

Safety Annex for Architecture Analysis and Design Language

Danielle Stewart¹, Jing (Janet) Liu², Michael W. Whalen¹, and Darren Cofer²

¹ University of Minnesota

Department of Computer Science and Engineering
dkstewar, whalen@cs.umn.edu

² Rockwell Collins

Advanced Technology Center
Jing.Liu, darren.cofer@rockwellcollins.com

Abstract. This paper describes a new methodology with tool support for model based safety analysis. It is implemented as a *Safety Annex* for the Architecture Analysis and Design Language (AADL). The Safety Annex provides the ability to describe faults and faulty component behaviors in AADL models. In contrast to previous AADL-based approaches, the Safety Annex leverages a formal description of the nominal system behavior to propagate faults in the system. This approach ensures consistency with the rest of the system development process and simplifies the work of safety engineers. The language for describing faults is extensible and allows safety engineers to weave various types of faults into the nominal system model. The Safety Annex supports the injection of faults into component level outputs, and the resulting behavior of the system can be analyzed using model checking through the Assume-Guarantee Reasoning Environment (AGREE).

Keywords: Model-based systems engineering, fault analysis, safety engineering

1 Introduction

System safety analysis techniques are well-established and are a required activity in the development of safety-critical systems. While model based development methods are widely used in the aerospace industry, these methods are only recently being applied to system safety analysis. Model-based systems engineering (MBSE) methods and tools based on formal methods now permit system-level requirements to be specified and analyzed early in the development process [?, ?]. These tools can also be used to perform safety analysis based on the system architecture and initial functional decomposition. Design models can be integrated into the safety analysis process to help guarantee accurate and consistent results. This integration is especially important as the amount of safety-critical hardware and software in various domains has drastically increased due to the demand for greater autonomy, capability, and connectedness.

We have developed a Safety Annex for the Architecture Analysis and Design Language (AADL) [?] that provides the ability to reason about faults and faulty component behaviors in AADL models. In the Safety Annex approach, we use formal

assume-guarantee contracts to define the nominal behavior of system components. The nominal model is then verified using the Assume Guarantee Reasoning Environment (AGREE) [?]. The Safety Annex provides a way to weave faults into the nominal system model and analyze the behavior of the system in the presence of faults. The Safety Annex also provides a library of common fault node definitions that is customizable to the needs of system and safety engineers. Our approach adapts the work of Joshi et. al in [?] to the AADL modeling language. More information on the approach is available in [?], and the tool and relevant documentation can be found at: <https://github.com/loonwerks/AMASE/>.

There are other tools purpose-built for safety analysis, including AltaRica [?], smartIFlow [?] and xSAP [?]. These notations are separate from the system development model. Other tools extend existing system models, such as HiP-HOPS [?] and the AADL Error Model Annex, Version 2 (EMV2) [?]. 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. [It can serve as the shared model to capture system design and safety-relevant information, and produce both qualitative and quantitative description of the causal relationship between faults and system safety requirements.](#)

[What are our contributions?](#)

2 Preliminaries

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

2.1 Safety Assessment Process

ARP4754A, the Guidelines for Development of Civil Aircraft and Systems [?], has been recognized by the Federal Aviation Administration (FAA) as an “acceptable method for establishing a development assurance process” [?]. It provides guidance on applying development assurance at each hierarchical level throughout the development life cycle of highly-integrated/complex aircraft systems.

The safety assessment process is a starting point at each hierarchical level of the development life cycle, and is tightly coupled with the system development and verification processes. It is used to show compliance with certification requirements, and meeting a company’s internal safety standards [?]. ARP4761, the Guidelines and Methods for Conducting Safety Assessment Process on Civil Airborne Systems and Equipment [?], identifies a systematic means to show compliance. The guidelines presented

in ARP4761 include industry accepted safety assessment processes (Functional Hazard Assessment (FHA), Preliminary System Safety Assessment (PSSA), and System Safety Assessment (SSA)), and safety analysis methods to conduct the safety assessment, such as Fault Tree Analysis (FTA), Failure Modes and Effect Analysis (FMEA), and Common Cause Analysis (CCA).

