AADL-Based Safety Analysis Using Formal Methods Applied to Aircraft Digital Systems

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
dkstewar, whalen, heimdahl@cs.umn.edu
[2] Collins Aerospace
Trusted Systems - Enterprise Engineering
Jing.Liu, Darren.Cofer@collins.com
[3] Collins Aerospace
Flight Controls Safety Engineering - Avionics
Michael.Peterson@collins.com

Abstract. Model-based engineering 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 paper 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 verify safety properties in the presence of faults and comprehensively enumerate all applicable fault combinations leading to failure conditions under quantitative objectives as part of the safety assessment process. The approach allows exploration of the effects of faulty component behavior on system level failure conditions without requiring explicit propagation specifications. It also supports a shared system model, a modeling language that can describe real-time embedded systems, and useable safety analysis artifacts.

Keywords: Model-based safety analysis; Model-based systems engineering; fault analysis; safety engineering; AADL; fault injection

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. 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 beahvior from multiple sources. Comprehensive identification of all unsafe interactions in increasingly complex software-intensive systems is also a challenge. Leveraging model-based system development in critical systems and use of a common formal model shared between system development and safety analysis holds great promise. The model-based approach can help eliminate ambiguity, promote artifact consistency, analysis accuracy, and minimize design/safety analysis iterations [8, 25, 28, 31, 32].

Model-based Safety Analysis/Assessment (MBSA) approaches have been developed for a variety of modeling languages, including SysML [21, 27, 34], Architecture Analysis and Design Language (AADL) [3, 20], SLIM [9], Simulink [29, 33]. Each language has a targeted domain of application and containts different levels of formalism. AADL, an Society of Automotive Engineering (SAE) standard modeling language for Model-based Systems Engineering (MBSE) [3], provides a more rigorous system description and run-time semantics and is well suited for modeling real-time embedded systems. AADL has sufficiently well-defined semantics to allow for formal model checking approaches which is why this language was chosen for our approach.

The approaches used in MBSA tools differ significantly. A consideration is whether errors are propagated explicitly or through behavioral modeling. Tools such as the AADL Error Model Annex, Version 2 (EMV2) [20], HiP-HOPS for EAST-ADL [16], and Ansys Medini [1] are explicit propagation approaches. Given many possible faults, these propagation relationships require substantial user effort to understand and define. In addition, missing propagations lead to unsound analyses.

Another important consideration is whether models and notations are purpose built for safety analysis or if the existing system model is extended with safety analysis information. SmartI-Flow [28], The Safety Analysis and Modeling Language (SAML) [25], and AltaRica [6, 36] are examples of such safety analysis tools. By developing the system model separate from the safety model, this requires close communication between development groups on every iteration of model development. Important changes to the system model are not automatically factored into the safety model.

In this paper we describe the *safety annex* for the system engineering language AADL. 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. This paper is an extension of a shorter conference paper that introduces the safety annex [40].

Our work can be viewed as a continuation of work conducted by Joshi et al. where they explored a behavioral approach of model-based safety analysis defined over Simulink/Stateflow models [29–31, 33]. Our current work extends and generalizes this work and provide new modeling and analysis capabilities not previously available. It also moves the analysis from a component implementation language (Simulink) to an architecture language for real-time embedded systems (AADL). The safety annex allows modeling both implicit and explicit error propagation, supports compositional verification, and provides exploration of the nominal system behavior as well as the system’s behavior under failure conditions. Our work is also related to the existing safety analysis approaches, in particular, the AADL Error Annex (EMV2) [20], COMPASS [9], and AltaRica [6, 36]. Our approach is significantly different from previous work in that unlike EVM2 we leverage the behavioral model 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 aims and objectives of this research are as follows.

- support a shared model that captures the current state of the system design as it moves through the development lifecycle, allowing all participants of the Aerospace Recommended Practices (ARP)4754A process to be able to communicate and review the system design.
- integrate behavioral fault analysis into a system modeling language with well-defined semantics,
- support behavioral specification of faults and their implicit propagation (both symmetric and asymmetric) through behavioral relationships in the model,
- use formal methods to automatically verify safety properties in the presence of faults and generate evidence of the analysis performed to meet objectives of the safety assessment process

The organization of the paper is as follows. Section 2 provides preliminary information and background, the implementation of the safety annex is discussed in Section 3, and Section 4 provides the details of using the safety annex to model a representative aircraft system. It is followed by a thorough description of the analysis of the case study in section 5. The paper ends with a discussion of related work in Section 6 and finally the conclusion.

2 Preliminaries

We are using the Architectural Analysis and Design Language (AADL) [19] 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" [3]. From its conception, AADL has been applied to the design and construction of avionics systems. Rather than being merely descriptive, AADL models can be made specific enough to support system-level code generation. Thus, results from analyses conducted, including the new safety analysis proposed here, correspond to the system that will be built from the model.

An AADL model describes a system in terms of a hierarchy of components and their interconnections, where each component can either represent a logical entity (e.g., application software functions, data) or a physical entity (e.g., buses, processors, memory). 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) [17] 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 [26] and then queries the JKind model checker [22] 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 [17].

3 Methodology

As a running example in this methodology, we introduce the Wheel Brake System (WBS) described in Aerospace Information Report (AIR) 6110 [2]. This system is a well-known example that has been used as a case study for safety analysis, formal verification, and contract based design [8, 12, 13, 29].

3.1 Wheel Brake System Overview

The preliminary work for the safety annex was based on a simple model of the WBS [42]. To demonstrate a more complex fault modeling process, we constructed a functionally and structurally equivalent AADL version of the more complex WBS which was captured in NuSMV/xSAP models [13]. Figure 1 shows only one pair of wheels and their interactions with the rest of the system for clarity. The full version that was modeled in AADL contains a total of 8 wheels.

