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*

September 30, 2019

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 [10, 31, 33, 37, 41], 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 [43] models [15, 35–37]. 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) [26], COMPASS [11], and AltaRica [7, 47]. 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 [19], EMV2 [26]) and other related toolsets (COMPASS [11], Cecilia [6], 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 [52] and ARP4761 [51]. 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) [25] 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

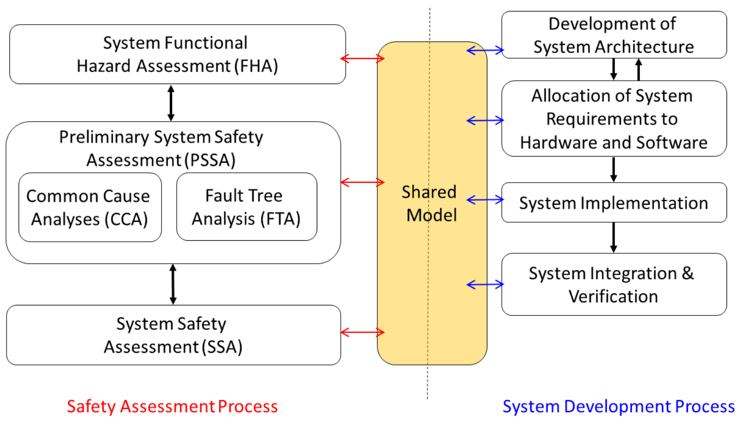

System Functional Hazard Assessment (FHA)
Preliminary System Safety Assessment (PSSA)
Common Cause Analyses (CCA)
Fault Tree Analysis (FTA)
System Safety Assessment (SSA)
Shared Model
Development of System Architecture
Allocation of System Requirements to Hardware and Software
System Implementation
System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

Figure 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) [22] 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 [32] 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 [22].

In our prior work [55], 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 [47], smartIFlow [34] and xSAP [8]. These tools and their accompanying notations are separate from the system development model. Other tools extend existing system models, such as HiP-HOPS [19] and the AADL Error Model Annex, Version 2 (EMV2) [26]. 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 [37] to the AADL modeling language. Stewart, et. al provide more information on the approach [55], 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 [52], 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 [51], 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 [10, 14, 15, 35]. The preliminary work for the safety annex was based on a simple model of the WBS [55]. 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 [15].

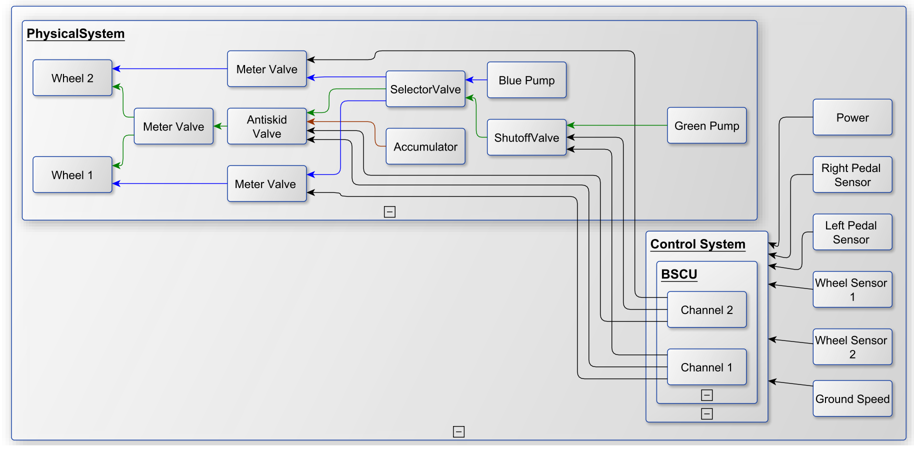


Figure 2: Wheel Brake System

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 anti-skid 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 [52]. 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 [53]. 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 [53].

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.

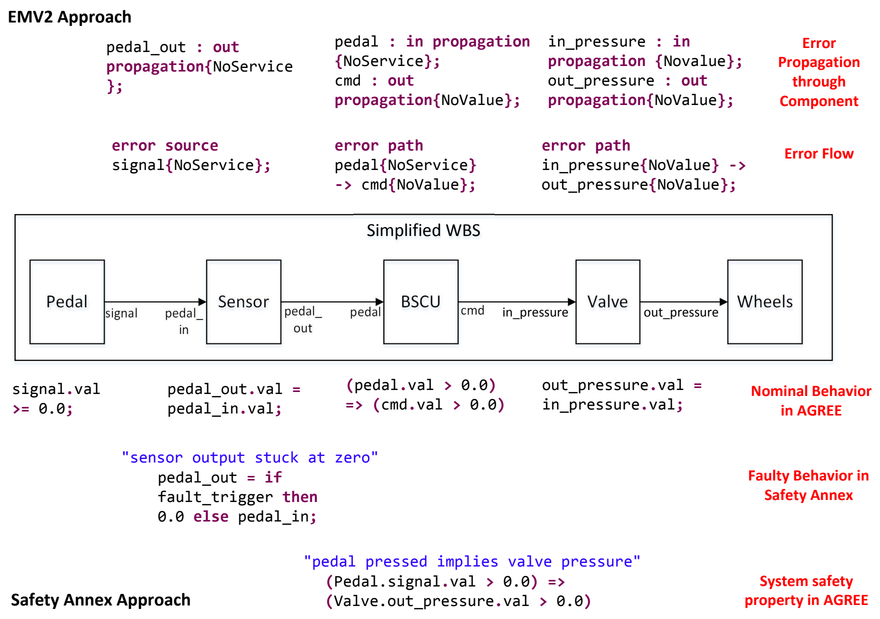
3.2 Implicit Error Propagation

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) [26] 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



```
pedal_out : out
propagation{NoService
};

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

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

error source
signal{NoService};

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

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

signal.val
>= 0.0;

pedal_out.val =
pedal_in.val;

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

out_pressure.val =
in_pressure.val;

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

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

Figure 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 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;
}
```

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

In the WBS example, assume that both the green and blue hydraulic pumps are located in the same compartment in the aircraft and an explosion in this compartment rendered both pumps inoperable. The HW fault definition can be modeled first in the green hydraulic pump component as shown in Figure 3.3. 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 [23]. 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.

