obtained.

6.2.1 Verification in the Presence of Faults: Max N Analysis

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

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

assumption that the direct sub-level contracts are valid. This form of analysis is helpful to see weaknesses in a given layer of the system.

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

6.2.2 Verification in the Presence of Faults: Probabilistic Analysis

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

Probabilistic analysis done in this way must utilize the monolithic AGREE option. For compositional probabilistic analysis, see Section 6.2.4.

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

Algorithm 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);
```

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 HW faults are added to the combination as appropriate. The dependencies are implemented in the *Verify in the Presence of Faults* options for analysis, but not yet implemented in the *Generate Minimal Cut Sets* analysis options.

6.2.3 Generate Minimal Cut Sets: Max N Analysis

As described in Section 5, *Generate Minimal Cut Sets* analysis uses the All-MIVCs algorithm to provide a full enumeration of all minimal set of model elements necessary for the proof of each top-level safety property in the model, and then transforms all MIVCs into all minimal cut sets. In Max N analysis, the minimal cut sets are pruned to include only those with at cardinality less or equal to the max N number specified in the fault hypothesis and displayed to the user.

Generation of minimal cut sets was performed on the Wheel Brake System and results are shown in Table 2. Notice in Table 2, 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 2 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 1.)

Table 2: WBS Minimal Cut Set Results for Max n Hypothesis

Property	$\mathbf{n = 1}$	$\mathbf{n = 2}$	$\mathbf{n = 3}$	$\mathbf{n = 4}$	$\mathbf{n = 5}$
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

As can be seen in Table 2, 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 impossible to manually sift through multiple thousands of cut sets, but an analyst will instead filter out the combinations that are sufficiently unlikely to occur based on a truncation limit. In the next subsection (Generate Minimal Cut Sets: Probabilistic Analysis), we discuss the use of a truncation limit through probabilistic analysis. The probabilistic approach presents more realistic and useful number of cut sets for consideration.

6.2.4 Generate Minimal Cut Sets: Probabilistic Analysis

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

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

As shown in Table 3, the number of allowable combinations drops considerably when given probabilistic threshold as compared to just fault combinations of certain cardinalities. For example, one contract (inadvertent wheel braking of all wheels) had over a million minimal cut sets produced when looking at it in terms of max N analysis, but after taking probabilities into account, it is seen on Table 3 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.

Table 3: WBS Minimal Cut Set Results for Probabilistic Hypotheses

Property	$\mathbf{n = 1}$	$\mathbf{n = 2}$	$\mathbf{n = 3}$	$\mathbf{n = 4}$	$\mathbf{n = 5}$
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

In Table 3, the property 0321 has a truncation limit of 1.0×10^{-9} with 8 single points of failure. If this property has a catastrophic classification, these single points of failure must be eliminated. Likewise with cut sets of cardinality n = 2, there are a total of 3665 combinations that a safety analyst must manually examine. Within this analysis framework, there are multiple ways to address the number of cut sets. One is to re-examine how the faults are modeled (e.g., consolidate a valve's two failure modes into one as fail-open and fail-closed cannot occur the same time) and another is to re-evaluate the design of the model which is discussed in detail in an upcoming subsection (Use of Analysis Results to Drive Design Change).

6.2.5 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 21.

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 22.

3. The tool can also generate fault tree and minimal cut set information formatted as input to the SOTERIA tool [53] to produce hierarchical fault trees that

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

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

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

Figure 21: Detailed Output of MinCutSets

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

Figure 22: Tally Output of MinCutSets

are consistent with the system architecture/component verification layers, or flat fault trees consist of minimal cut sets only, both in graphical form. A sample graphical fault tree output from the SOTERIA tool is shown in Figure 23. The SOTERIA tool is also able to compute the probabilities for the top level event from a given fault tree. However, based on experience with the WBS example, our tool was a more scalable solution as it produces minimal cut sets for more complex systems, also in shorter amount of time. The text format of the minimal cut sets seemed anecdotally easier to read than the graphical format for larger systems.

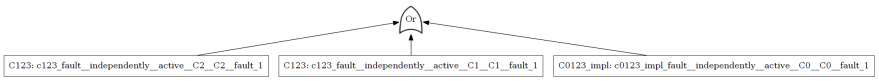


Figure 23: Example SOTERIA Fault Tree

6.2.6 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 design flaws and how faults can affect the behavior of the system). This safety property description can be found in detail in Section 3. Upon running max n compositional fault analysis with $n = 1$, this particular fault was shown to be a single point of failure for this safety property. A counterexample is shown in Figure 24 showing the active fault on the pedal sensor.

Problems | Properties | AADL Property Values | Classifier Information | AGREE Results | AGREE Counterexample | Console

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 24: AGREE counterexample for inadvertent braking safety property

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.

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 25 outlines these architectural changes that were made in the model.

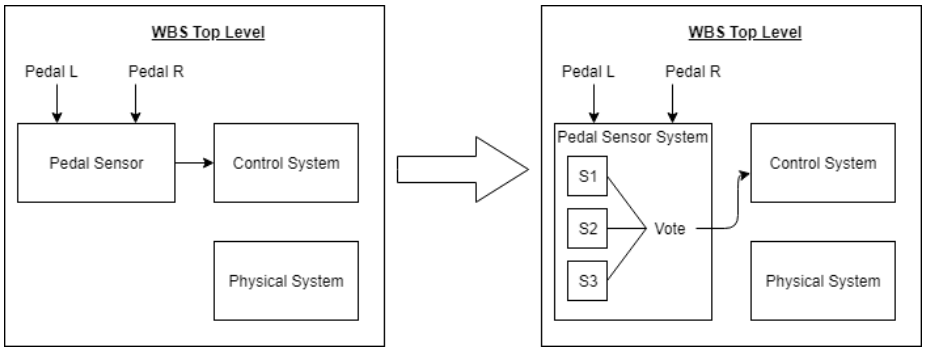


Figure 25: Changes in the architectural model for fault mitigation

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 [69]. This repository also contains all models used in this project.

7 Theoretical Foundations

There are two different types of fault analysis that can be performed on a fault model, *Verification in the Presence of Faults*, and *Generate Minimal Cut Sets*, as introduced in Section 5. The theoretical foundations used to verify a model in the presence of faults relies on AGREE and the theory underlying the assume guarantee environment [28];

this theory will not be discussed further in this report. The underlying theoretical framework used in the generation of minimal cut sets is described in detail in this section.