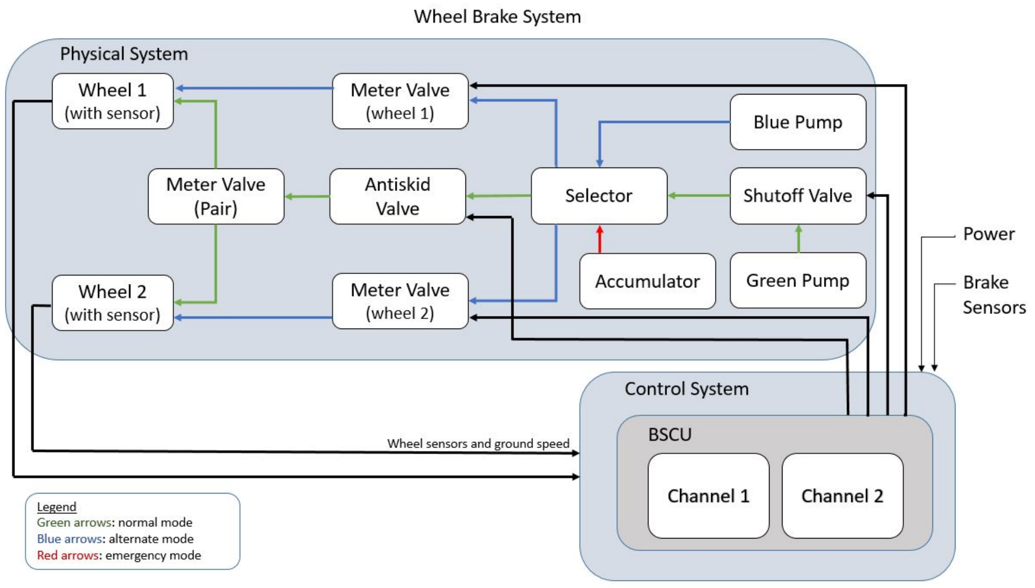


Fig. 1: A Two-Wheel Diagram of the Wheel Brake System

The WBS is composed of two main parts: the control system and the electro-mechanical physical system. The physical system consists of redundant hydraulic circuits (designated green and blue) running from hydraulic pumps to wheel brakes as well as valves that control the hydraulic fluid flow. The physical system provides braking force to each of the eight wheels of the aircraft. The wheels are all mechanically braked in pairs. The control system commands electronic control of the physical system. The Braking System Control Unit (BSCU) consists of two channels for redundancy in case a detectable fault occurs in the active channel. The BSCU also commands antiskid braking and controls the operating mode of the system through commands to the selector valve. These commands are sent to a selector valve component which selects which hydraulic pump supplies pressure depending on which operating mode the system is currently in.

Top level inputs to the system include the mechanical pedal sensors and the power. These are considered black box components. The only pilot interaction modeled in this system is through mechanical braking command.

There are three operating modes in the WBS model:

- In *normal* mode, the system uses the green hydraulic pump and one meter valve per each of the eight wheels (in Figure 1, this corresponds to e.g., ”Meter Valve (wheel 1)”. Each of the meter valves are controlled through electronic commands coming from the active channel of

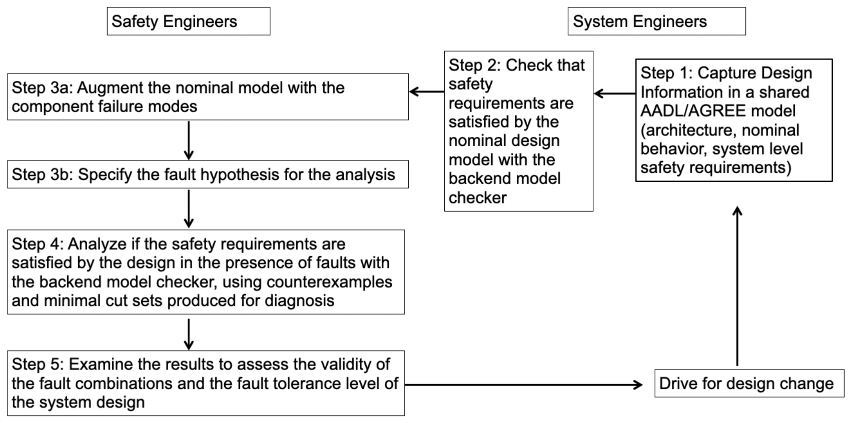

Safety Engineers
System Engineers
Step 1: Capture Design Information in a shared AADL/AGREE model (architecture, nominal behavior, system level safety requirements)
Step 2: Check that safety requirements are satisfied by the nominal design model with the backend model checker
Step 3a: Augment the nominal model with the component failure modes
Step 3b: Specify the fault hypothesis for the analysis
Step 4: Analyze if the safety requirements are satisfied by the design in the presence of faults with the backend model checker, using counterexamples and minimal cut sets produced for diagnosis
Step 5: Examine the results to assess the validity of the fault combinations and the fault tolerance level of the system design
Drive for design change

Fig. 2: Proposed Safety Assessment Process Backed by Formal Methods

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 and detection of skid.
- In *alternate* mode, the system uses the blue hydraulic pump, four meter valves (one per wheel pair as shown in Figure 1: ”Meter Valve (Pair)”), and four antiskid shutoff valves (one per wheel pair). The meter valves are mechanically commanded through the pilot pedal corresponding to each wheel pair. If the selector detects lack of pressure in the green circuit, it switches to the blue circuit. Alternatively, if the BSCU detects a fault in the normal (green) mode of operation, the BSCU may likewise shut off the green pump and force a switch to alternate (blue) mode of operation.
- *Emergency* mode is triggered when the blue hydraulic pump fails. The accumulator component 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.

3.2 Methodology Overview

We propose a model-based safety assessment process backed by formal methods to help safety engineers with early detection of design issues. This process uses a single unified model to support both system design and safety analysis. It is based on the following steps as shown in Figure 2 and outlined below.

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 verify that the nominal model supports the requirements.
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 model in the presence of faults. If the model 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 sets of fault combinations that can cause the safety requirement to be violated.
5. The safety engineers examine the results to assess the validity of the fault combinations and the fault tolerance level of the system design. If a design change is warranted, the model will be updated with the latest design change and the above process is repeated.

In the remainder of this section, we describe how the safety annex is implemented and then we describe how the steps outlined above can be achieved using the safety annex for AADL.

3.3 Implementation Overview

The safety annex is written in Java as a plug-in for the Open Source AADL Tool Environment (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 [17]. The architecture of the safety annex is shown in Figure 3.

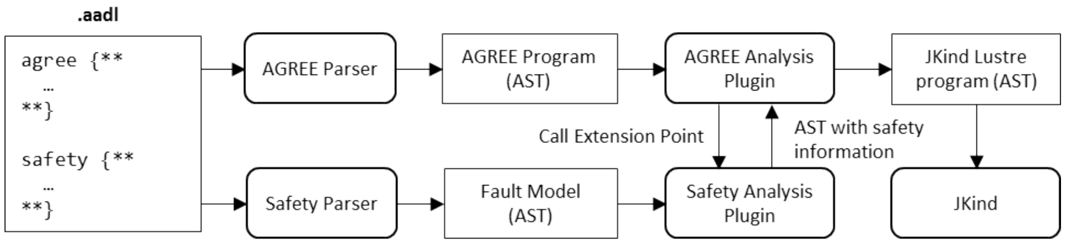
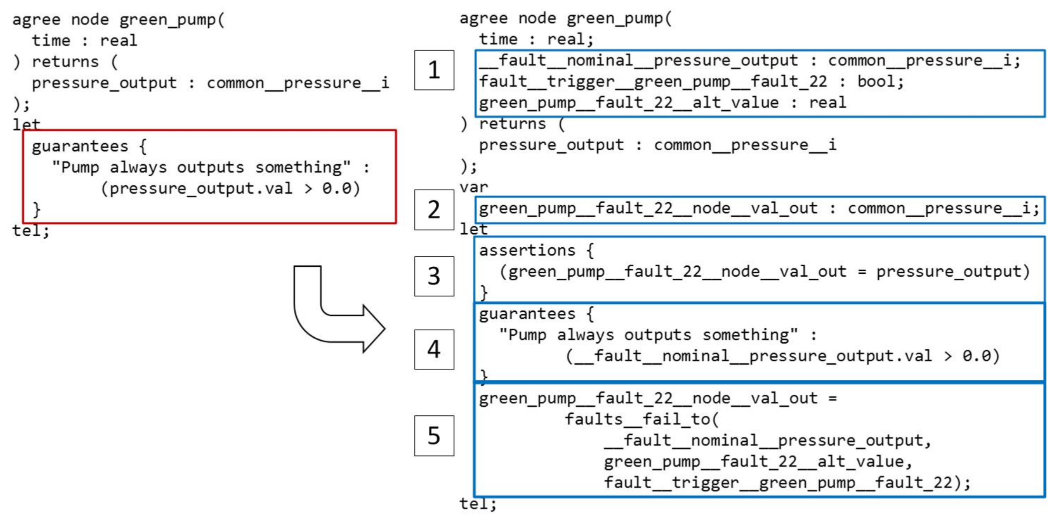