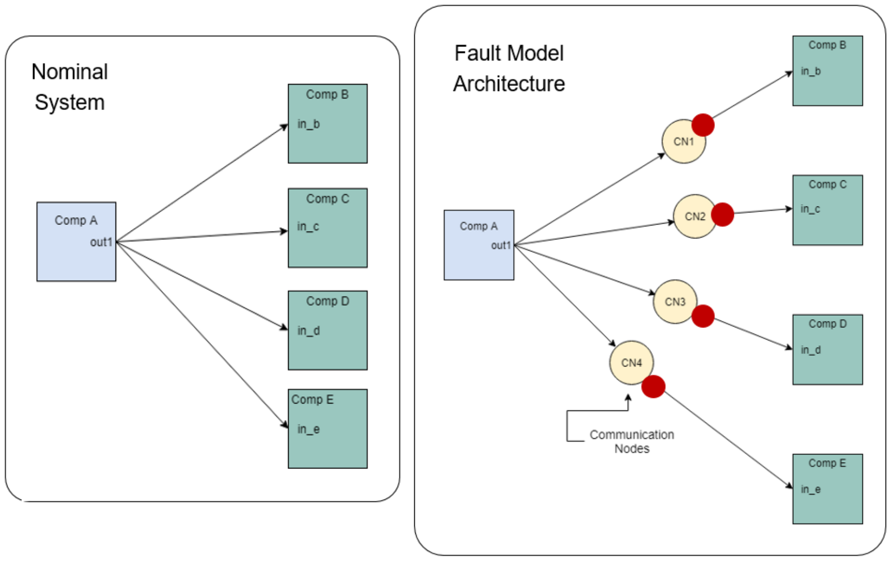


Figure 7: Communication Nodes in Asymmetric Fault Implementation

```
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 [18, 23], 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.

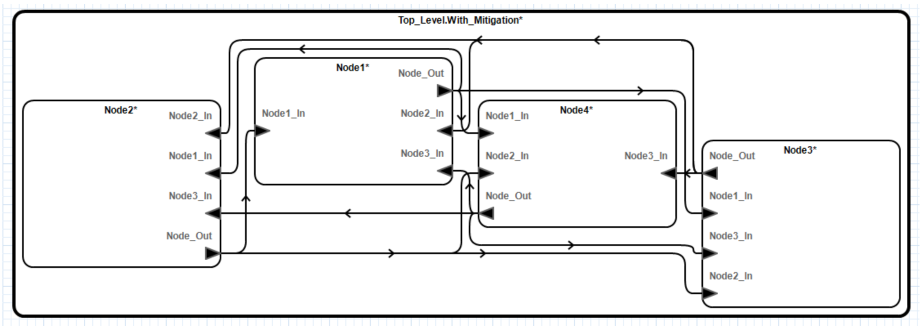
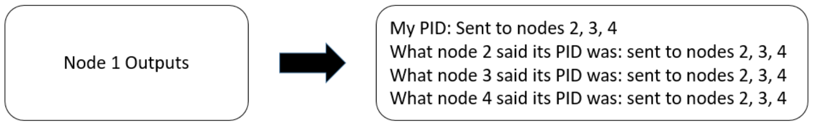


Figure 9: Updated PID Example Architecture

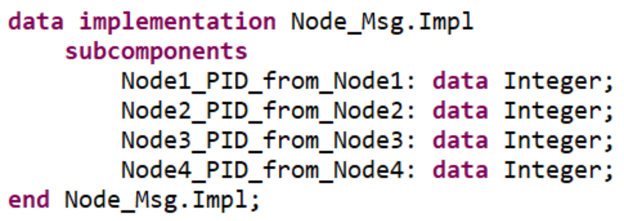
Each node reports its own PID to all other nodes in the first round of communication. In the second round, each node informs the others what they saw in terms of everyone's PIDs. The outputs from a node are described in Figure 10. These outputs



My PID: Sent to nodes 2, 3, 4
What node 2 said its PID was: sent to nodes 2, 3, 4
What node 3 said its PID was: sent to nodes 2, 3, 4
What node 4 said its PID was: sent to nodes 2, 3, 4

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

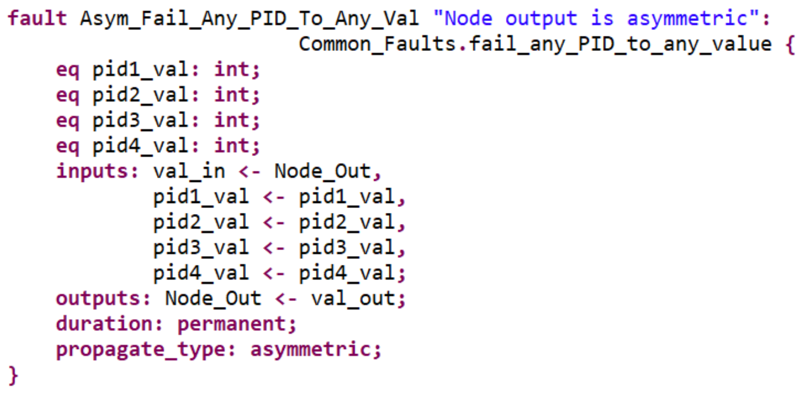
are modeled as a nested data implementation in AADL and each field corresponds to a PID from a node. The AADL code fragment defining this data implementation is shown in Figure 11.