A prerequisite of performing the safety assessment of a system design is to understand how the system works, primarily focusing on the integrity of the outputs and the availability of the product. The safety engineers then use the acquired understanding to conduct safety analysis, construct the safety analysis artifacts, and compare the analysis results with established safety objectives and safety related requirements.

In practice, prior to performing the safety assessment of a system, the safety engineers are often equipped with fair amount of knowledge on how system works in general, but not necessarily with the specific details for each function associated with the system. I don't know what the previous sentence means...do you mean 'in principle?' rather than 'in general'. It was meant to say that they know at a high level how system like this should work and the safety pitfalls, but not the detailed information about the specific system. Acquiring the knowledge on the content and behavior of a specific function has shown to be time consuming to get it right. In one real case example, it took a safety engineer two full working days to understand how the software works in a Stall Warning System (a small function in comparison to a Fly-By-Wire function within a flight control computer). a dual channel system involving six main system-level functions such as input signal processing 'solid' might be overkill. this needs a little bit of wordsmithing changed 'two solid days' to 'two full working days'. The primary task includes connecting the signal and function flows to relate the input and output signals from end-to-end and understanding the causal effect between them. This is the same amount of time, if not more, as performing the analysis and constructing the analysis artifacts. In another real case example, it took a safety engineer several months to finalize the PSSA document for a Horizontal Stabilizer Control System, involving two major revisions and multiple rounds of reviews with system, hardware, and software engineers.

Capturing failure mode in models and generating safety analysis artifacts directly from models can enhance communication and synchronization between system design process and safety assessment process, and the ability to analyze complex systems. Industry practitioners have come to realize the benefits and importance of using models to assist the safety assessment process (either by augmenting the existing system design model, or by building a separate safety model), and a revision of the ARP4761 with model based safety analysis appendix is under way.

Using the same system design model to conduct both system design and safety analysis can help reduce the gap in comprehending the system behavior and transferring the knowledge between the design model to the safety analysis model. It maintains a living model that captures the latest state of the system design as the process flows per the system development lifecycle. It also allows all participants of the ARP4754A process to be able to communicate and review the system design using the "single source of truth".

In order to allow performing system design and safety analysis on the same model, the system design model will need to be augmented to include both the system design information (e.g., system architecture, functional behavior) and safety-relevant information (e.g., failure mode, failure rate), at the same time keeping the two types of information distinguishable yet interactable from each other.

Figure 1 **This reference should be sooner or the figure should be later in the paper - also the figure needs larger font size and a little less space between the boxes updated the figure and location** presents our proposed use of the shared system design and safety analysis model (to be referred as “the shared model” in the rest of this section) inside the ARP4754A Safety Assessment Process Model (Figure 7 of [?]). As seen in the figure, the shared model represents a system development artifact from the “Development of System Architecture” and “Allocation of System Requirements to Item” activities in the System Development Process, which interacts with the PSSAs and SSAs activities in the Safety Assessment Process. The shared model can serve as a wrapper and interface to capture the information relevant to safety analysis from the system design and implementation.