7.1 Introduction

Risk and safety analyses are important activities used to ensure that critical systems operate in an expected way. From nuclear power plants and airplanes to heart monitors and automobiles, critical systems are ubiquitous in our society. These systems are required to operate safely under nominal and faulty conditions. Proving that the system operates within some level of safety when failures are present is an important aspect of critical systems development and falls under the discipline of safety analysis. Safety analysis produces various safety related artifacts that are used during development and certification of critical systems [65]. Examples include *minimal cut sets* – each set represents the minimal set of faults that must all occur in order to violate a safety property and *fault trees* – the evaluation that determines all credible failure combinations which could cause an undesired top level hazard event. The fault tree can be transformed to an equivalent Boolean formula whose literals appear in the minimal cut sets. Since the introduction of minimal cut sets in the field of safety analysis, much research has been performed to address the generation of these sets and associated formulae [34, 64, 73]. As critical systems get larger, more minimal cut sets are possible with increasing cardinality. In recent years, symbolic model checking has been used to address scaling the analysis of systems with millions of minimal cut sets [8, 21, 67].

The state space explosion is a challenge when performing formal verification on industrial sized systems. This problem can arise from combining parallel processes together and attempting to reason monolithically over them. Compositional reasoning takes advantage of the hierarchical organization of a system model. A compositional approach verifies each component of the system in isolation and allows global properties to be inferred about the entire system [5]. The *assume-guarantee* paradigm is commonly used in compositional reasoning where the assumed behavior of the environment implies the guaranteed behavior of the component [29].

Using an assume-guarantee reasoning framework, we extend the definition of the nominal transition system to allow for unconstrained guarantees. We use this idea to reason about all possible violations of a safety property per layer of analysis and then compose the results.

After we provide the formalization, we describe the implementation in the OSATE tool for the Architecture Analysis and Design Language (AADL) [35]. AADL has two annexes that are of interest to us: the Assume-Guarantee Reasoning Environment (AGREE) [29] and the safety annex [68]. AGREE provides the assume-guarantee reasoning required for the transition system extension, and the safety annex allows us to define faults on component outputs. To implement the formalization, we look to recent work in formal verification. 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_MIVCs`) [39, 40]. Inductive validity cores produce the minimal sets of model elements necessary to prove a property. Each set contains the behavioral contracts – the requirement specifications of compo-

nents – used in a proof. We collect all MIVCs per layer to generate the minimal cut sets and thus the fault trees to be composed.

This section presents a compositional approach to generating *fault forests* (sets of fault trees) and associated minimal cut sets, allowing us to reason uniformly about faults in various types of system components and their impact on system properties. The main contributions of this research include the formalization of the composition of fault forests and its implementation. Our objective in creating this compositional analysis approach is to provide safety engineers with better tools so that they do not lose sight of the fault forest for the trees.

7.2 Running Example

In a typical Pressurized Water Reactor (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 steam line directs the steam to the main turbine causing it to turn the turbine generator, which in turn 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 simplified version of a subsystem within a PWR that monitors these factors. Each subsystem contain three sensors that monitor pressure, temperature, and radiation. If any of these conditions are too high, a shut down command is sent from the sensors to the parent components. The temperature, pressure, and radiation sensor subsystems each contain three associated sensors for redundancy. Each sensor reports the associated environmental condition to a majority voter component. If the majority of the sensors reports high, a shut down command is sent to the subsystem. If any subsystem reports a shut down command, the top level system will shut down. Pressure, radiation, and temperature all have associated thresholds for high values which we refer to as T_p, T_r, and T_t respectively. The safety property P of interest in this system is: *shut down if and only if any of the thresholds are surpassed* and is reflected by the shut down command at the top level::

$$shutdown = (Env\_Temp > T_t) \vee (Env\_Pressure > T_p) \vee (Env\_Radiation > T_r)$$

For reference throughout this paper, we provide Figure 26 which shows the guarantees and faults of interest for this running example. We do not show all guarantees and assumptions that are in the model, but only the ones of interest for the illustration.

7.3 Formalization

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

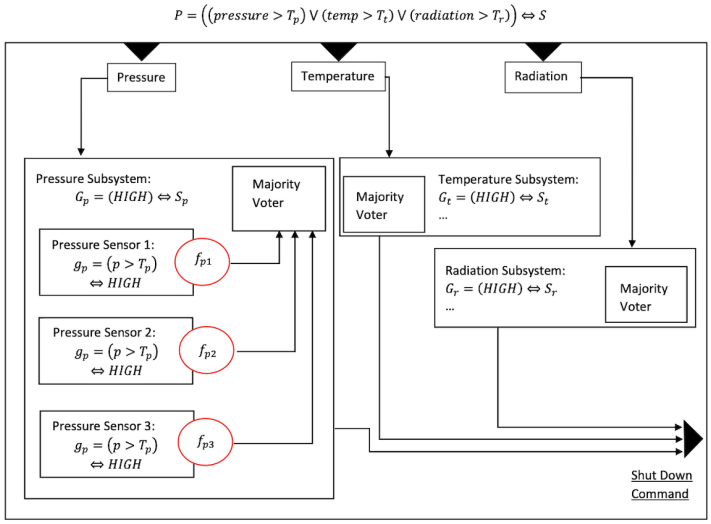


Figure 26: Sensor System Nominal and Fault Model Details

following formulas:

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

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 $(I, T) \vdash P$. We assume the transition relation has the structure of a top level conjunction. Given $T(u, u') = T_1(u, u') \wedge \cdots \wedge T_n(u, u')$ we will write $T = \wedge_{i=1..n} T_i$ for short. By further abuse of notation, T is identified with the set of its top-level conjuncts $\wedge_{i=1..n} T_i$. Thus, $T_i \in T$ means that T_i is a top-level conjunct of T, and $S \subseteq T$ means all top level conjuncts of S are top-level conjuncts of T.