```
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;
```

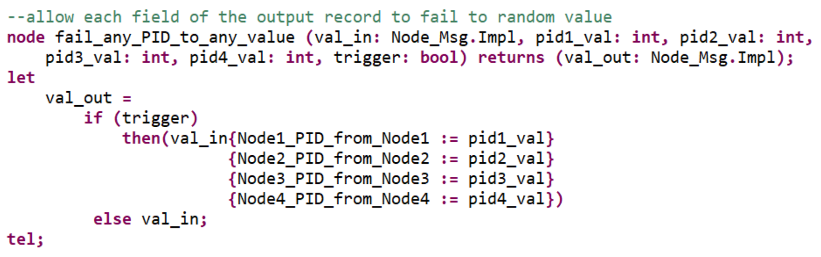
Figure 11: Data Implementation in AADL for Node Outputs

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



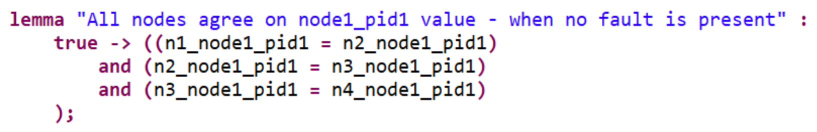
```
--allow each field of the output record to fail to random value
node fail_any_PID_to_any_value (val_in: Node_Msg.Impl, pid1_val: int, pid2_val: int,
    pid3_val: int, pid4_val: int, trigger: bool) returns (val_out: Node_Msg.Impl);
let
    val_out =
        if (trigger)
            then(val_in{Node1_PID_from_Node1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

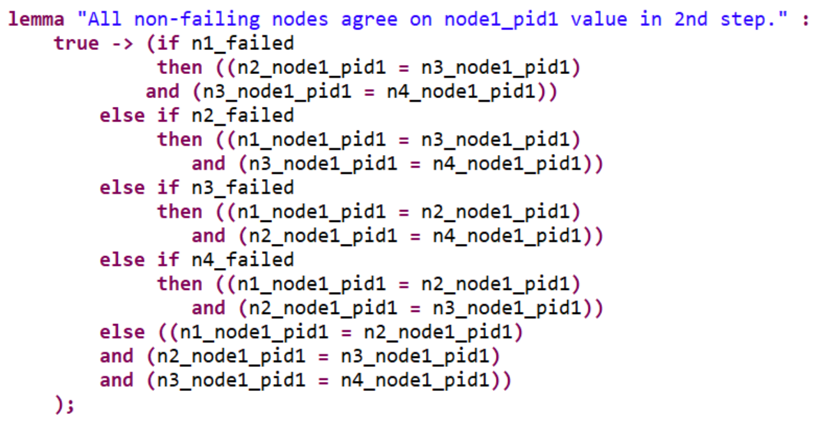
- In the case of no active faults, all nodes must agree on the correct PID of all other nodes.
- In the case of an active fault on a node, all non-failed nodes must agree on a PID for all other nodes.

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)
    );
```

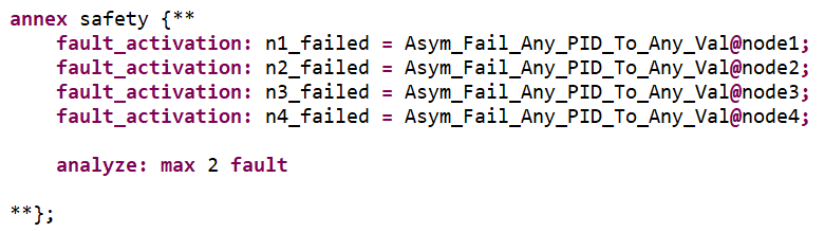
Figure 14: Agreement Protocol Contract in AGREE for No Active Faults



```
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 [53].

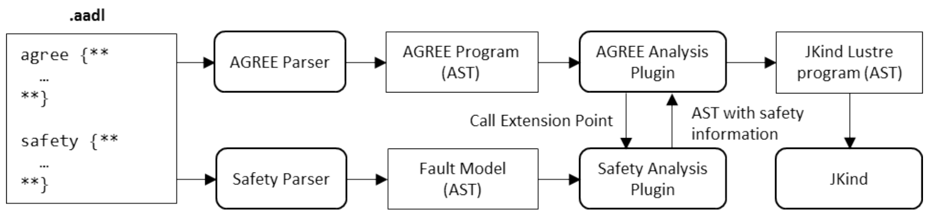


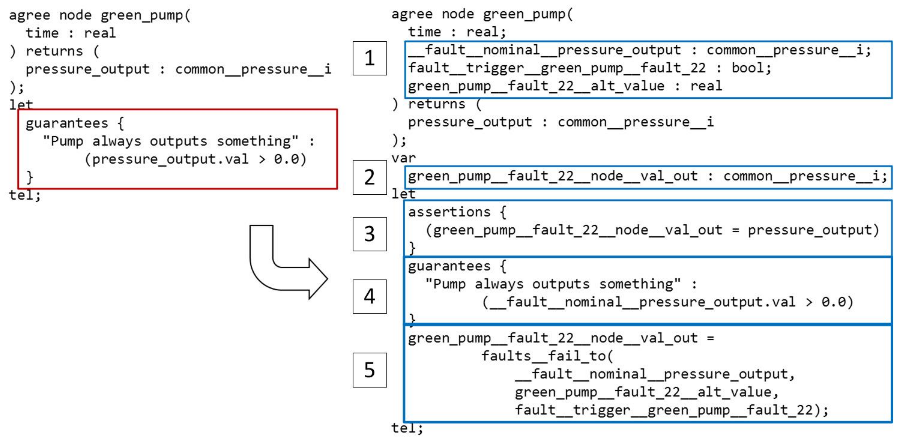
Figure 17: Safety Annex Plug-in Architecture

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 [22]. 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 [32] 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 [32]) follows the standard translation path to the model checker JKind [27], an infinite-state model checker for safety properties.