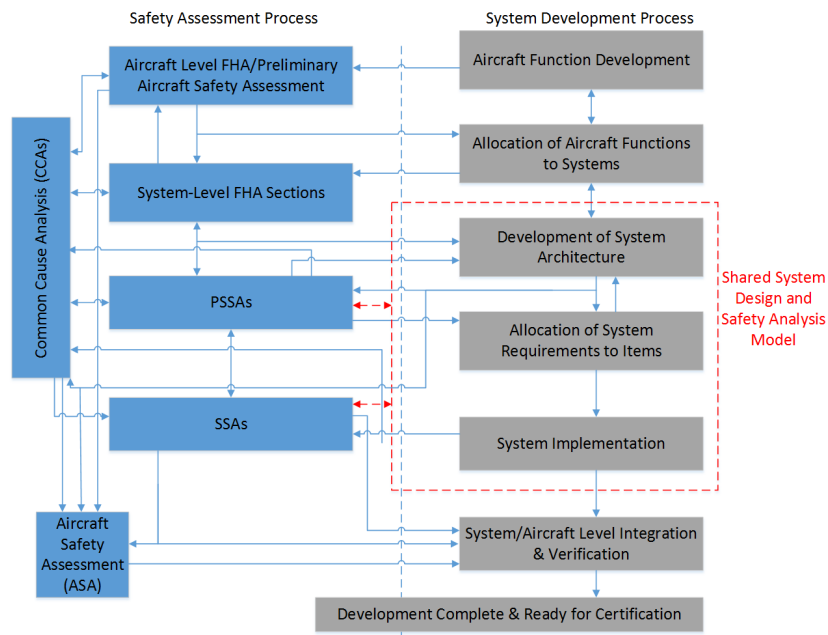


Fig. 1. Using the Shared System/Safety Model in the ARP4754A Safety Assessment Process

Figure 2 shows an example how the preliminary FTAs and FTAs (artifacts from the PSSA and SSA activities in the Safety Assessment Process) can guide and be updated from the shared model. The next subsection will describe the foundation of the modeling technique in more details.

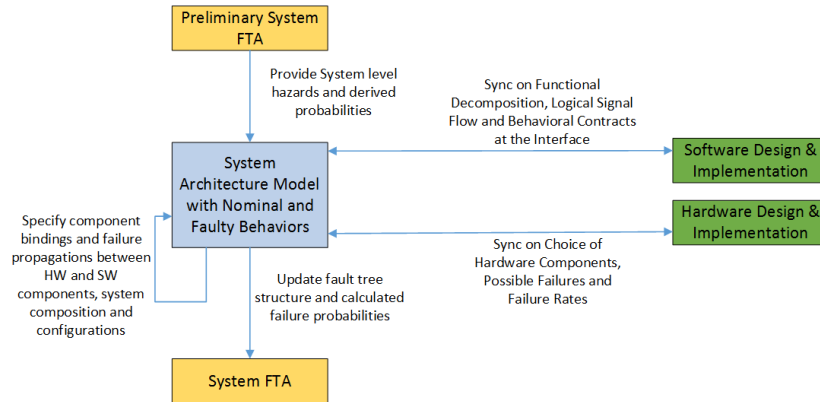


Fig. 2. Example Interactions between the Shared System/Safety Model and the FTAs

2.2 Modeling Language for System Design

The Architectural Analysis and Design Language (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” [?]. An AADL model describes a system in terms of a hierarchy of components and their inter-connections, 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 associated with properties and 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 semantics supports formal analysis tools that allow for early phase error/fault detection.

The Assume Guarantee Reasoning Environment (AGREE) [?] implements 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 on how the state of the component can evolve over time.

AGREE translates an AADL model and the behavioral contracts into Lustre [?] and then query a user-selected model checker to conduct the back-end analysis. The analysis is performed compositionally along the architecture hierarchy such that analysis at a higher level is only using the components and their behavioral contracts from its immediate lower level. This allows the analysis to scale for the design of large and complex systems.

In the avionics context, a physical system (e.g., a Line-Replaceable Unit) is represented by an AADL component, and the functions the physical system performs (e.g., input signal processing system function) is represented by the logical model elements in AADL (e.g., input and output data ports and the connections between the data ports) and the behavioral contracts in AGREE (e.g., assumptions on the inputs value range, constraints on the output value range, how outputs are calculated from inputs). Func-

tions and systems and their interactions are all captured in the same AADL/AGREE model.