Fig. 3: Safety Annex Plug-in Architecture

AGREE contracts are used to define the nominal behaviors of system components as *guarantees* that hold when *assumptions* about the values the component's environment are met. When an AADL model is annotated with AGREE contracts and the fault model is created using the safety annex, the model is transformed through AGREE into a Lustre model [26] 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 4. The left column of the figure shows the nominal Lustre pump definition with an AGREE contract on the output. 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 [26]) follows the standard translation path to the model checker JKind [22], an infinite-state model checker for safety properties.

The Lustre formulae are represented in JKind as a transition system, and reasoning is performed using k-induction. When performing safety analysis over the model, each fault is defined as an *activation literal* and given limited constraint. If the assignment to an activation literal is *true*, this



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

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

Fig. 4: Nominal AGREE Node and Extension with Faults

corresponds to an active fault and potentially violated guarantee. If that assignment violates a guarantee, then this violation will be reflected in the analysis results. At a system level, it can be seen if a violated guarantee will in turn violate the top level property. Hence it is seen how active faults at leaf level components violate the system level properties.

This analysis approach allows for implicit propagation of violations throughout the system. It also allows for arbitrary temporal activations of faults. There are no explicit constraints put on faults stating when an activation can occur, which allows the model checking procedure free reign to activate the faults at the worst possible times. If there are dependencies regarding fault activations, these are handled through the use of explicit error propagations (see Section **??**). While the model checker may choose various permutations of fault activation, these permutations of faults in terms of exposure time and order of occurrence are not part of the minimal cut set output of this analysis.

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

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

Verification in the Presence of Faults: This analysis returns a counterexample if any guarantee or system level property is violated by active faults in the system. The counterexample shows a concrete scenario why a property is violated, with assignments to each signal in the model by the model checker, possibly over a multi-step progression. 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 sets 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 [23]. 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 [24].

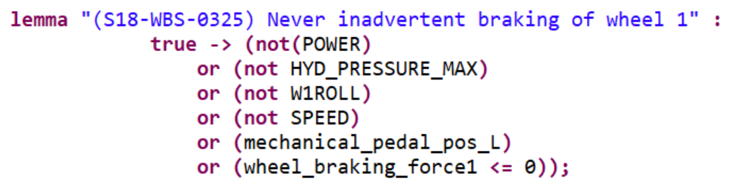
In this approach, we use the All-MIVCs algorithm to compute the minimal set of model elements (including component contracts and fault activation literals) necessary to prove the top level property, and transform them to minimal sets of faults for the violation of the top level property [39].

To access the tool plugin, user manual, or models, see the repository located at `https://github.com/loonwerks/AMASE/`.

3.4 Nominal Model Development

The system model is developed in AADL and extended with behavioral information in what we call the nominal model. The nominal, or 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 the 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: *Inadvertent wheel braking on one wheel during takeoff shall be less than 1E-9 per takeoff*. Inadvertent braking means that braking force is applied at the wheel but the pilot has not pressed the brake pedal. In addition, the inadvertent braking requires that power and hydraulic pressure are both present, the plane is not stopped, and the wheel is rolling (not skidding). The property is stated in AGREE such that inadvertent braking does *not* occur, as shown in Figure 5. (The expression shown in Figure 5 *true* $\rightarrow$ *property* in AGREE is true in the initial state and then afterwards it is only true if property holds.)



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

Fig. 5: AGREE Contract for Top Level Property: Inadvertent Braking

3.5 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 all 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 smaller problems. A more comprehensive description of these types of proofs and analyses is found in [4, 17]

The WBS has a total of 13 safety properties at the top level that are supported by subcomponent assumptions and guarantees, shown in Table 1. Since there are 8 wheels, contract S18-WBS-0325-wheelX is repeated 8 times, one for each wheel. The system includes both left (L) and right (R) side braking, so S18-WBS-R/L-0322 occurs twice. The behavioral model in total consists of 36 assumptions and 246 supporting guarantees.

Table 1: Safety Properties of WBS

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. .

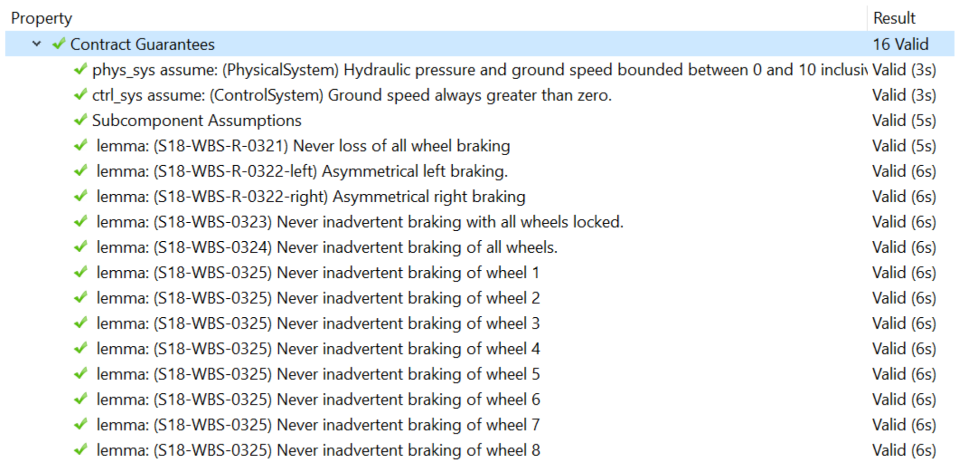
The analysis results are shown in Figure 6.

The lemmas are the specifications of all top level safety properties in the model. The results show that the model supports the specifications and a proof is found for each lemma. The child component contracts are used to prove the validity of the safety properties.