The set of all nominal guarantees of the system G consists of conjunctive constraints $g \in G$. Given no faults (i.e., nominal system) and a transition relation T consisting of conjunctive constraints T_i, each g is one of the transition constraints T_i where:

$$T = g_1 \wedge g_2 \wedge \cdots \wedge g_n \tag{1}$$

We consider an arbitrary layer of analysis of the architecture and assume the property holds of the nominal relation $(I, T) \vdash P$. Let the set of all faults in the system be denoted as F. A fault $f \in F$ is a modification of the nominal constraint imposed by a guarantee. 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.

We extend the transition system so that we can view the system behavior in the presence of faults—or equivalently the absence of nominal constraints. 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 $gf_i \in GF$.

$$gf_i = \textit{if } af_i \textit{ then } f_i \textit{ else } g_i$$

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:

$$T' = gf_1 \wedge gf_2 \wedge \cdots \wedge gf_n \wedge \neg af_1 \wedge \neg af_2 \wedge \cdots \wedge \neg af_n \tag{2}$$

Theorem 1. *If $(I, T) \vdash P$ for T defined in equation 1, then $(I, T') \vdash P$ for T' defined in equation 2.*

Proof. By the mapping of each constrained activation literal $\neg af_i$ to the associated guarantee g_i and the constraint of the activation literals to be false, 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 an SMT solver for satisfiability during the model checking engine procedures.

If the $af_i \in AF$ defined in T' are unconstrained, this allows more behaviors to the transition system and could cause a violation of P. If so, a counterexample may be produced. For each counterexample, we can partition AF into two sets that we call *non-faulty variables (NFV)* and *faulty variables (FV)*. The set NFV consists of a set of activation literals that are constrained to be false throughout the counterexample, and FV contains those that can be non-deterministically assigned any valuation at some point in the trace. By mapping some of the variables in AF to false, we know that their associated guarantees in GF are non-faulty for all considered executions. We define $T'(NFV)$ as a relaxation of T' (2):

$$T'(NFV) = gf_1 \wedge gf_2 \wedge \cdots \wedge gf_n \wedge \bigwedge \{\neg af_i | af_i \in NFV\}$$

The activation literals constrained to be false in $T'(NFV)$ indicate that their associated guarantees to be valid. In the remainder of this section, we assume that all $af_i \in AF$ are unconstrained and when given a true valuation will lead to a violation of the associated guarantee. This violation causes the output that the guarantee constrains to become non-deterministic. The Boolean variables in FV correspond to Boolean variables in the fault tree.

Definition 1. *A fault tree FT is a pair $(r, \mathcal{L})$ where:*

r: the root r is a negated desirable property,

$\mathcal{L}$: a Boolean equation whose literals are faulty variables.

All literals af of the Boolean equation $\mathcal{L}$ are elements of the set FV. A fault tree may correspond to a single layer of the system architecture where the root r is a violated guarantee or a violated safety property depending on the parent component under analysis. The tree may also describe the relationship between faults and multiple layers of the system architecture. The root r still corresponds to a violated guarantee or property, but the structure of the Boolean formula $\mathcal{L}$ will reflect the layers of the system architecture. If r is a violated safety property, then $r \in P$. If r is a violated guarantee for some lower level parent component, then $r \in \pi$, where π is the set of parent component guarantees.

Definition 2. *A fault tree $FT = (r, \mathcal{L})$ is valid if and only if a true valuation for r and for all $af \in \mathcal{L}$ is satisfiable given the respective transition system constraints.*

The hierarchy of the fault tree is dependent on the associated Boolean formula. A more intuitive structure is that of *disjunctive normal form* (DNF) as seen in both fault trees depicted in Figure 27, but DNF is not required under our definition of a fault tree.

Traditionally, a safety property is a property of the system and in the assume-guarantee reasoning environment is a top level guarantee. In the following formalism, each layer of analysis is viewed as distinct from the system hierarchy as the proof is being constructed, and the properties we wish to prove are guarantees of a component. We use the notation P to refer to the set of all parent properties at a given layer of analysis. If the analysis is being performed at the top level, these are all safety properties of the system. If the analysis is being performed at an intermediate level, these are all guarantees of the parent component.

A goal of compositional safety analysis is to reflect failures of leaf and intermediate components at the top level. Not all guarantees must be valid to prove a parent level guarantee. To this end, we wish to make a distinction between all guarantees of a component and those that are required to prove parent guarantees. The subset π of P are the guarantees that must be valid to prove the guarantees of a parent component. These are the critical guarantees of a component.

Given that there may be multiple safety properties and multiple intermediate level guarantees, we do not compose single fault trees per layer, but rather forests of trees.

Definition 3. *A fault forest FF is a set of fault trees.*

Definition 4. *A fault forest FF is valid if and only if for all $FT \in FF$, the fault tree FT is valid as per Definition 2.*

The goal of this formalization is to show that the composition of fault forests results in a valid fault forest. First, we assume we can derive all minimal counterexamples to the proof of a property (or guarantee) at any layer of compositional assume-guarantee analysis. Then we prove that after composition, the tree we obtain is a fault tree describing the system in the presence of faults. In Section 7.4, we discharge the assumption and show how we derive a valid fault forest for each layer of analysis. Since a fault forest is only valid with respect to the transition system from whence it came, we will now iteratively extend the model with each composition step.

To prove each parent component guarantee $\pi_i \in \pi$, a certain subset of child guarantees are required to be non-faulty, i.e., the associated activation literals are given a

false valuation. We use the set NFV to denote the non-faulty variables of the children components that are required to prove parent guarantees π. These non-faulty variables are used in the relaxation of T' (Equation 2). This can be stated as $(I, T'(NFV)) \vdash \pi$.

The violation of certain child guarantees may lead to the violation of a parent guarantee π_i. The activation literals of the child are given a true valuation and are denoted as FV: faulty variables. A set of faulty variables of the children components contain the activation literals that correspond to leaves of a fault tree $\mathcal{L}$ with the root $r = \neg\pi_i$ for parent guarantee π_i. In other words, the fault tree $FT_i \in FF$ is associated with a property π_i. The non-faulty variables NFV contain the valid child guarantees that are required to prove π_i, and the fault tree FT_i reflects the child guarantee violations that may lead to the violation of π_i.