AADL with the AGREE extension serves as a good candidate as the modeling language for describing the system design aspects of a shared system design and safety analysis model. Our prior work [?] adds the initial failure effect modeling capability to this language and tool set. However, the following goals are yet to be achieved before the combined language and tool set can be used to satisfy system safety objectives of ARP4754A [?] and ARP4761 [?]:

1. Being able to provide a comprehensive, qualitative description of the causal relationship between basic events and system level safety requirements.
2. Being able to provide an accurate, quantitative description of the contribution relationship between failure rates of the fault tree basic events and numerical probability requirements at the system level.

The remainder of the paper describes our approach towards both of the goals.

3 The Safety Annex

In this section, we describe the main features and functionality of the Safety Annex.

3.1 Basic Functionality

An AADL model of the nominal system behavior specifies the hardware and software components of the system and their interconnections. This nominal model is then annotated with assume-guarantee contracts using the AGREE annex [?] for AADL. The nominal model requirements are verified using compositional verification techniques based on k -induction model checking [?].

Once the nominal model behavior is defined and verified, the Safety Annex can be used to specify possible faulty behaviors for each component. The faults are defined on each of the relevant components using a customizable library of fault nodes and the faults are assigned a probability of occurrence. A probability threshold is also defined at the system level. This extended model can be analyzed to verify the behavior of the system in the presence of faults. Verification of the nominal model with or without the fault model is controlled through the safety analysis option during AGREE verification.

To illustrate the syntax of the Safety Annex, we use an example based on the Wheel Brake System (WBS) described in [?] and used in our previous work [?]. The fault library contains commonly used fault node definitions. An example of a fault node is shown below:

```
node fail_to(val_in: real, alt_val: real, trigger: bool) returns (val_out: real);
let
  val_out = if (trigger) then alt_val else val_in;
tel;
```

The *fail_to* node provides a way to inject a faulty input value. When the *trigger* condition is satisfied, the nominal component output value is overridden by the *fail_to*

```

annex safety {**
  fault pump_nondeterministic_out_fault "In pump: pressure_output failed closed.": faults.fail_to {
    inputs: val_in <- pressure_output.val,
           alt_val <- 0.0;
    outputs: pressure_output.val <- val_out ;
    probability: 1.0E-4 ;
    duration: permanent;
  }
**};

```

failure value. In the WBS, the pump component generates an expected amount of pressure to a hydraulic line. Declaration of a non-deterministic fault in the pump component is shown below:

The *fault statement* consists of a unique description string, the fault node definition name, and a series of *fault subcomponent* statements.

Inputs in a fault statement are the parameters of the fault node definition. In the example above, *val_in* and *alt_val* are the two input parameters of the fault node. These are linked to the output from the Pump component (*pressure_output.val*), and *alt_value*, a nondeterministic value defined within the Safety Annex. When the analysis is run, these values are passed into the fault node definition.

Outputs of the fault definition correspond to the outputs of the fault node. The fault output statement links the component output (*pressure_output.val*) with the fault node output (*val_out*). If the fault is triggered, the nominal value of *pressure_output.val* is overridden by the failure value output by the fault node. Faulty outputs can take deterministic or non-deterministic values.

Probability (optional) describes the probability of a fault occurrence.

Duration describes the duration of the fault; currently the Safety Annex supports transient and permanent faults.

3.2 Hardware Failures and Dependent Fault

Failures in hardware (HW) components can trigger behavioral faults in the software (SW) or system (SYS) components that depend on them. For example, a CPU failure may trigger faulty behavior in threads bound to that CPU. In addition, an fault in one HW component may trigger failures in other HW components located nearby, such as cascading failure caused by a fire or water damage.

Faults propagate in AGREE as part of a systems nominal behavior. This means that any propagation in the HW portion of an AADL model would have to be artificially modeled using data ports and AGREE behaviors in SW. This is less than ideal as there may not be concrete behaviors associated with HW components. In other words, faulty behaviors mainly manifest themselves on the SW/SYS components that depend on the hardware components.