3.6 Fault Modeling

The usage of the terms error, failure, and fault are defined in ARP4754A and are described here for ease of understanding [37]. 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 potential faulty behaviors to a component model. Within the AADL component instance model, an annex is added which contain the fault definitions for the



Property	Result
✓ Contract Guarantees	16 Valid
✓ phys_sys assume: (PhysicalSystem) Hydraulic pressure and ground speed bounded between 0 and 10 inclusiv	Valid (3s)
✓ ctrl_sys assume: (ControlSystem) Ground speed always greater than zero.	Valid (3s)
✓ Subcomponent Assumptions	Valid (5s)
✓ lemma: (S18-WBS-R-0321) Never loss of all wheel braking	Valid (5s)
✓ lemma: (S18-WBS-R-0322-left) Asymmetrical left braking.	Valid (6s)
✓ lemma: (S18-WBS-R-0322-right) Asymmetrical right braking	Valid (6s)
✓ lemma: (S18-WBS-0323) Never inadvertent braking with all wheels locked.	Valid (6s)
✓ lemma: (S18-WBS-0324) Never inadvertent braking of all wheels.	Valid (6s)
✓ lemma: (S18-WBS-0325) Never inadvertent braking of wheel 1	Valid (6s)
✓ lemma: (S18-WBS-0325) Never inadvertent braking of wheel 2	Valid (6s)
✓ lemma: (S18-WBS-0325) Never inadvertent braking of wheel 3	Valid (6s)
✓ lemma: (S18-WBS-0325) Never inadvertent braking of wheel 4	Valid (6s)
✓ lemma: (S18-WBS-0325) Never inadvertent braking of wheel 5	Valid (6s)
✓ lemma: (S18-WBS-0325) Never inadvertent braking of wheel 6	Valid (6s)
✓ lemma: (S18-WBS-0325) Never inadvertent braking of wheel 7	Valid (6s)
✓ lemma: (S18-WBS-0325) Never inadvertent braking of wheel 8	Valid (6s)

Fig. 6: Nominal model analysis results for WBS

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 [41]. 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 lead to a violation of 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.

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 7 shows the AADL pedal sensor component with a contract for its nominal behavior. (The expression *true* $\rightarrow$ *property* in AGREE is true in the initial state and then afterwards it is only true if property holds.) 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.

One possible failure for the pedal sensor is inversion of its output value. This fault can be triggered with probability 5.0×10^{-6} as described in AIR6110 (in practice, the component failure probability is collected from hardware specification sheets). The safety annex definition for this fault is shown in Figure 8. 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 fault definitions and 141 fault instances. The large number of fault instances is due to the redundancy in the system design and its replication to control 8 wheels.

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

  -- Behavioral contracts for subcomponent
  annex agree {**

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

Fig. 7: 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;
  }
};
```

Fig. 8: The Safety Annex for the Pedal Sensor

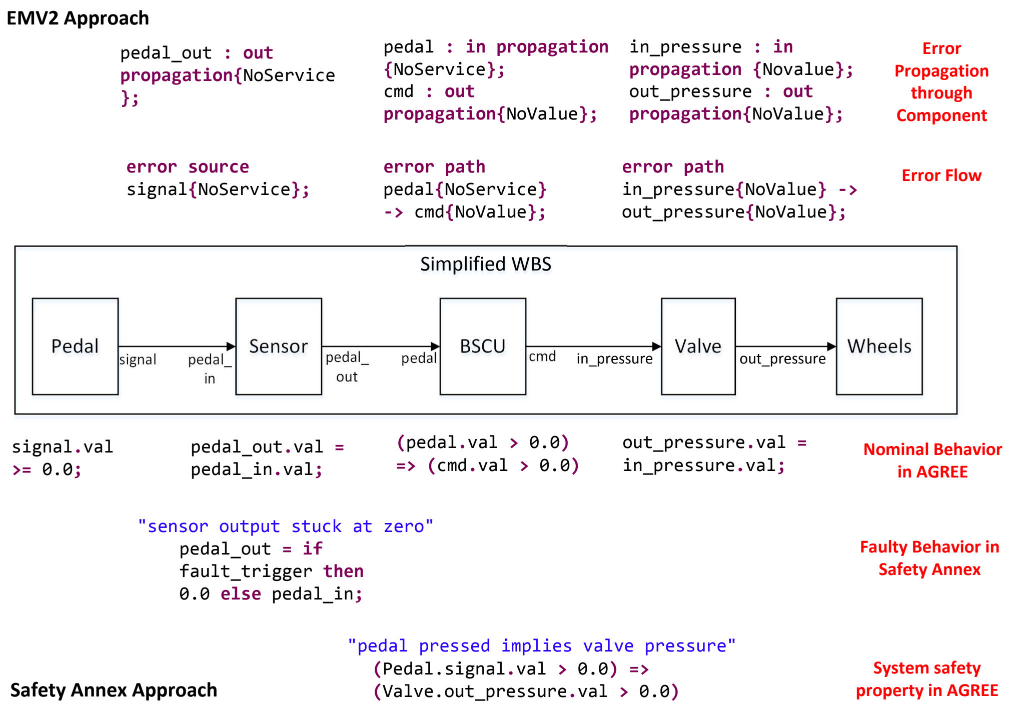
Implicit Error Propagation In this approach, faults are captured as faulty behaviors that augment the system behavioral model in AGREE contracts. No explicit error propagation is necessary since the faulty behavior propagates through the nominal behavior contracts in the system model just as in the real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

By contrast, in the AADL Error Model Annex, Version 2 (EMV2) [20] approach, all errors must be explicitly propagated through each component (by applying fault types on each of the output ports) 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 (Figure 9).

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 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 9), the "NoService" fault is explicitly propagated through all of the components. These fault types are essentially tokens rather than a specifiction of the faulty behavior. At the system level, analysis tools supporting the EMV2 annex can aggregate the propagation information from different components to compose an overall fault flow diagram or fault tree.

When a fault is triggered in the safety annex (bottom half of Figure 9), 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 signal and responds 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*.



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

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

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

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

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

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

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

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

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

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

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

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

Fig. 9: Differences between Safety Annex and EMV2

Explicit Error Propagation Failures in Hardware (HW) components can trigger behavioral faults in the system components that depend on them. For example, a Central Processing Unit (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, whether dependent or independent. The explicit propagation to non behavioral faults is similar to that provided in EMV2.

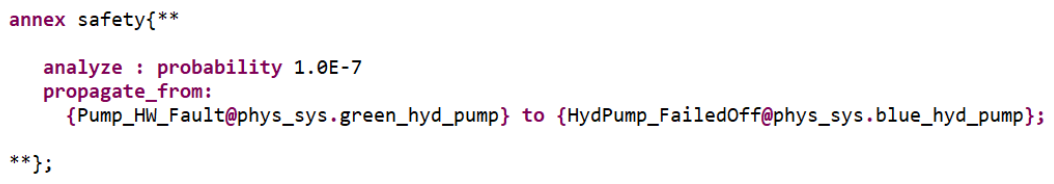
To better model faults at the system level that are dependent on HW failures, a fault model element is introduced called a *hardware fault*. Users are not required to specify behavioral effects for the HW faults, nor are data ports necessary on which to apply the fault definition. An example of a model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

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

Fig. 10: Hardware Fault Definition

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

In the WBS example, assume that both the green and blue hydraulic pumps are located in the same compartment in the aircraft and an explosion in this compartment rendered both pumps inoperable. The HW fault definition can be modeled first in the green hydraulic pump component as shown in Figure 10. The activation of this fault triggers the activation of related faults as seen in the *propagate_to* statement shown in Figure 11. 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};

**};
```