Definition 5. *A component is the tuple $Comp(M, FF, NFV, \pi)$ where:*

- *M: the model consisting of the set of all children properties P_c extended with non-deterministic faults: $gf_i \in P_c$ where $gf_i =$ if af_i then f_i else g_i,*
- *FF: the ordered set of fault trees for this component,*
- *NFV: the set of non-faulty variables, $NFV \subseteq P_c$,*
- *π: the ordered set of properties $\pi \subseteq P$ such that $(I, T'(NFV)) \vdash \pi$, i.e., all properties π hold if the variables in NFV are given a true valuation.*

and $FT_i \in FF$ corresponds to $\pi_i \in \pi$ for each of the i properties: the root of FT_i is $\neg\pi_i$.

Given the definition of a component, we now discuss what it means to compose components. Each layer of composition moves iteratively closer to a monolithic model by the enlargement of each set described in a component. To begin this iterative process, we define the composition of fault forests.

To show that the composition of fault trees results in a valid fault tree, let ϕ be a function $\phi : B \times B \to B$ for Boolean equations B. We use this mapping to define the composition of parent component fault tree FT_p and child component fault tree FT_c, where $FT_c = (r_c, \mathcal{L}_c)$ and $FT_p = (r_p, \mathcal{L}_p)$.

$$FT_c \circ FT_p = \phi(FT_c, FT_p) = \begin{cases} (r_p, \mathcal{L}_p(r_c, \mathcal{L}_c)) & r_c \in \mathcal{L}_p \\ (r_p, \mathcal{L}_p) & r_c \notin \mathcal{L}_p \end{cases} \tag{3}$$

where $\mathcal{L}_p(r_c, \mathcal{L}_c)$ is the replacement of af_{r_c} in $\mathcal{L}_p$ with $(r_c, \mathcal{L}_c)$. Intuitively, each of the violated guarantees has an associated activation literal. If an activation literal is found in the parent leaf equation $\mathcal{L}_p$, replace that activation literal (af_{r_c}) with the associated violated child guarantee (r_c).

Let n be the number of properties for some parent component p and let m be the number of properties for some child component c. Then the parent fault forest FF_p is a mapping $FF_p : S_1 \to B$ for $S_1 = \{1, 2, \ldots, m\}$ and the set of Boolean equations B and $FF_c : S_2 \to B$ for $S_2 = \{1, 2, \ldots n\}$.

Let ϕ_F be a function $\phi_F : seq(B) \times seq(B) \rightarrow seq(B)$ for finite sequences of Boolean equations $seq(B)$. We use this function to define the composition of parent and child component fault forests $FF_p = \{(r_{p1}, \mathcal{L}_{p1}), \ldots, (r_{pm}, \mathcal{L}_{pm})\}$ and $FF_c = \{(r_{c1}, \mathcal{L}_{c1}), \ldots, (r_{cn}, \mathcal{L}_{cn})\}$. ϕ_F is a mapping such that for all $i \in S_1$ and for all $j \in S_2$:

$$FF_c \circ FF_p = \phi_F(FF_c, FF_p) = \begin{cases} (r_{pi}, \mathcal{L}_{pi}(r_{cj}, \mathcal{L}_{cj})) & r_{cj} \in \mathcal{L}_{pi} \\ (r_{pi}, \mathcal{L}_{pi}) & r_{cj} \notin \mathcal{L}_{pi} \end{cases} \tag{4}$$

where $\mathcal{L}_{pi}(r_{cj}, \mathcal{L}_{cj})$ is the replacement of $af_{r_c ij}$ in $\mathcal{L}_p i$ with $(r_c j, \mathcal{L}_c j)$.

Each literal in the formula $\mathcal{L}_p$ is a fault activation literal af_i. If af_i has its associated guarantee gf_i in the set of child roots r_c, then the mapping ϕ_F will extend af_i in $\mathcal{L}_p$ with the leaf formula of the child root gf_i. The resulting fault forest is a sequence of fault trees $FF = \{(r_{pk}, \mathcal{L}_k) : k = 1, \ldots, m\}$. The roots of the resulting forest are the same roots as the parent forest while the leaf formulae may change based on replacement.

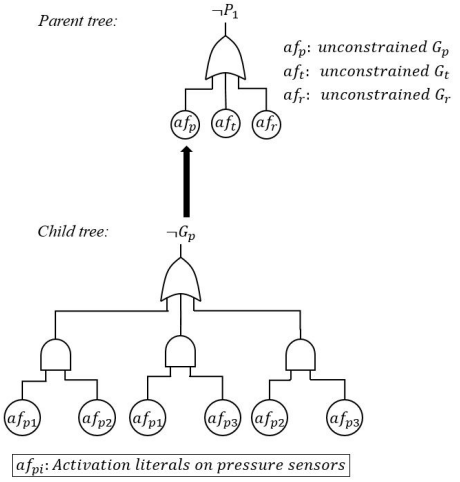


Figure 27: Sensor System Composition of Fault Trees

We return to the sensor system example to illustrate this mapping. Graphically, this is represented in Figure 27. The top level (parent) component is defined as: $Comp_p(M_p, FF_p, NFV_p, \pi_p)$ and $FF_p = \{(\neg P, af_p \vee af_t \vee af_r)\}$ where each activation literal is associated with the unconstrained guarantees G_p, G_t, and G_r. The child layer has a fault forest consisting of three fault trees, one for each subsystem.

The pressure subsystem fault tree is $FT_p = (\neg G_p, (af_{p1} \wedge af_{p2}) \vee (af_{p1} \wedge af_{p3}) \vee (af_{p2} \wedge af_{p3})$. The leaf formulae for each subsystem tree corresponds to pairwise combinations of active sensor faults. We now show the composition of the pressure subsystem child and top level parent fault trees.

The mapping ϕ_F iterates through each tree in the parent forest – in this case, we have only one. Then for each parent tree it iterates through the Boolean literals in $\mathcal{L}$.

If there is a match between a child root and a parent leaf, the replacement is made. We represent the unconstrained (violated) guarantee as $\neg G_p$ and it is associated with the fault activation literal af_p. Thus, af_p will be extended with $\{\neg G_p, (af_{p1} \wedge af_{p2}) \vee (af_{p1} \wedge af_{p3}) \vee (af_{p2} \wedge af_{p3})\}$. This extension is done for each leaf formula in $\mathcal{L}_p$ from the parent fault forest. The end result of the replacement is easy to see in Figure 27.