To better model faults at the system level dependent on HW failures, we have introduced a new fault model element for HW components. In comparison to the basic fault statement introduced in the previous section, users are not specifying behavioral effects for the HW failures, nor data ports to apply the failure. An example of a model component fault declaration is shown below:

```

HW_fault valve_failed "Valve failed": {
  probability: 1.0E-5;
  duration: permanent;
}

```

In addition, users specify fault dependencies/propagations outside of fault statements and inside safety annex, typically in the system implementation where the system configuration that causes the dependencies (e.g., binding between SW and HW components, co-location of HW components) becomes clear. This is because fault propagations are typically tied to the way components are connected or bound together; this information may not be available when faults are being specified for individual components. Having fault propagations specified outside of a components fault statements also makes it easier to reuse the component in different systems. An example of a fault dependency specification is shown below:

```
annex safety{**
  analyze : max 1 fault
  propagate_from: {valve_failed@shutoff} to {pressure_fail_blue@selector};
**};
```

3.3 Architecture and Implementation

The architecture of the Safety Annex is shown in Figure 3. It 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 AGREE AADL annex and associated tools [?]. AGREE allows *assume-guarantee* behavioral contracts to be added to AADL components. The language used for contract specification is based on the Lustre dataflow language [?]. AGREE improves scalability of formal verification to large systems by decomposing the analysis of a complex system architecture into a collection of smaller verification tasks that correspond to the structure of the architecture.

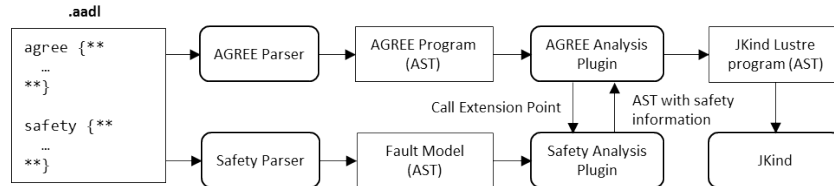



Fig. 3. Safety Annex Plug-in Architecture

AGREE contracts are used to define the nominal behaviors of system components as *guarantees* that hold when *assumptions* about the values the component's environment are met. The Safety Annex extends these contracts to allow faults to modify the behavior of component inputs and outputs. To support these extensions, AGREE implements an Eclipse extension point interface that allows other plug-ins to modify the generated abstract syntax tree (AST) prior to its submission to the solver. If the Safety Annex is enabled, these faults are added to the AGREE contract and, when triggered, override the nominal guarantees provided by the component. An example of a portion of an initial AGREE node and its extended contract is shown in Figure 4. The `__fault` variables

and declarations are added to allow the contract to override the nominal behavioral constraints (provided by guarantees) on outputs. In the Lustre language, *assertions* are constraints that are assumed to hold in the transition system.



```

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

agree node green_pump(
  time : real;
  __fault__nominal__pressure_output : common__pressure__i;
  fault_trigger_green_pump_fault_22 : bool;
  green_pump_fault_22_alt_value : real
) returns (
  pressure_output : common__pressure__i
);
var
  green_pump_fault_22_node_val_out : common__pressure__i;
let
  assertions {
    (green_pump_fault_22_node_val_out = pressure_output)
  }
  guarantees {
    "Pump always outputs something" :
      (__fault__nominal__pressure_output.val > 0.0)
  }
  green_pump_fault_22_node_val_out =
    faults_fail_to(
      __fault__nominal__pressure_output,
      green_pump_fault_22_alt_value,
      fault_trigger_green_pump_fault_22);
tel;

```

Fig. 4. Nominal AGREE node and its extension with faults

An annotation in the AADL model determines the fault hypothesis. This may specify either a maximum number of faults that can be active at any point in execution (typically one or two), or that only faults whose probability of simultaneous occurrence is above some probability threshold should be considered. In the former case, we assert that the sum of the true *fault_trigger* variables is 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). 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).

Once augmented with fault information, the AGREE model follows the standard AGREE translation path to the model checker JKind [?], an infinite-state model checker for safety properties. The augmentation includes traceability information so that when counterexamples are displayed to users, the active faults for each component are visualized.

4 Case Studies

To demonstrate the effectiveness of the Safety Annex, we describe two case studies.