Fig. 11: Hardware Fault Propagation Statement

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

Asymmetric Faults and Implementation A *Byzantine* or *asymmetric* fault is a fault that presents different symptoms to different observers [18]. Consider a source component with an output that is connected to multiple inputs on different destination components. In this configuration, a *symmetric* fault will result in all destination components observing the same faulty value from the source component. In an *asymmetric* fault, the destination components may observe different values from the source. To capture the behavior of asymmetric faults it was necessary to extend our fault modeling mechanism in AADL.

To illustrate our implementation of asymmetric faults, assume a source component A has a 1-to-many output connected to four destination components (B-E) as shown in Figure 12 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 automatically inserted on each connection from component A to components B, C, D, and E (shown in Figure 12 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 12.

An asymmetric fault is defined for component A as in Figure 13. 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 component 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

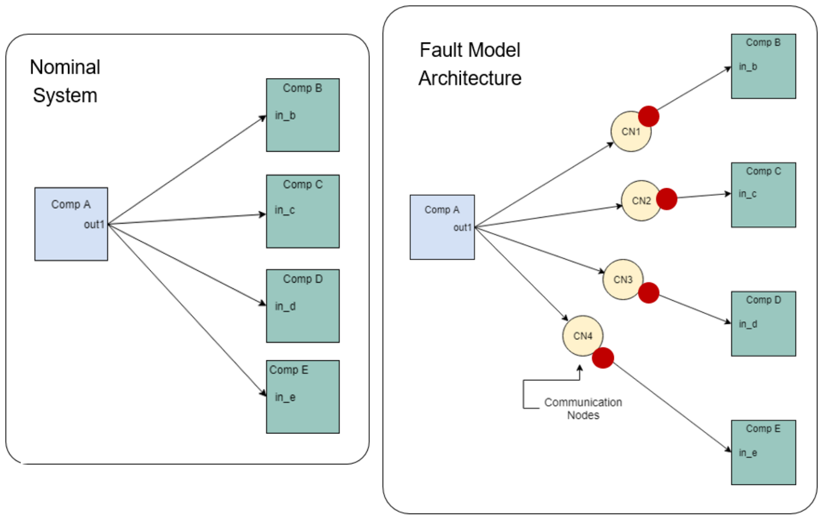


Fig. 12: Communication Nodes in Asymmetric Fault Implementation

```
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;
}
```

Fig. 13: Asymmetric Fault Definition in the Safety Annex

see an incorrect value is completely nondeterministic. Any number of the communication node faults (0. . . all) may be triggered upon activation of the main asymmetric fault on the source output.

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 the maximum number of faults that can be active at any point in execution (Figure 14).

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

Fig. 14: Max N Faults Analysis Statement

Alternatively, the fault analsis statement may specify that the only faults to be considered are those whose probability of simultaneous occurrence is above some probability threshold (Figure 15).

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 re-

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

Fig. 15: Probability Analysis Statement

stricting the cutsets to only those whose simultaneous probability is above some set value. In the former case, we assert that the sum of the true *fault__trigger* variables is at or below some integer threshold. In the latter, we determine all combinations of faults whose probabilities are above the specified probability threshold, and describe this as a proposition over *fault__trigger* variables.

With the introduction of dependent faults, active faults are divided into two categories: independently active (activated by its own triggering event) and dependently active (activated when the faults they depend on become active). The top level fault hypothesis applies to independently active faults. Faulty behaviors augment nominal behaviors whenever their corresponding faults are active (either independently active or dependently active).

3.7 Fault Model Analysis

There are two main options for fault model analysis. The first option injects faulty behavior allowed by the faulty hypothesis into the AGREE model and returns this fault annotated Lustre program to JKind for analysis. The injection of faulty behavior into the AGREE model 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 for analysis is used to generate minimal cut sets for the model. The path from the user written fault model to JKind is the same in both kinds of analysis, but the fault annotations specify which results to compute and display to the user.

Verification in the Presence of Faults: Probabilistic Analysis Given a probabilistic fault hypothesis, this corresponds to performing analysis with the combinations of faults whose simultaneous 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. This form of analysis is not performed using a probabilistic model checker, but rather probabilistic computations are performed after behavioral analysis is complete. It is assumed that the faults occur independently and possible combinations of faults are computed and passed to the Lustre model to be checked by the model checker.

Algorithm 1: Monolithic Probability Analysis

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

As seen in Algorithm 1, the computation first removes all faults from consideration that are too unlikely given the probability threshold. The remaining faults are arranged in a priority queue $\mathcal{Q}$ from high to low. Assuming independence in the set of faults, we take a fault with highest probability from the queue (step 5) and attempt to combine the remainder of the faults in $\mathcal{R}$ (step 7). If this combination is lower than the threshold (step 8), then we do not take into consideration this set of faults and instead remove the tail of the remaining faults in $\mathcal{R}$. In this calculation, we assume independence among the faults.

Generate Minimal Cut Sets: Max N Analysis 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	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0321	7	0	0	256	57,600
0322-R	75	0	0	0	0
0322-L	75	0	0	0	0
0323	182	0	0	0	0
0324	8	3,665	28,694	883,981	-
0325-WX	33	0	0	0	0

As can be seen in Table 2, the number of cut sets increases exponentially 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.

Generate Minimal Cut Sets: Probabilistic Analysis Both probabilistic analysis and max n analysis use the same underlying minimal cut set generation algorithm (see Section 3.3), but in probabilistic analysis the minimal cut sets are pruned to include only those fault combinations whose probability of simultaneous occurrence exceed the given threshold in the hypothesis.

The probabilistic analysis for the WBS was given a top level threshold per property as stated in AIR6110 and shown in Table 1. The faults associated with various components were all given probability of occurrence according to the AIR6110 document [2]. The table shows the property name and associated probability. The generation of minimal cut sets provided all sets that violate that property whose combined probabilities (assuming independence) are greater than the threshold. The number of sets per cardinality are listed in the table.