We have provided the foundational definitions necessary to discuss what it means to compose components. The composition of child component $Comp_c$ and parent component $Comp_p$ is defined as:

Definition 6. $Comp_c(M_c, FF_c, NFV_c, \pi_c) \circ Comp_p(M_p, FT_p, NFV_p, \pi_p)$ $= Comp_\circ(M', FF', NFV', \pi')$ *where:*

- $M' = M_c \cup M_p$ *is the iterative enlargement of the model by combining children guarantees with parent guarantees,*
- $FF_c \circ FF_p$ *is the composed fault forest,*
- $NFV' = NFV_c \cup NFV_p$ *is the set of non-faulty variables,*
- $\pi' = \pi_c \cup \pi_p$ *are valid properties such that* $(I, T'(NFV')) \vdash \pi'$.

The enlargement of the model, M', iteratively flattens the composed layers by taking the union of children guarantees and parent guarantees. The fault forests are composed into a set of fault trees describing the enlarged model. The non-faulty variables from child and parent are combined into a set NFV' such that $(I, T'(NFV')) \vdash \pi'$.

Given that in child and parent components, the properties π can be derived from the non-faulty variables, we show that this relationship holds after composition. To state $(I, T'(NFV)) \vdash \pi$, we use the shorthand $NFV \vdash \pi$.

Theorem 2. *If* $NFV_c \vdash \pi_c$ *and* $NFV_p \vdash \pi_p$*, then* $NFV' \vdash \pi'$

Proof. Assume antecedent. Let $p' \in \pi'$. If $p' \in \pi_c$ then $NFV_c \vdash p'$ and likewise if $p' \in \pi_p$, then $NFV_p \vdash p'$. In either case, $NFV_c \cup NFV_p = NFV' \vdash \pi'$. □

We work under the *monotonicity assumption*, commonly adopted in safety analysis, that an additional fault cannot cancel the effect of existing faults. Given Definition 2, we show that the composition of two fault trees results in a valid fault tree. We will then extend this to show that the composition of two fault forests results in a valid fault forest.

Lemma 1. *If* FT_c *and* FT_p *are valid fault trees, then their composition* $\phi(FT_c, FT_p)$ *is also a valid fault tree.*

Proof. Assume the antecedent. Then $(r_c, \mathcal{L}_c)$ is satisfiable with regard to the child component transition system and all $af \in \mathcal{L}_c$ and r_c are given true valuations.

Case 1: If the child root $\neg g_i$ does not have an associated $af_i \in \mathcal{L}_p$, then $\phi(FT_c, FT_p) = FT_p$ and the inclusion of the additional constraints from the child transition system in M_c does not negate the effects of the faults in FT_p. Thus, it is a valid fault tree.

Case 2: If the child root $\neg g_i$ has an associated $af_i \in \mathcal{L}_p$, then af_i has a true valuation. Given the mapping defined between guarantees and activation literals, replacement of $af_i \in \mathcal{L}_p$ with $\neg g_i$ preserves satisfiability. Furthermore, by the monotonicity assumption, the addition of more constraints ($af \in \mathcal{L}_c$) to the Boolean formula does not change satisfiability in the extended transition system.

In all cases, $\phi(FT_c, FT_p)$ is a valid fault tree. □

Lemma 2. *If FF_c and FF_p are valid fault forests, then their composition $\phi(FF_c, FF_p)$ is also a valid fault forest.*

Proof. Assume the antecedent. Then for all $FT_j \in FF_p$ and $FT_i \in FF_c$, FT_i and FT_j are valid fault trees as per Definition 4. For each iteration defined in the mapping ϕ_F, apply Lemma 1 and the monotonicity assumption.

□

We have shown that a single layer of composition produces valid fault forests. To perform this analysis across n layers of architecture we use induction to show that the resulting fault forest is valid.

The notation ϕ_F^n indicates the iterated function ϕ_F which is a successive application of ϕ_F with itself n times. Assume the fault forest FF_0 is obtained at the leaf level of the architecture.

Theorem 3. *If $\phi_F^n(FF_{n-1}, FF_n)$ is a valid fault forest, then $\phi^{n+1}(FF_n, FF_{n+1})$ is a valid fault forest.*

Proof. Base case: Each fault forest per layer is valid by construction. By Lemma 2, $\phi_F(FF_0, FF_1)$ is a valid fault forest.

Inductive assumption: Assume $\phi_F^n(FF_{n-1}, FF_n)$ is a valid fault forest.

$$\begin{aligned}\phi_F^{n+1}(FF_n, FF_{n+1}) &= ((FF_0 \circ FF_1) \circ FF_2) \circ \cdots \circ FF_n) \circ FF_{n+1})) \\ &= \phi_F^n(FF_{n-1}, FF_n) \circ FF_{n+1}\end{aligned}$$

By inductive assumption and Lemma 2, $\phi_F^{n+1}(FF_n, FF_{n+1})$ is a valid fault forest.

□

In this section, we have formalized the idea that fault trees (and forests) can be composed without losing the validity of each composed tree. We proved that this can be performed iteratively across an arbitrary number of layers. Now that we have the formal foundations laid, we proceed towards the implementation.

7.4 Implementation of the Formalism

To implement the formalism described in Section 7.3, we must compute minimal cut sets per layer of analysis, transform them into their related Boolean formula, and compose them. As previously described, Ghassabani et al. developed the *all minimal inductive validity core* algorithm (`All_MIVCs`) [39, 40]. The `All_MIVCs` algorithm 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 for the property. Thus, if we violate at least one contract from every MIVC set, we have in essence "broken" every proof. The idea is that the hitting sets of all MIVCs produces the minimal cut sets.

Next we outline the formal background and toolsuite used in the implementation and then describe the algorithm that is implemented in the safety annex for AADL.

7.5 Formal Background

JKind is an open-source industrial infinite-state inductive model checker for safety properties [37]. Models and properties in JKind are specified in Lustre [42], 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. The JKind model checker uses k-*induction* which unrolls the property P over k steps of the transition system.