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

```
  agree node green_pump(
    time : real;
1   __fault__nominal__pressure_output : common__pressure__i;
    fault__trigger__green_pump__fault_22 : bool;
    green_pump__fault_22__alt_value : real
  ) returns (
    pressure_output : common__pressure__i
  );
  var
2   green_pump__fault_22__node__val_out : common__pressure__i;
  let
    assertions {
3     (green_pump__fault_22__node__val_out = pressure_output)
    }
    guarantees {
      "Pump always outputs something" :
4             (__fault__nominal__pressure_output.val > 0.0)
    }
    green_pump__fault_22__node__val_out =
            faults__fail_to(
5               __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 [29]. 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 [30].

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 7.2, 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 (see Section 7 for more details on the transformation algorithms).

6 Analysis of the Model

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

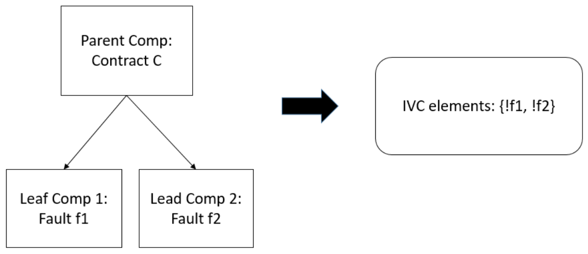


Figure 19: IVC Elements used for Consideration in a Leaf Layer of a System

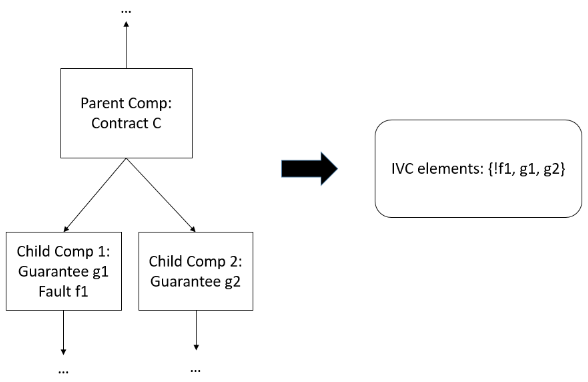


Figure 20: IVC Elements used for Consideration in a Middle Layer of a System

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, 21]

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  $\mathcal{F} = \{\}$ : 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($\{q, r_i\}, \mathcal{F}$);
```

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.

Generate MinCutSet analysis 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 (C) and how many cut sets of that cardinality. When the analysis is run, the user specifies the value N. This gives cut sets of cardinality *less than or equal to* N. (For the full text of the properties, see Table 1.)

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

Table 2: WBS MinCutSet Analysis Results for Cardinality c

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

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 probablistic hypothesis, *Verify in the Presence of Faults* is performed using only monolithic analysis, but generating minimal cut sets is performed using compositional analysis.

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

As shown in Table 3, the number of allowable combinations drops considerably when given probabilistic threshold as compared to just fault combinations of certain cardinalities. For example, one contract (inadvertent wheel braking of all wheels) had over a million minimal cut sets produced when looking at it in terms of max N analysis, but after taking probabilities into account, it is seen that only one combination of faults can violate this property. (For the full text of the properties, see Table 1.)

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

Table 3: WBS MinCutSet Results for Probabilistic Analysis

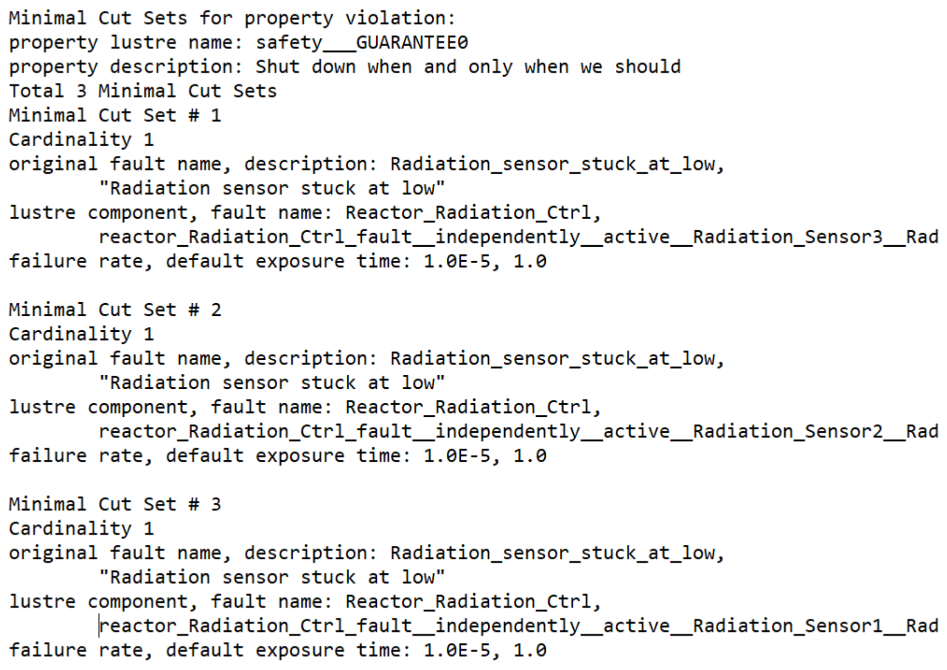
6.2.5 Results from Generate Minimal Cut Sets

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 [42] to produce hierarchical fault trees that are consistent with the system architecture/component verification layers, or flat fault trees consist of minimal cut sets only, both in graphical form. A sample graphical fault tree output from the SOTERIA tool is shown in Figure 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 anectodally easier to read than the graphical format for larger systems.