Table 3: WBS Minimal Cut Set Results for Probabilistic Hypotheses

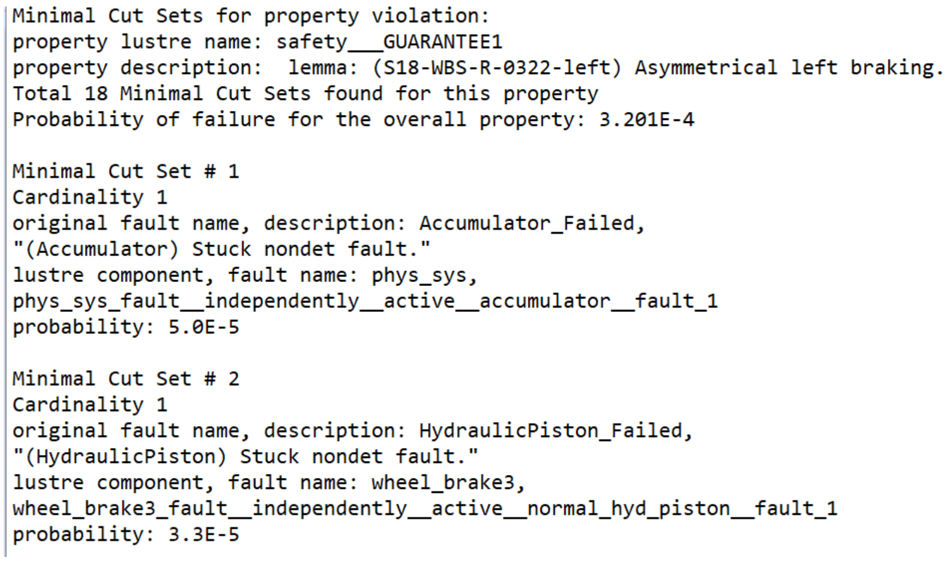
Property	$n = 1$	$n = 2$	$n = 3$	$n = 4$	$n = 5$
0321: 5.0×10^{-7}	7	0	0	256	0
0322-R: 5.0×10^{-7}	75	0	0	0	0
0322-L: 5.0×10^{-7}	75	0	0	0	0
0323: 1.0×10^{-9}	182	0	0	0	0
0324: 1.0×10^{-9}	8	3665	0	0	0
0325-W1: 1.0×10^{-9}	33	0	0	0	0

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.

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).

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

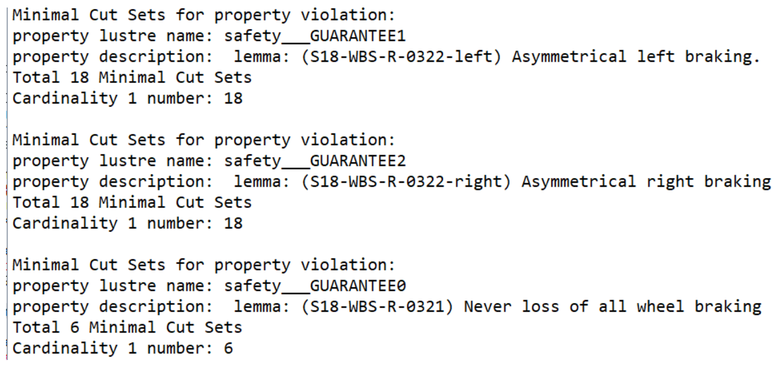
1. The minimal cut sets can be presented in text form with the total number per property, cardinality of each, and description strings showing the property and fault information. A sample of this output is shown in Figure 16.



```
Minimal Cut Sets for property violation:
property lustre name: safety___GUARANTEE1
property description:  lemma: (S18-WBS-R-0322-left) Asymmetrical left braking.
Total 18 Minimal Cut Sets found for this property
Probability of failure for the overall property: 3.201E-4

Minimal Cut Set # 1
Cardinality 1
original fault name, description: Accumulator_Failed,
"(Accumulator) Stuck nondet fault."
lustre component, fault name: phys_sys,
phys_sys_fault__independently__active__accumulator__fault_1
probability: 5.0E-5

Minimal Cut Set # 2
Cardinality 1
original fault name, description: HydraulicPiston_Failed,
"(HydraulicPiston) Stuck nondet fault."
lustre component, fault name: wheel_brake3,
wheel_brake3_fault__independently__active__normal_hyd_piston__fault_1
probability: 3.3E-5
```

Fig. 16: Detailed Output of MinCutSets

2. The minimal cut set information can be presented in tally form. This does not contain the fault information in detail, but instead gives only the tally of cut sets per property. This is useful in large models with many cut sets as it reduces the size of the text file. An example of this output type is seen in Figure 17.

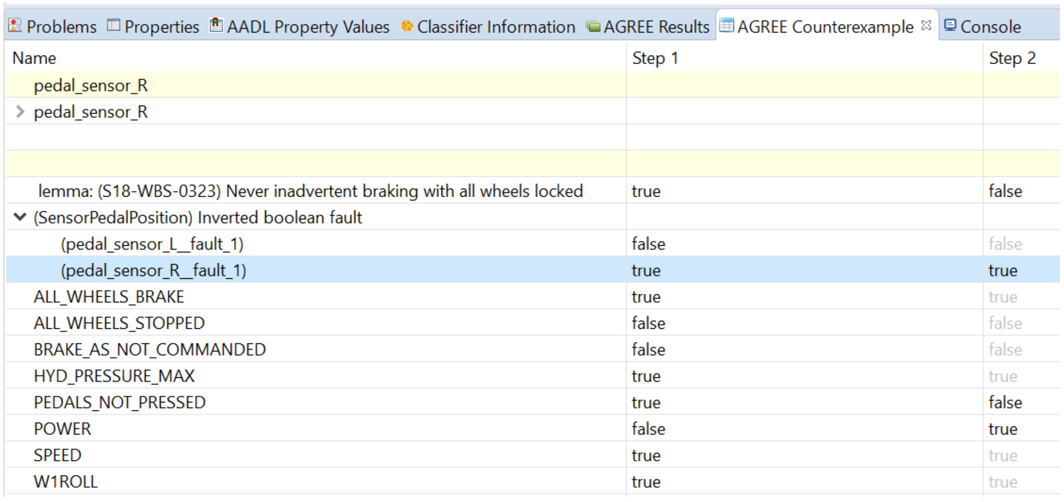


```
Minimal Cut Sets for property violation:
property lustre name: safety___GUARANTEE1
property description:  lemma: (S18-WBS-R-0322-left) Asymmetrical left braking.
Total 18 Minimal Cut Sets
Cardinality 1 number: 18

Minimal Cut Sets for property violation:
property lustre name: safety___GUARANTEE2
property description:  lemma: (S18-WBS-R-0322-right) Asymmetrical right braking
Total 18 Minimal Cut Sets
Cardinality 1 number: 18

Minimal Cut Sets for property violation:
property lustre name: safety___GUARANTEE0
property description:  lemma: (S18-WBS-R-0321) Never loss of all wheel braking
Total 6 Minimal Cut Sets
Cardinality 1 number: 6
```

Fig. 17: Tally Output of MinCutSets

3.8 Use of Analysis Results to Drive Design Change

We use a single top level requirement of the WBS to illustrate how Safety Annex can be used to detect design flaws and how faults can affect the behavior of the system (S18-WBS-0323 : Never inadvertent braking with all wheels locked). This safety property description can be found in detail in Table 1. 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 18 showing the active fault on the pedal sensor.