Each step of induction is sent to an SMT (Satisfiability Modulo Theory)-solver to check for *satisfiability*, i.e., there exists a total truth assignment to a given formula that evaluates to true. If there does not exist such an assignment, the formula is considered *unsatisfiable*. 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 [33]. Expression of the base and induction steps of a temporal induction proof as SAT problems is straightforward and is shown below for step k:

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

When proving correctness it is shown that the formulas are *unsatisfiable*, i.e., the property P is provable. The idea behind finding an *inductive validity core* (IVC) for a given property P is based on inductive proof methods used in SMT-based model checking, such as k-induction and IC3/PDR [32, 48]. Generally, an IVC computation technique aims to determine, for any subset $S \subseteq T$, whether P is provable by S. A minimal subset that satisfies P is seen as a minimal proof explanation and called a minimal inductive validity core.

Definition 7. *Inductive Validity Core (IVC) [39]:* $S \subseteq T$ *for* $(I, T) \vdash P$ *is an Inductive Validity Core, denoted by* $IVC(P, S)$*, iff* $(I, S) \vdash P$*.*

Definition 8. *Minimal Inductive Validity Core (MIVC) [40]:* $S \subseteq T$ *is a minimal Inductive Validity Core, denoted by* $MIVC(P, S)$*, iff* $IVC(P, S) \wedge \forall T_i \in S.\ (I, S \setminus \{T_i\}) \nvdash P$*.*

The *constraint system* consists of the constrained formulas of the transition system and the negation of the property. The `All_MIVCs` algorithm collects all *minimal unsatisfiable subsets* (MUSs) of a constraint system generated from a transition system at each induction step [4, 40].

Definition 9. *A Minimal Unsatisfiable Subset (MUS) M 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 infeasibility of this system is the set $\{G_p, G_t, G_r, \}$. If all three guarantees hold, then P (the disjunction of these guarantees) is provable.

In the case of an UNSAT system, we may ask: what will correct this unsatisfiability? A related set answers this question:

Definition 10. *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 $\forall M' \subset M : C \setminus M'$ is unsatisfiable.*

An MCS can be seen to "correct" the infeasibility of the constraint system by the removal from C the constraints found in an MCS. Returning to the PWR example, the MCSs of the constraint system C are $MCS_1 = \{G_t\}$, $MCS_2 = \{G_p\}$, $MCS_3 = \{G_r\}$. If any single guarantee is violated, a shut down from that subsystem may not get sent when it should and the safety property P will be violated. This corresponds exactly to the definition of a minimal cut set.

For the following definitions, we remind readers of the extended transition system defined in Equation 2 of Section 7.3 and that the elements of T' are the set $GF \cup AF$ for potentially faulty guarantees GF and activation literals AF. We use the notation $af \rightarrow \{true, false\}$ to indicate a constraint on the literal af.

Definition 11. *Given a constraint system C, a cut set S of a top level event $\neg P$ is a set $S \subseteq AF \subseteq C$ such that $\forall af \in S$, $af \rightarrow \{true\}$ and $S \cup \{\neg P\}$ is satisfiable in C.*

Intuitively, a cut set is a true valuation for some subset of fault activation literals within a constraint system containing such that the constraint system is satisfiable given those true valuations and the violation of a safety property.

Definition 12. *A cut set S is minimal if and only if $\forall af \in S$, $S \setminus \{af\} \cup \{\neg P\}$ is unsatisfiable.*

Our approach in computing minimal cut sets through the use of inductive validity cores is to supply activation literals constrained to be false to the algorithm. The resulting MCSs consist of elements $\neg af_i$. The removal of this constraint from the constraint system results in non-deterministically true activation literals. By the definition of an MCS, we know that $C \setminus MCS$ is satisfiable. This removal of constraints from C removes the *false* constraint from each element in the MCS. Liffiton et. al showed that any subset of a satisfiable set is also satisfiable [51], so we know that for set S

consisting of elements of MCS with constraints removed, $S \cup \{\neg P\}$ is also satisfiable. This is the definition of a cut set. Minimality comes directly from the definition of a minimal correction set.

A duality exists between the MUSs of a constraint system and the MCSs as established by Reiter [63]. This duality is defined in terms of *Minimal Hitting Sets* (*MHS*).

Definition 13. *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. This is noted in previous work [30, 51] and the proof of such is given by Reiter (Theorem 4.4 and Corollary 4.5) [63].

7.6 Toolsuite Used for Implementation

Architecture Analysis and Design Language We are using the Architectural Analysis and Design Language (AADL) to construct system architecture models of performance-critical, embedded, real-time systems [2]. Language annexes to AADL provide a rich set of modeling elements for various system design and analysis needs, and the language definition is sufficiently rigorous to support formal analysis tools that allow for early fault detection.

Assume Guarantee Reasoning Environment The Assume Guarantee Reasoning Environment (AGREE) is a tool for formal analysis of behaviors in AADL models and supports compositional verification [29]. It is implemented as an AADL annex and is used to annotate AADL components with formal behavioral contracts. Each component's contracts includes assumptions and guarantees about the component's inputs and outputs respectively. AGREE translates an AADL model and the behavioral contracts into Lustre [42] and then queries the `JKind` model checker to conduct the back-end analysis [37].

Safety Annex for AADL The Safety Annex for AADL provides the ability to reason about faults and faulty component behaviors in AADL models [68, 72]. In the safety annex approach, AGREE is used to define the nominal behavior of system components, faults are introduced into the nominal model, and the JKind model checker is used to analyze the behavior of the system in the presence of faults.

7.7 Algorithm Implementation in the Safety Annex

In the formalism, any guarantee in the model had an associated fault activation literal and could be unconstrained. In the implementation, we rely on the fault model created in the safety annex to dictate which output constraints are modified (i.e., which guarantees can be violated) and how they are modified. A user may define multiple, single, or no faults on a single output. Each explicit fault defined in the safety annex is added to the Lustre program as are associated fault activation literals [68, 72]. This corresponds to the f_i and af_i described in Section 7.3.

The `All_MIVCs` algorithm requires specific equations in the Lustre model to be flagged for consideration in the analysis; these we call *IVC algorithm elements*. All

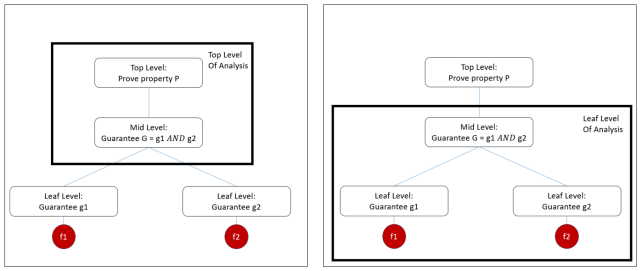


Figure 28: Illustration of Two Layers of Analysis

equations in the model can be used as IVC algorithm elements or one can specify directly the equations to consider. In this implementation, the IVC algorithm elements are added differently depending on the layer. In the leaf architectural level, fault activation literals are added to the IVC algorithm elements and are constrained to *false*. In middle or top layers, supporting guarantees are added. This is shown in Figure 28.