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 indadvertent braking with all wheels locked to illustrate how Safety Annex can be used to detect design flaws and how faults can affect the behavior of the system). This safety property

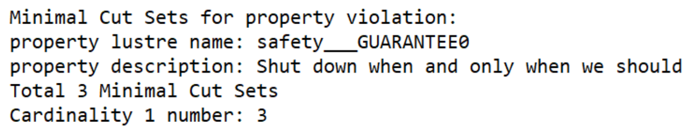


```
Minimal Cut Sets for property violation:
property lustre name: safety___GUARANTEE0
property description: Shut down when and only when we should
Total 3 Minimal Cut Sets
Minimal Cut Set # 1
Cardinality 1
original fault name, description: Radiation_sensor_stuck_at_low,
        "Radiation sensor stuck 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

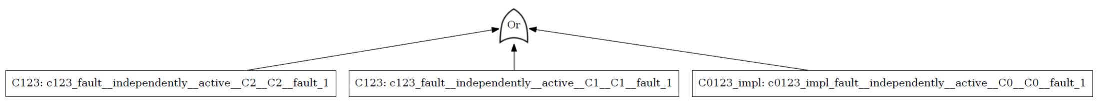
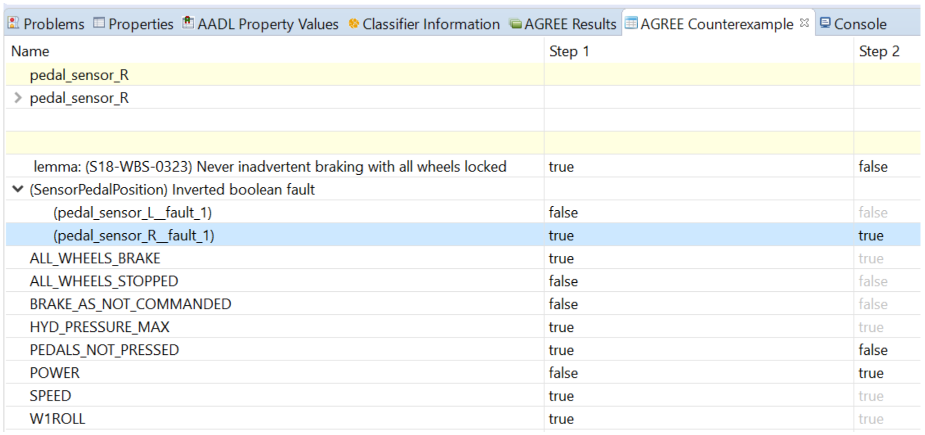

Or
C123: c123_fault__independently__active__C2__C2__fault_1
C123: c123_fault__independently__active__C1__C1__fault_1
C0123_impl: c0123_impl_fault__independently__active__C0__C0__fault_1

Figure 23: Example SOTERIA Fault Tree

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.



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.

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

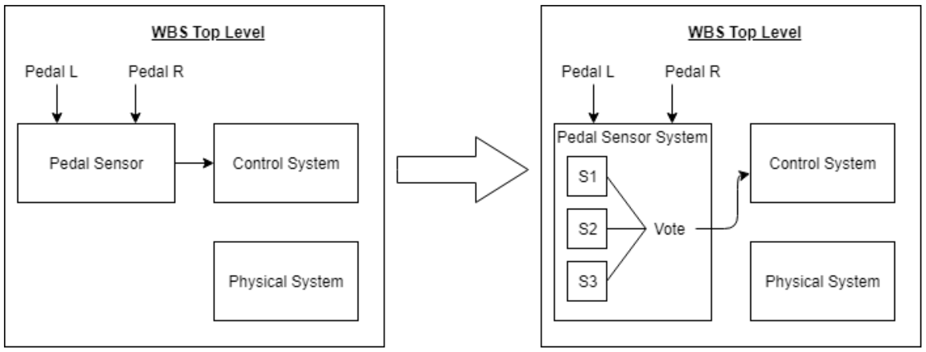


Figure 25: Changes in the architectural model for fault mitigation

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 [53]. 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 [21]; 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 Fault Tree Analysis

The use of fault trees are common in many safety assessment processes and the ability to generate the cut sets needed for the construction of the fault tree is a useful part of any safety analysis tool. The fault tree is a safety artifact commonly referenced in requirement protocol documents such as ARP4761, ARP4754, and AIR6110 [1,51,52].

A Fault Tree (FT) is a directed acyclic graph whose leaves model component failures and whose gates model failure propagation [50]. The system failure under examination is the root of the tree and is called the Top Level Event (TLE). The node types in a fault tree are *events* and *gates*. An event is an occurrence within the system, typically the failure of a subsystem down to an individual component. Events can be grouped into Basic Events (BEs), which occur independently, and *intermediate events* which occur dependently and are caused by one or more other events [24]. These events model the failure of the system (or subsystem) under consideration. The gates represent how failures propagate through the system and how failures in subsystems can cause system wide failures. The two main logic symbols used are the Boolean logic AND-gates and OR-gates. An AND-gate is used when the undesired top level event can only occur when all the lower conditions are true. The OR-gate is used when the undesired event can occur if any one or more of the next lower conditions is true. This is not a comprehensive list of gate types, but we focus our attention on these two common gate types.

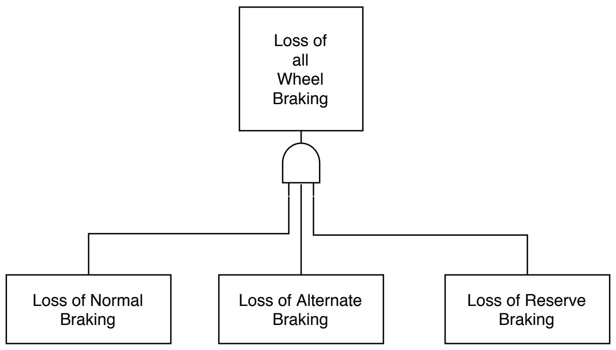


Figure 26: A simple fault tree

Figure 26 shows a simple example of a fault tree based on SAE ARP4761 [51]. In this example, the top level event corresponds to an aircraft losing all wheel braking. In order for this event to occur, all of the basic events must occur. This is seen through the use of the AND gate below the top level event. The gates in the fault tree describe how failures propagate through the system. Each gate has one output and one or more inputs. In Figure 26, the AND gate has three inputs and one output. The leaves of the tree represent the basic events of the system. In the case of this fault tree, these three events are also the Minimal Cut Sets (MinCutSets) for this top level event. A MinCutSet is the minimal set of basic events that must occur together in order to cause the TLE to occur. Generating and analyzing these MinCutSets is important to FTA and has been an active area of interest in the research community since fault trees were first described in Bell Labs in 1961 [24, 50].

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

7.2 Definitions

Intuitively a constraint system contains the contracts that constrain component behavior and faults that are defined over these components. In the case of a nominal model augmented with faults, a constraint system is defined as follows. Let F be the set of all fault activation literals defined in the model and G be the set of all component contracts (guarantees).

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

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

Given a state space S, a transition system (I, T) consists of the initial state predicate $I : S \rightarrow \{0, 1\}$ and a transition step predicate $T : S \times S \rightarrow \{0, 1\}$. Reachability for (I, T) is defined as the smallest predicate $R : S \rightarrow \{0, 1\}$ that satisfies the following formulas:

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

A safety property $\mathcal{P} : S \rightarrow \{0, 1\}$ is a state predicate. A safety property $\mathcal{P}$ holds on a transition system (I, T) if it holds on all reachable states. More formally, $\forall s.R(s) \Rightarrow \mathcal{P}(s)$. When this is the case, we write $(I, T) \vdash \mathcal{P}$ [30].

Given a transition system that satisfies a safety property P, it is possible to find which model elements are necessary for satisfying the safety property through the use of the *All Minimal Inductive Validity Cores All-MIVCs* algorithms [4, 30]. This algorithm collects all minimal unsatisfiable subsets of a given transition system in terms of the negation of the top level property. The minimal unsatisfiable subsets consist of component contracts constrained to *true*. When the constraints on these model elements are removed from the constraint system C, this results in an UNSAT system. This can be seen as the minimal explanation of the constraint systems infeasibility. Recall that this constraint system is in terms of the *negation* of the safety property. Thus, this algorithm provides all model elements required for the proof of the safety property.