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

Fig. 18: AGREE counterexample for inadvertent braking safety property

To mitigate this problem, redundancy can be added to handle a single faulty sensor by using three sensors. The overall output from the sensor system may use a voting scheme to determine validity of the sensor reading. There are multiple voting schemes that are possible, one of which is a majority voting. 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 19 outlines these architectural changes that were made in the model.

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

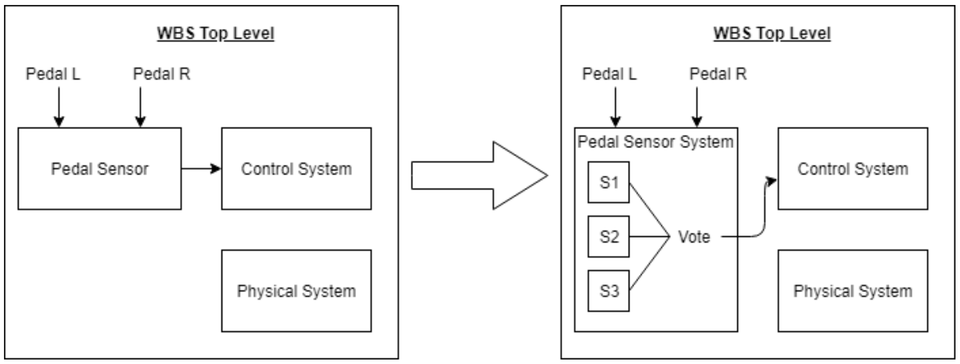


Fig. 19: Changes in the architectural model for fault mitigation

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

4 Discussion

The approach outlined in this research is not meant to supplant the safety analyst, but rather to provide the analyst with additional insight into complex critical systems under development. Other aspects of safety assessment that are parallel to the qualitative and quantitative safety analysis as supported by our approach will continue to be handled through traditional means (e.g., development/design assurance levels or integrity levels of components). Our contributions do not serve to replace the expertise of a safety analyst or encompass all of the assessment process, but instead to provide automated and comprehensive analysis to verify safety requirements in the presence of faults and generate evidence for the assessment process. This is especially useful for increasingly complex software-intensive avionics systems where it becomes ever more challenging for manual analysis to comprehensively enumerate all possible failure causation paths.

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. 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 challenging and time-consuming process to acquire information about the behaviors of the software applications hosted in a system and their impact on the overall system safety.

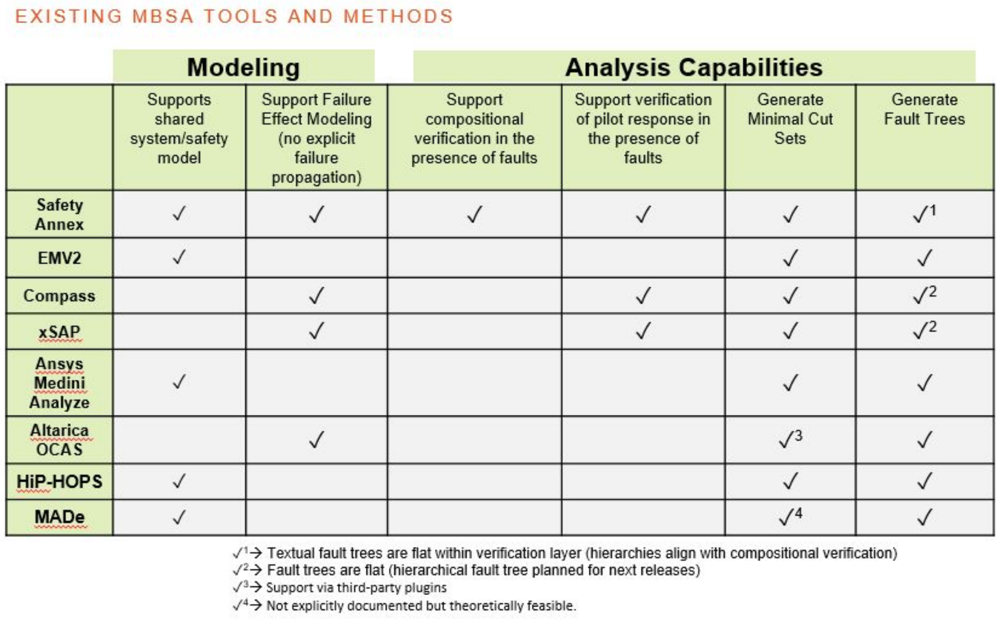
One of our goals is to transition the tools we have developed into use by the safety engineers who perform safety assessment of aircraft digital systems. Therefore, we need to understand how the tools and the models will fit into the existing safety assessment and certification process. In our domain of interest, the current safety assessment process at the system level is based on ARP 4754A [37] and ARP4761 [38].

A model-based approach for safety analysis was proposed by Joshi et. al in [29–31]. 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 (Existing System Model (ESM)), causal, compositional, mixed FLM/FEM approach.

Tools such as the AADL Error Model Annex, Version 2 (EMV2) [20], HiP-HOPS for EAST-ADL [16], and Ansys Medini [1] are *FLM*-based *ESM* approaches. As previously discussed, given many possible faults, these propagation relationships require substantial user effort and become more complex. In addition, it becomes the analyst's responsibility to determine whether faults can propagate; missing propagations lead to unsound analyses. In the safety annex, propagations occur through system behaviors (defined by the nominal contracts) with no additional user effort.



EXISTING MBSA TOOLS AND METHODS

	Modeling		Analysis Capabilities			
	Supports shared system/safety model	Support Failure Effect Modeling (no explicit failure propagation)	Support compositional verification in the presence of faults	Support verification of pilot response in the presence of faults	Generate Minimal Cut Sets	Generate Fault Trees
Safety Annex	✓	✓	✓	✓	✓	✓[1]
EMV2	✓				✓	✓
Compass		✓		✓	✓	✓[2]
xSAP		✓		✓	✓	✓[2]
Ansys Medini Analyze	✓				✓	✓
Altarica OCAS		✓			✓[3]	✓
HiP-HOPS	✓				✓	✓
MADe	✓				✓[4]	✓

✓[1]→ Textual fault trees are flat within verification layer (hierarchies align with compositional verification)
✓[2]→ Fault trees are flat (hierarchical fault tree planned for next releases)
✓[3]→ Support via third-party plugins
✓[4]→ Not explicitly documented but theoretically feasible.