The figure shows an arbitrary architecture with two analysis layers: top and leaf. The top layer analysis adds G as IVC algorithm element; the leaf layer analysis adds f_1 and f_2.

A requirement of the hitting set algorithm is that to find all MCSs, all MUSs must be known. Ghassabani et al. [40] showed that finding all MIVCs is as hard as model checking. It is a requirement of this analysis that all MIVCs are found. Once the MIVC analysis is complete for a property at a given layer, a hitting set algorithm is used to generate the related MCSs [38]. Depending on the layer of analysis, the MCSs contain either guarantees (mid layer) or fault activation literals (leaf layer).

Algorithm 2: Compose Results

```
1  R ← All_MCSs(P) = ∨_{i=1}^{n} MCS_i
2  where MCS_i = ∧_{j=1}^{m} gf_j
3  Function resolve(R):
4      for ∀ OR-node in R do
5          for ∀gf_j in OR-node do
6              if ∃MCS(gf_j) then
7                  R ← replace gf_j in R with All_MCSs(gf_j);
8                  resolve (All_MCSs(gf_j));
9              else
10                 R ← replace gf_j in R with af_j;
11     convert R to DNF
```

The composition of these results is performed top down and shown in Algorithm 2. For each guarantee found in an MCS, a replacement is made with the guarantee's own MCSs. This is done recursively until all replacements have been made (line 7, 8 of Algorithm 2). If on the other hand there are no MCSs for a given guarantee, that

guarantee is replaced by its associated fault activation literal (line 10). At the leaf level of analysis, no guarantees have associated MCSs (there are no children properties) and thus reaches the end of recursion. At that time, the formula is converted back into disjunctive normal form of fault activation literals to finish the translation into the traditional fault tree (line 11).

Theorem 4. *Algorithm 2 terminates*

Proof. No infinite sets are generated by the `All_MIVCs` or minimal hitting set algorithms [40, 56]; therefore, for all g_i in the model, `All_MCSs`(g_i) is a finite set and $MCS(g_i)$ is a finite set. Each call to `Resolve` processes a guarantee that was not previously resolved, and for all g_i at the leaf layer of analysis, `All_MCSs`$((g_i) = \emptyset$. Given that there are finite layers in a model, the algorithm terminates. □

Given that the growth of the DNF formula can grow quite quickly in the worst case, we implemented strategies to prune the size of the intermediate fault trees.

7.8 Pruning to Address Scalability

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 explosion in the size of minimal cut sets for larger models.

Guarantees with no associated faults If a guarantee is found in a minimal correction set and no fault has been defined in the model that can violate it, this minimal correction set (and hence the entire subtree) is pruned.

Max n faults analysis The max n fault hypothesis in the safety annex restricts the number of faults that can be independently active simultaneously. This statement restricts the cardinality of minimal cut sets generated to n. If the number of elements in an MCS exceeds the threshold of the hypothesis statement, that MCS is eliminated from consideration and its subtree is pruned.

Probabilistic analysis pruning A probabilistic hypothesis statement restricts the cut sets by use of a probabilistic threshold. Assuming independence between faults, any cut sets with combined probability higher than the given probabilistic threshold are removed from consideration. The allowable combinations of faults are calculated before Algorithm 2 begins; this allows for dynamic pruning of minimal correction sets. If the fault activation literals within an MCS are not a subset of any allowable combination, that MCS is pruned from the formula.

To access the algorithm implementation or example models, see the repository [69].

8 Related Work

A model-based approach for safety analysis was proposed by Joshi et. al in [45–47]. 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) [36] and HiP-HOPS for EAST-ADL [26] are *FLM*-based *ESM* approaches. As previously discussed, given many possible faults, these propagation relationships require substantial user effort and become more complex. In addition, it becomes the analyst's responsibility to determine whether faults can propagate; missing propagations lead to unsound analyses. In our Safety Annex, propagations occur through system behaviors (defined by the nominal contracts) with no additional user effort.

Closely related to our work is the model-based safety assessment toolset called COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [14]. 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 [15, 20]. 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 [27, 57], MRMC (Markov Reward Model Checker) [49, 55], and RAT (Requirements Analysis Tool) [61]. The safety analysis tool xSAP [10] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [11]. COMPASS is an impressive tool suite, but some of the features that make AADL suitable for SW/HW architecture specification: event and event-data ports, threads, and processes, appear to be missing, which means that the SLIM language may not be suitable as a general system design notation (ESM).

SmartIFlow [43] 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 [43]: "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) [41] is a *FEM*-based, *purpose-built*, *monolithic causal* safety analysis language. System models constructed in SAML can be 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 [27], PRISM (Probabilistic Symbolic Model Checker) [50], or the MRMC probabilistic model checker [49].

AltaRica [9,60] 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 [16]. 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 [10, 16, 22]. This approach has limitations in terms of scalability and readability of the fault trees generated. Work has been done towards mitigating these limitations by the scalable generation of readable fault trees [18].

Minimal cut sets generated by monolithic analysis look at explicitly defined faults throughout the architecture and attempt through various techniques to find the minimal violating set for a particular property. We outline some of the common monolithic approaches to minimal cut set generation in this section.

The representation of Boolean formulae as Binary Decision Diagrams (BDDs) was first formalized in the mid 1980s [25] and was extended to the representation of fault trees not many years later [62]. After this formalization, the BDD approach to FTA provided a new approach to safety analysis. The model is constructed using a BDD, then a second BDD - usually slightly restructured - is used to encode minimal cut sets. Unfortunately, due to the structure of BDDs, the worst case is exponential in size in terms of the number of variables [25, 62]. In industrial sized systems, this is not realistically useful.

SAT based computation was introduced to address scalability problems in the BDD approach; initially it was used as a preprocessing step to simplify the decision diagram [17], but later was extended to allow for all minimal cut set processing and generation without the use of BDDs [13]. Since then, much research has focused on leveraging the power of model checking in the problems of safety assessment [8, 13, 23, 67, 68, 74].