We utilize this algorithm by providing not only component contracts (constrained to *true*) as model elements, but also fault activation literals constrained to *false*. Thus the resulting MIVCs will contain the required contracts and constrained fault activation literals in order to prove the safety property. This information is used throughout this section to provide the underlying theory behind the generation of minimal cut sets from all MIVCs.

Definitions 1-3 are taken from research by Liffiton et. al. [40].

Definition 2. *: A Minimal Unsatisfiable Subset (MUS) of a constraint system C is : $\{MUS \subseteq C \mid MUS$ is UNSAT and $\forall c \in MUS$: $MUS \setminus \{c\}$ is SAT$\}$. This is the minimal explaination of the constraint systems infeasability.*

A closely related set is a *minimal correction set* (MCS). The MCSs describe the minimal set of model elements for which if constraints are removed, the constraint system is satisfied. For constraint system C, this corresponds to which faults are not constrained to inactive (and are hence active) and violated contracts which lead to the violation of the safety property. In other words, the minimal set of active faults and/or violated properties that lead to the top level event.

Definition 3. *: A Minimal Correction Set (MCS) of a constraint system C is : $\{MCS \subseteq C \mid C \setminus MCS$ is SAT and $\forall S \subset MCS : C \setminus S$ is UNSAT$\}$. A MCS can be seen to "correct" the infeasability of the constraint system.*

A duality exists between MUSs of a constraint system and MCSs as established by Reiter [49]. This duality is defined in terms of *minimal hitting sets*. A hitting set of a collection of sets A is a set H such that every set in A is "hit" be H; H contains at least one element from every set in A [40]. The MCSs can be generated from the MUSs if all MUSs are known. Thus, the use of the All-MIVCs algorithm is required.

Definition 4. *: Given a collection of sets K, a hitting set for K is a set $H \subseteq \cup_{S \in K} S$ such that $H \cap S \neq \emptyset$ for each $S \in K$. A hitting set for K is minimal if and only if no proper subset of it is a hitting set for K.*

Utilizing this approach, the MCSs are generated from the MUSs that are provided by the All-MIVCs algorithm [30] and a minimal hitting set algorithm developed by Murakami et. al. [28, 45].

A Minimal Cut Set ($MinCutSet$) is a minimal collection of faults that lead to the violation of the safety property (or in other words, lead to the top level event in the fault tree). We define a minimal cut set consistently with much of the research in this field [24, 50]

Definition 5. *: A Minimal Cut Set can be defined as the set of faults in a system that cause the violation of the safety property. Furthermore, any strict subset of these faults will not cause violation of the safety property.*

7.3 Transformation of All-MIVCs into Minimal Cut Sets

All Minimal Inductive Validity Cores are collected by use of the All-MIVCs algorithm [30]. These are all of the Minimal Unsatisfiable Subsets (MUSs) which can be used as input to the Minimal Hitting Set algorithm [28, 45] in order to collect all Minimal Correction Sets (MCSs). The MCSs are then transformed into Minimal Cut Sets according to the following theoretical results.

The definition of the constraint system follows Definition 1 in Section 7.2.

Theorem 1. *The MinCutSet can be generated by the transformation of the Minimal Correction Set (MCS).*

Proof. All MCSs are of the form $MCS = G \cup \overline{F}$ where G consists of contracts in the system and $\overline{F}$ consists of faults constrained to false for constraint system C.

Case 1: $G = \emptyset$
In the leaf level of the system, only constrained faults are contained in MCS. According to the defintion of an MCS, upon removing the constraints from C of the elements contained in the MCS, the constraint system is satisfiable. Furthermore, the MCS is the minimal such set. Thus, the constraint system with the *active* faults from MCS will cause the *negation* of the safety property to be satisfied. The set of unconstrained faults found in the MCS is the defintion of a minimal cut set.

Case 2: $G \neq \emptyset$
Let $MCS = \{\neg f_1, ..., \neg f_n, g_1, ..., g_m\}$ where $f_j \in F$ and $g_k \in G$ and $\neg f$ is a fault activation literal f constrained to *false*. For all $g_i \in MCS$, we know that the validity of g_k is required in the proof of the top level property P due to its generation through the All-MIVCs algorithm. For $g_1 \in MCS$, there exists a set containing all minimal cut sets of g_1. Call this $Cut(g_1) = \{F_{1i} \subseteq F | F_{1i} \text{ is a min cut set for } g_1, i = 1, ..., p_1\}$.

Replace g_1 with F_{1i} for all $i = 1, ..., p_1$. This produces p_1 new MCSs of the form:

$$\begin{gathered} \{f_1, ..., f_n, F_{11}, ..., g_m\} \\ \{f_1, ..., f_n, F_{12}, ..., g_m\} \\ \vdots \\ \{f_1, ..., f_n, F_{1p_1}, ..., g_m\} \end{gathered}$$

Perform this replacement for all $g_k \in MCS$, $k = 1, ..., m$.

Let I_α be one of the sets generated by full replacement of all contracts in some MCS_α with their respective minimal cut set and all fault activation literals constrainted to *true*.

Claim: I_α is a minimal cut set for safety property P
By the definition of MCS_α, $C \setminus MCS_\alpha$ is SAT. Thus the fault activation literals in MCS_α are *true* and the contracts in MCS_α are violated. This combination will cause violation of the safety property (i.e., the constraint system $C \setminus MCS_\alpha$ is SAT).

The generation of I_α consists of replacing all contracts in MCS_α by their respective minimal cut sets and unconstraining all fault activation literals in MCS_α. These unconstrained faults cause the violation of all contracts in MCS_α. Hence together, the violation of P since $C \setminus MCS_\alpha$ is SAT. Therefore I_α is a cut set for P.