Fig. 20: Related MBSA Tools and Methods

Figure 20 shows a reference table listing a few of the related work tools we describe in the remainder of this section. The figure highlights important features of the support provided. Closely related to our work is the model-based safety assessment toolset called COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [9]. COMPASS is a mixed *FLM/FEM*-based, *causal* tool suite that uses the SLIM language, which is based on a subset of AADL, for its input models [10, 14]. In SLIM, a nominal system model and the error model are developed separately and then transformed into an extended system model and verification is performed over this extended model.

Other related work includes SmartIFlow [28], which 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. The Safety Analysis and Modeling Language (SAML) [25] is a *FEM*-based, *purpose-built*, *monolithic causal* safety analysis language. AltaRica [6, 36] is a *FEM*-based, *purpose-built*, *monolithic* safety analysis language with several dialects. There is one dialect of AltaRica which uses 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 [5]. MADe is a model-based integrated toolset that allows users to identify failures based on functional dependencies captured in the model and generates graphical representations of failure propagations [35].

Formal verification tools based on model checking have been used to automate the generation of safety artifacts [7, 11, 15], but 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 [12].

In contrast to the related work discussed previously, 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 extends and adapts the work of Joshi et al. [31] to the AADL modeling language. The tool and documentation are made available under a BSD license and can be located at: `https://github.com/loonwerks/AMASE/`.

5 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. The nominal model is extended with fault definitions, which allows safety analysis and system implementation to be driven from a single common model. We found that a close integration of behavioral fault analysis into AADL provided a close connection between system and safety analysts. Changes made to the system model were immediately reflected in both the nominal analysis and the safety analysis.

The use of formal methods supports comprehensive exploration on the effect of faulty component behaviors on the system level failure condition without the need to add separate propagation specifications to the model. Once the interface specifications were written using AGREE and the faults were defined on component outputs, the verification process allowed the analyst to see immediately how an error propagation affected the system. The effect of an active fault did not need to be manually defined in order to see the behavior of the system in the presence of faults.

During the development of this approach we worked closely with safety engineers to ensure that the needs of the analysts are supported. This approach was illustrated through the use case of an aircraft system, but can be applied on the development of critical systems in multiple domains (e.g., cyber-physical systems, nuclear power plants, automotive development).

Future work includes compilation of minimal cut sets into graphical fault tree format, expanding the user interface to provide ease in fault model creation, and transforming the counterexample into a sequence flow showing how the system changes as faults are activated. The research presented in this paper, as well as the contributions of future work, all serve to support the safety assessment process. These contributions do not encompass all of the assessment process, but instead aim to provide automated and comprehensive analysis and also to generate evidence for the assessment process.

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

Glossary

AADL Architecture Analysis and Design Language.
AGREE Assume Guarantee REasoning Environment.
AIR Aerospace Information Report.
ARP Aerospace Recommended Practices.

BSCU Braking System Control Unit.

CPU Central Processing Unit.

ESM Existing System Model.

FEM Failure Effect Modeling.
FLM Failure Logic Modeling.

HW Hardware.

MBSA Model-based Safety Analysis/Assessment.
MBSE Model-based Systems Engineering.

OSATE Open Source AADL Tool Environment.

SAE Society of Automotive Engineering.
SASM Safety Analysis System Model.
SW Software.

WBS Wheel Brake System.

References

1. Ansys Medini Tool. `https://www.ansys.com/products/systems/ansys-medini-analyze/medini-analyze-capabilities`, last accessed on 2020-11-17.
2. AIR 6110. Contiguous aircraft/system development process example, Dec. 2011.
3. AS5506C. Architecture Analysis & Design Language (AADL), Jan. 2017.
4. 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.
5. 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.
6. 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.
7. 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.
8. M. Bozzano, A. Cimatti, A. Griggio, and C. Mattarei. Efficient anytime techniques for model-based safety analysis. In *Computer Aided Verification*, 2015.

9. M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The COMPASS approach: Correctness, modeling and performability of aerospace systems. In *Computer Safety, Reliability, and Security*. Springer Berlin Heidelberg, 2009.
10. M. Bozzano, A. Cimatti, J.-P. Katoen, V. Yen Nguyen, T. Noll, and M. Roveri. Model-based codesign of critical embedded systems. 507, 2009.
11. 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.
12. M. Bozzano, A. Cimatti, C. Mattarei, and S. Tonetta. Formal safety assessment via contract-based design. In *Automated Technology for Verification and Analysis*, 2014.
13. 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.
14. 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.
15. M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic fault tree analysis for reactive systems. In *ATVA*, 2007.
16. 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.
17. 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.
18. K. Driscoll, H. Sivencrona, and P. Zumsteg. Byzantine fault tolerance, from theory to reality. In *SAFECOMP*, LNCS, 2003.
19. P. Feiler and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.
20. 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.
21. S. Friedenthal, A. Moore, and R. Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.
22. A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The JKind model checker. *CAV 2018*, 10982, 2018.
23. E. Ghassabani, A. Gacek, and M. W. Whalen. Efficient generation of inductive validity cores for safety properties. *CoRR*, abs/1603.04276, 2016.
24. 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.
25. M. Gudemann and F. Ortmeier. A framework for qualitative and quantitative formal model-based safety analysis. In *HASE 2010*, 2010.
26. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. In *IEEE*, volume 79(9), pages 1305–1320, 1991.
27. P. Helle. Automatic SysML based safety analysis. In *Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems*, pages 19–24, 2012.
28. P. Hönig, R. Lunde, and F. Holzapfel. Model based safety analysis with smartIflow. *Information*, 8(1), 2017.
29. 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.
30. 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.
31. A. Joshi, S. P. Miller, M. Whalen, and M. P. Heimdahl. A proposal for model-based safety analysis. In *Proceedings of 24th Digital Avionics Systems Conference*, 2005.
32. 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.

33. MathWorks. The MathWorks Inc. Simulink Product Web Site, 2004. `http://www.mathworks.com/products/simulink`, last accessed on 2017-09-30.
34. F. Mhenni, N. Nguyen, and J.-Y. Choley. Automatic fault tree generation from SysML system models. In *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pages 715–720. IEEE, 2014.
35. PHM Technology. MADe for model-based FTA. `https://www.phmtechnology.com/assets/downloads/default/MADe%20for%20Model-based%20FTA.pdf`, last accessed on 2021-03-12.
36. 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.
37. SAE ARP 4754A. Guidelines for development of civil aircraft and systems, December 2010.
38. SAE ARP 4761. Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment, December 1996.
39. D. Stewart, J. Liu, M. Heimdahl, M. Whalen, D. Cofer, and M. Peterson. Architectural modeling and analysis for safety engineering (AMASE), NASA final report. `https://github.com/loonwerks/AMASE/tree/master/doc/AMASE_Final_Report_2019`, 2019.
40. 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*, 2020.
41. 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.
42. D. Stewart, M. Whalen, D. Cofer, and M. P. Heimdahl. Architectural modeling and analysis for safety engineering. In *IMBSA 2017*, pages 97–111, 2017.