4.1 Wheel Brake System

The Wheel Brake System (WBS) described in ARP4761 has been used as a case study for safety analysis, formal verification, and contract based design in numerous studies. In order to show scalability compare results with other tools and studies, the AADL model of the WBS used in [?] was enhanced using as a guide the NuSMV ARCH4 model as described in [?]. This version of the WBS model was chosen due to the complexity of the model and because this model addresses required safety concerns (for description of these concerns, see [?]). Due to the added complexity of this WBS system, we provide a short description of the subcomponents and behavior.

WBS architecture description The WBS is composed of two main systems: the control system and the physical system. The control system electronically controls the physical system and contains a redundant Braking System Control Unit (BSCU) in case of failure. The physical system consists of the hydraulic circuits running from hydraulic pumps to wheel brakes. This is what provides braking force to each of the 8 wheels of the aircraft.

Just as in the simple WBS model, there are three operating modes. In *normal* mode, the system uses the *green* hydraulic circuit. The normal system is composed of the green hydraulic pump and one meter valve per each of the 8 wheels. Each of the 8 meter valves are controlled through electronic commands coming from the BSCU. These signals provide brake commands as well as antiskid commands for each of the wheels. The braking command is determined through a sensor on the pilot pedal position. the antiskid command is calculated based on information regarding ground speed, wheel rolling status, and braking commands.

In *alternate* mode, the system uses the *blue* hydraulic circuit. The wheels are all mechanically braked in pairs (one pair per landing gear). The alternate system is composed of the blue hydraulic pump, four meter valves, and four antiskid shutoff valves. The meter valves are mechanically commanded through the pilot pedal corresponding to each landing gear. If the system detects lack of pressure in the green circuit, the selector valve switches to the blue circuit. This can occur if there is a lack of pressure from the green hydraulic pump, if the green hydraulic pump circuit fails, or if pressure is cut off by a shutoff valve. If the BSCU unit becomes invalid, the shutoff valve is closed.

The last mode of operation of the WBS is the *emergency* mode. This is supported by the blue circuit but operates 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.

WBS translation from NuSMV to AADL/AGREE The ARCH4 NuSMV model is annotated with LTL formulae for each of the subcomponent behavioral contracts. To assist in the process of translation from LTL into AGREE, a translation tool was developed. [Not sure how we want to cite this or if we should even include it here....](#)

During compositional analysis of the model, details of the system behavior contracts needed further clarification. For instance, the control system of the WBS consists of a BSCU which has two channels for redundancy. All of the electrical components

calculate system validity, electronic pedal commands, and braking commands to the wheels. The contracts specified in the BSCU and all subcomponents do take into account whether power is supplied to the control system. The higher level component Control System does not require power to be supplied in order to have correct behavior. In the AGREE behavioral model specifications, this needed to be added in order to complete compositional verification.

I think we should sum up the number of subcomponents, depth of the model in terms of layers, show time of verification for each of the compositional layers, number of fault nodes defined in total, and time it takes for fault analysis verification. I will create a little table that shows some of these values. Does anyone know if there is a way to easily count the subcomponents without manually counting? An option in aadl perhaps...?

Once the behavioral contracts were specified, both monolithic and compositional verification were run on the WBS model. There is a maximum layer depth of 6 for compositional verification and the time of verification was between 0 and 4 seconds. Monolithic verification was run on the top level of the system which took 24 seconds.

Fault Analysis of WBS using Safety Annex After the verification was completed, we defined faults equivalent to those described in the xSAP model for the NuSMV WBS system in [?]. A total of 33 fault definitions were given to 18 different components and the top level compositional verification was run with one permanent fault introduced into the system.

Compositional analysis on the top level WBS system was run with maximum one permanent fault present in the system. This caused the top level property *Inadvertant braking at the wheel* to fail. This was caused by a fault on the pedal position sensor. This sensor determines if the mechanical brakes are pressed. If so, it sends an electrical command to the BSCU to command braking. The fault was an inverted boolean fault which inverted the mechanical pedal value (false) and created a true electrical pedal value. Thus, we have a situation where no braking is commanded mechanically, but braking is commanded electronically nonetheless.