Minimality follows from the defintion of MCS:

Assume that $I_\beta \subset I_\alpha$. Then there is at least one $f \in I_\alpha$ where $f \notin I_\beta$. Let $\neg f$ be the fault activation literal f constrained to *false*.

If $\neg f \in MCS_\alpha$ and $f \notin I_\beta$, then by removing all constraints of the fault literals in I_β from C ($C \setminus I_\beta$), we see that the resulting constraint system is UNSAT by the minimality of MCS_α. Therefore I_β is not a cut set for P.

If $f \in MinCut(g)$ where $MinCut(g)$ is a minimal cut set for some $g \in MCS_\alpha$, then $C \setminus I_\beta$ will not remove constraints for all of the elements in $MinCut(g)$ and therefore g remains unviolated, i.e., the constraint that g is *true* is not removed. Thus

$C \setminus I_\beta$ will not remove constraints for all of the elements in MCS_α which means $C \setminus I_\beta$ is UNSAT. Therefore I_β is not a cut set for P.

$\square$

The generation of MIVCs traverses the program in a top down fashion. The transformation of MIVCs to MinCutSets traverses this tree in a bottom up fashion if and only if All-MIVCs have been generated. It is a requirement of the minimal hitting set algorithm that *all* MUSs are used to find the MCSs [28,40,45]. Thus, once All-MIVCs have been found and the minimal hitting set algorithm has completed, the MinCutSets Generation algorithm can begin.

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

Notes regarding the following algorithm: at the onset, the current property P is a top level property. Each of the properties has a list of associated $MCSs$. When the algorithm states that constraints on these elements are removed, more specifically the fault activation literals in MCS are constrained to *true* and the component contracts are constrained to *false*. $List(I)$ is the collection of all $MCSs$ with all constraints removed. Assuming All-MIVCs have been found and the minimal hitting set algorithm has terminated, giving us a list of $MCSs$ for each property in the system.

Algorithm 2: MinCutSets Generation Algorithm

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

The number of replacements R that are made in this algorithm are constrained by the number of minimal cut sets there are for all α contracts within the set *I*. We call the set of all minimal cut sets for a contract g: $Cut(g)$. The following formula defines an upper bound on the number of replacements. The validity of this statement follows directly from the general multiplicative combinatorial principle. Therefore, the number of replacements R is bounded by the following formula where g_1 is the first contract being processed in the set *I*.

$$R \leq |Cut(g_1)| + \sum_{i=1}^{\alpha}(\prod_{j=1}^{i} |Cut(g_j)|)$$

It is also important to note that the cardinality of $List(I)$ is bounded, i.e. the algorithm terminates. Every new I that is generated through some replacement of a contract with its minimal cut set is added to $List(I)$ in order to continue the replacement process for all contracts in I. Thus, the same bound for the number of replacements exists for the size of $List(I)$. Since no infinite sets (either MIVCs or MCSs) are generated by the All-MIVC or minimal hitting set algorithms, all sets are finite.

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

The results from this algorithm are the minimal cut sets for the top level properties. These are presented to the user as described in Section 6.

The MinCutSets are filtered during this process based on the fault hypothesis given. Algorithm 2 is the general approach, but this changes slightly depending on which form of analysis is being performed. These differences are outlined in the following subsections.

7.3.1 Transformation Algorithm for Max N Analysis

Given a top level fault hypothesis statement of maximum N faults, the transformation algorithm filters out any MinCutSets of cardinality greater than N. To assist in performance and scalability, if a minimum cut set exceeds the cardinality of N, it will no longer be considered for the algorithm. We call this *pruning*. This pruning is done in Algorithm 2 in Section 7.3. If the number of faults in an intermediate set I exceeds N, any further replacement of remaining contracts in that intermediate set can never decrease the total number of faults in I. Only the remaining sets are displayed.

7.3.2 Probabilistic Analysis Algorithm

In order to complete the probabilistic analysis, we must first calculate allowable combinations of faults. This allows us the option to eliminate unnecessary combinations while performing the algorithm, thus increasing performance and diminishing the problem of combinatorial explosions in the size of minimal cut sets for larger models.

After the allowable combinations have been found, the algorithm described in Section 7.3 is performed with pruning in order to eliminate probabilistically impossible fault combinations.

Collect Allowable Fault Combinations The collection of allowable fault combinations is a prerequsite needed for both *Verify in the Presence of Faults* and *Generate Minimal Cut Set* analysis and the algorithm used is described in Section 6.2.2. In order to calculate allowable combinations, we must take into account probabilities of the combined faults and compare with the given threshold. Each of the allowable fault combinations has a combined probability that we must later access. These combinations are saved in a list called $\mathcal{A}$:

$\mathcal{A} = \{A_i = FaultCombinations \mid P(FaultCombinations) > threshold\}$

The allowable combinations in $\mathcal{A}$ consist of independent faults.

Prune MinCutSets During the transformation process, if the minimal cut set for a contract is not a subset of any allowed fault combinations, this intermediate set containing this contract is pruned. (Recall that an intermediate set I is obtained by removing all constraints from an MCS and contract replacement has begun.) This pruning occurs in the algorithm described in Section 7.3 and the resulting allowable combinations are displayed to the user.