Bozzano et al. formulated a Bounded Model Checking (BMC) approach to the problem by successively approximating the cut set generation and computations to allow for an "anytime approximation" in cases when the cut sets were simply too large and numerous to find [13]. These algorithms are implemented in xSAP [10] and COMPASS [11].

The model based safety assessment tool AltaRica 3.0 [59] performs a series of processing to transform the model into a reachability graph and then compile to Boolean formula in order to compute the minimal cut sets. Other tools such as HiP-HOPS [58]

have implemented algorithms that follow the failure propagations in the model and collect information about safety related dependencies and hazards. The Safety Analysis Modeling Language (SAML) [41] provides a safety specific modeling language that can be translated into a number of input languages for model checkers in order to provide model checking support for minimal cut set generation.

To our knowledge, a fully compositional approach to generating fault forests or minimal cut sets has not been introduced.

9 Conclusion

We have developed an extension to the AADL language with tool support for formal analysis of system safety properties in the presence of faults. Faulty behavior is specified as an extension of the nominal model, allowing safety analysis and system implementation to be driven from a single common model. Both symmetric and asymmetric faulty behaviors are supported. This new Safety Annex leverages the AADL structural model and nominal behavioral specification (using the AGREE annex) to propagate faulty component behaviors without the need to add separate propagation specifications to the model. Implicit error propagation enables safety engineers to inject failures/faults at component level and assess the effect of behavioral propagation at the system level. It also supports explicit error propagation that allows safety engineers to describe dependent faults that are not easily captured using implicit error propagation. Generation of minimal cut sets collects all minimal set of fault combinations that can cause violation of the top level properties. For more details on the tool, models, and approach, see the technical report [70] and other publications from this research [68, 71, 72]. To access the tool plugin, users manual, or models, see the repository [69].

Acknowledgments. This research was funded by NASA contract NNL16AB07T and the University of Minnesota College of Science and Engineering Graduate Fellowship.

References

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

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

[3] 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.

[4] 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.

[5] S. Berezin, S. Campos, and E. M. Clarke. Compositional reasoning in model checking. In *International Symposium on Compositionality*, pages 81–102. Springer, 1997.

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

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

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

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

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

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

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

[13] M. Bozzano, A. Cimatti, A. Griggio, and C. Mattarei. Efficient anytime techniques for model-based safety analysis. In *International Conference on Computer Aided Verification*, pages 603–621. Springer, 2015.

[14] 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.

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

[16] 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.

[17] M. Bozzano, A. Cimatti, O. Lisagor, C. Mattarei, S. Mover, M. Roveri, and S. Tonetta. Safety assessment of AltaRica models via symbolic model checking. *Science of Computer Programming*, 98:464–483, 2015.

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

[19] 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.

[20] 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.

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

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

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

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

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

[26] 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.

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

[28] 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.

[29] 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.

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

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

[32] N. Eén, A. Mishchenko, and R. Brayton. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 125–134. IEEE, 2011.

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

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

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

[36] 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.

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

[38] 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.

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

[40] 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.

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

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

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

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

[45] 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.

[46] 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.

[47] 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.

[48] T. Kahsai, P.-L. Garoche, C. Tinelli, and M. Whalen. Incremental verification with mode variable invariants in state machines. In *NASA Formal Methods Symposium*, pages 388–402. Springer, 2012.

[49] 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.

[50] 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.

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

[52] 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.

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

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

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

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

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

[58] Y. Papadopoulos and M. Maruhn. Model-based synthesis of fault trees from Matlab-Simulink models. In *2001 International Conference on Dependable Systems and Networks*, pages 77–82. IEEE, 2001.

[59] T. Prosvirnova. *AltaRica 3.0: a Model-Based approach for Safety Analyses*. Theses, Ecole Polytechnique, Nov. 2014.

[60] 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.

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

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

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

[64] 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.

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

[66] SAE ARP4761. Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

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

[68] 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.

[69] D. Stewart, J. Liu, M. Whalen, D. Cofer, and M. Peterson. Safety annex for AADL repository, 2018. `https://github.com/loonwerks/AMASE`, last accessed on 2020-10-17.

[70] 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.

[71] D. Stewart, J. J. Liu, D. Cofer, M. Heimdahl, M. W. Whalen, and M. Peterson. Aadl-based safety analysis using formal methods applied to aircraft digital systems. *Reliability Engineering & System Safety*, 213:107649, 2021.

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

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

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

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
06/11/2021	CONTRACTOR REPORT	9/6/2016-9/30/2019

4. TITLE AND SUBTITLE
Architectural Modeling and Analysis for Safety Engineering

5a. CONTRACT NUMBER
NNL16AA09B

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Danielle Stewart, University of Minnesota
Jing Liu, Collins Aerospace
Darren Cofer, Collins Aerospace
Mats Heimdahl, University of Minnesota
Michael W. Whalen, University of Minnesota
Michael Peterson, Collins Aerospace

5d. PROJECT NUMBER

5e. TASK NUMBER
NNL16AB07T

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration
Washington, DC 20546-001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA/CR-20210017388

12. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified - Unlimited
Subject Category
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES

14. ABSTRACT
We describe an extension to the Architecture Analysis and Design Language (AADL) that supports modeling of system behavior under failure conditions. This Safety Annex enables independent modeling of component failures and modeling of various types of fault behavior in the nominal system model. The accompanying tool provides model checking to propagate errors from their source to their effect on safety properties. The tool also captures all minimal set of fault combinations that can cause violation of safety properties. We describe the Safety Annex, illustrate its use with a representative example, and discuss and demonstrate the tool support enabling an analyst to investigate the system behavior under failure conditions.

15. SUBJECT TERMS
Model-Based Development; MBD; Model-Based Systems Engineering; MBSE; Safety-Critical Systems; AADL; Fault Analysis; Fault Tree Analysis; FTA

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			HQ - STI-infodesk@mail.nasa.gov
U	U	U	UU	60	19b. TELEPHONE NUMBER *(Include area code)* 757-864-9658