4.2 Quad-Redundant Flight Control System

In order to discuss Byzantine faults and Hardware dependencies, we applied the Safety Annex to the Quad-Redundant Flight Control System (QFCS) model [?]. Faulty behaviors were introduced in order to see the response of the system to several faults, and to evaluate fault mitigation logic in the model. The QFCS system-level properties failed when unhandled faulty behaviors were introduced.

We also used the Safety Annex to explore more complicated faults at the system level on a simplified QFCS model with cross-channel communication between its Flight Control Computers.

- Byzantine faults [?] were simulated by creating one-to-one connections from the source to multiple observers so that disagreements could be introduced by injecting faults on individual outputs. A system-level property failed due to the fault on the

baseline model, but did not fail on the model with Byzantine fault handling protocol added. Using the Safety Annex like this can test a system's vulnerability to Byzantine faults and verify mitigation mechanisms.

- Dependent failures in hardware were modeled by injecting faults to hardware components (physical layer), and faults to software components (logical layer) that are bound to the hardware components, then specifying fault propagations at the QFCS system level to indicate that the software faults are dependent on the hardware faults.

5 Related Work

In recent years, there has been an increase in the interest of Model Based Safety Analysis (MBSA) [?].

Formal model based systems engineering (MBSE) methods and tools now permit system level requirements to be specified and analyzed early in the development process [?, ?, ?, ?, ?]. Design models from which aircraft systems are developed can be integrated into the safety analysis process to help guarantee accurate and consistent results. This integration is especially important as the amount of safety critical hardware and software has dramatically increased in safety critical domains such as aerospace, automotive, and medical fields [?].

There are tools that currently support reasoning about faults in architecture description languages such as SysML and AADL. These tools include the AADL Error Model Annex, Version 2 (EMV2) [?] and HiP-HOPS for EAST-ADL [?]. These approaches primarily utilize *qualitative* reasoning. Faults are enumerated and the propagations through system components are explicitly described. Given many possible faults, these propagation relationships increase in complexity and understandability. Interactions are easily overlooked by analysts and thus not explicitly described. This is also the case with tools like SAML that incorporate both *qualitative* and *quantitative* reasoning [?].

In earlier work, an approach to MBSA was demonstrated using the Simulink notation [?, ?, ?, ?]. In this approach, a behavioral model of system dynamics was used to reason about the effects of faults in the system. We believe this approach allows an implicit and natural notion of fault propagation through the system. Since Simulink is not an architecture description language, notions such as hardware devices and non-functional aspects cannot be captured in system models. Using this idea of *quantitative* reasoning and implicit fault propagation, we wish to apply this to a more rich architecture language.

Formal verification tools based on model checking have been used to automate the generation of safety artifacts and used in safety critical system certification [?, ?, ?, ?, ?]. This approach has limitations in terms of scalability and readability of the fault trees generated. Work has been done to mitigate these limitations by the scalable generation of readable fault trees [?].

The Wheel Brake System (WBS) described in ARP4761 [?] has been used in the past as a case study for safety analysis, formal verification, and contract based design [?, ?, ?, ?, ?, ?]. The preliminary work for the safety annex used a simplified model of the WBS [?]. In order to show scalability and compare results with other studies, an AADL

version of the WBS was designed based off of arch4wbs NuSMV model described in previous work [?]. This model was chosen due to the number of subcomponents in the system and the complexity of behavior captured in the NuSMV model.

6 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. 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. Next steps will include extensions to automate injection of Byzantine faults as well as automatic generation of fault trees.

Acknowledgments. This research was funded by NASA contract NNL16AB07T and the University of Minnesota College of Science and Engineering Graduate Fellowship. The authors would like to thank Michael Peterson of Rockwell Collins for his invaluable and expert feedback.