The implementation of the Safety Annex utilizes the pruning as described above. Theoretically, there are two other options for pruning in this algorithm. They are outlined here for completion. Throughout this discussion, we use the following notation to describe these sets. They are listed here for convenience.

$\mathcal{A}$: the set of all allowable fault combinations

$\mathcal{A}_i$: an individual allowable combination and an element of $\mathcal{A}$

$List(I)$: the set of all intermediate sets

I_i : an element of $List(I)$.
$Cut(g_j)$: all minimal cut sets for contract g_j.
$cut_k(g_j)$: a specific cut set for contract g_j. This is an element of $Cut(g_j)$.

Option 1: Once the replacement is complete and the algorithm terminates, we have all $MinCutSets$ for a top level property. If $MinCutSet \not\subseteq \mathcal{A}_i$ for all $\mathcal{A}_i \in \mathcal{A}$, then we can safely eliminate $MinCutSet$. This is the latest form of pruning and the drawback is when the number of generated cut sets grows, no elimination occurs early. This can be difficult in terms of scalability.

Option 2: Pruning can also be done by checking the union of a minimal cut set for a contract in I with all the faults in that I. If the union of these faults is not an allowable combination, the minimal cut set for this contract in I will be skipped for the replacement. More formally, if $(\{f_n | f_n \in I_i\} \cup cut_k(g_j)) \not\subseteq \mathcal{A}_i$ for all $\mathcal{A}_i \in \mathcal{A}$, then we can safely eliminate this I_i.

8 Related Work

A model-based approach for safety analysis was proposed by Joshi et. al in [35–37]. 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) [26] and HiP-HOPS for EAST-ADL [19] 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) [11]. 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 [12, 16]. In SLIM, a nominal system model and the error model are developed separately and then transformed into an extended system model. This extended model is automatically translated into input models for the NuSMV model checker [20, 46], MRMC (Markov Reward Model Checker) [38, 44], and RAT (Requirements Analysis Tool) [48]. The safety analysis tool xSAP [8] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [9]. 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 [34] 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 [34]: "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) [31] is a *FEM*-based, *purpose-built*, *monolithic causal* safety analysis language. System models constructed in SAML can be used used for both qualitative and quantitative analyses. It allows for the combination of discrete probability distributions and non-determinism. The SAML model can be automatically imported into several analysis tools like NuSMV [20], PRISM (Probabilistic Symbolic Model Checker) [39], or the MRMC probabilistic model checker [38].

AltaRica [7,47] 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 [6]. For this dialect the Safety assessment, fault tree generation, and functional verification can be performed with the aid of NuSMV model checking [13]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [5]. 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 [8, 13, 17]. This approach has limitations in terms of scalability and readability of the fault trees generated. Work has been done towards mitigating these limitations by the scalable generation of readable fault trees [14].

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 [54]. To access the tool plugin, users manual, or models, see the repository [53].

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] 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, 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

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

[15] 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.

[16] M. Bozzano, A. Cimatti, M. Roveri, J. P. Katoen, V. Y. Nguyen, and T. Noll. Codesign of dependable systems: A component-based modeling language. In *2009 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, 2009.

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

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

[19] 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.

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

[21] 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.

[22] 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.

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

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

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

[26] 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.

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

[28] 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.

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

[30] 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.

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

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

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

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

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

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

[45] 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.

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

[47] 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.

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

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

[50] 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.

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

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

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

[54] 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.

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