Safety Annex for the Architecture Analysis and Design Language

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

[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
[3] Rockwell Collins
Commercial Systems Flight Controls Safety Engineering
Michael.Peterson@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. 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 [8, 20]. While model-based development methods are widely used in the aerospace industry, they are only recently being applied to system safety analysis.

In this paper, we describe a *Safety Annex* for the Architecture Analysis and Design Language (AADL) [23] 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) [20]. 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 [30] to the AADL modeling language, and provides a domain specific language for the kinds of analysis performed manually in previous work [41].

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 system 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/failures and system safety requirements. Thus, the contributions of the Safety Annex and this paper are:

- Close integration of behavioral fault analysis into the *architectural 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* through behavioral relationships in the model, in contrast to existing AADL-based annexes (HiP-HOPS and EMV2),
- additional support for *explicit* propagation of faults to capture binding relationships between hardware and software and logical and physical communications beyond what is supported in xSAP, 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 Safety Assessment Process

ARP4754A, the Guidelines for Development of Civil Aircraft and Systems [39], provides guidance on applying development assurance at each hierarchical level throughout the development life cycle of highly-integrated/complex aircraft systems, and has been recognized by the Federal Aviation Administration (FAA) as an acceptable method to establish the assurance process.

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 for meeting a company's internal safety standards. ARP4761, the Guidelines and Methods for Conducting Safety Assessment Process on Civil Airborne Systems and Equipment [38], 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 is intended to work, primarily focusing on the integrity of the outputs and the availability of the system. 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 the domain knowledge about the system, but do not necessarily have detailed knowledge of how the software functions are designed. Acquiring the required knowledge about the behavior and implementation of each software function in a system can be time-consuming.

For example, in a recent project it took one of our safety engineers two days to understand how the software in a Stall Warning System was intended to work. 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 at least as much time as was required to construct the safety analysis artifacts and perform the safety analysis itself. In another instance, it took a safety engineer several months to finalize the PSSA document for a Horizontal Stabilizer Control System, because of two major design revisions requiring multiple rounds of reviews with system, hardware, and software engineers to establish complete understanding of the design details.

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 to include *model based safety analysis* is under way. Capturing failure modes in models and generating safety analysis artifacts directly from models could greatly improve communication and synchronization between system designer and safety engineers, and provide the ability to more accurately analyze complex systems.

We believe that using a single unified model to conduct both system development and safety analysis can help reduce the gap in comprehending the system behavior and transferring the knowledge between the system designers and the safety analysts. It maintains a living model that captures the current state of the system design as it moves through the system development lifecycle. It also allows all participants of the ARP4754A process to be able to communicate and review the system design using a "single source of truth."

A model that supports both system design and safety analysis must describe both the system design information (e.g., system architecture, functional behavior) and safety-relevant information (e.g., failure modes, failure rates). It must do this in a way that

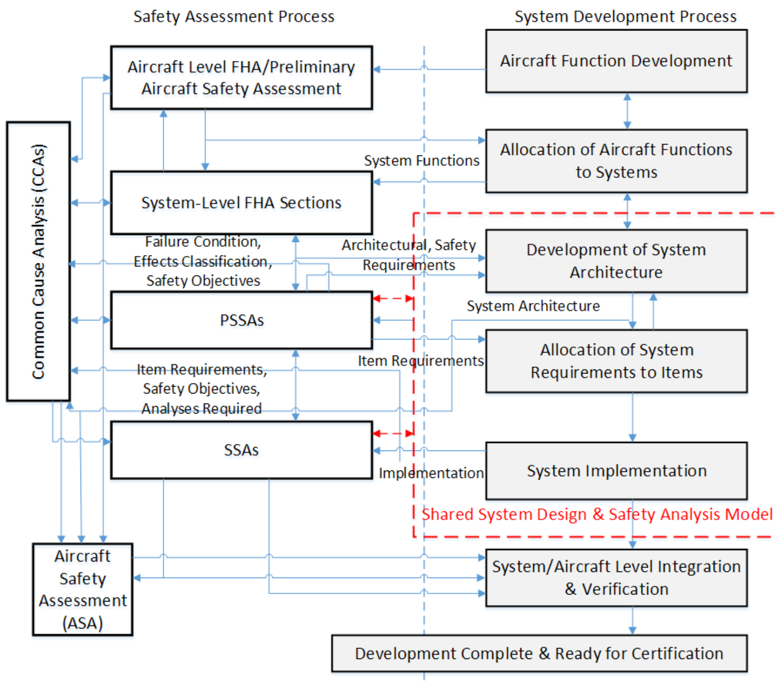

Fig. 1. Using the Shared System/Safety Model in the ARP4754A Safety Assessment Process

keeps the two types of information distinguishable, yet allows them to interact with each other.

Figure 1 presents our proposed use of this shared system design and safety analysis model in the context of the ARP4754A Safety Assessment Process Model (derived from Figure 7 of ARP4754A). The shared model is one of the system development artifacts 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 an interface to capture the information from the system design and implementation that is relevant for the safety analysis.

Figure 2 shows how the preliminary FTAs and final system FTAs (artifacts from the PSSA and SSA activities in the Safety Assessment Process) can guide and be updated from the shared model. The shared model is expected to be created and maintained in sync with the software and hardware design and implementation, and guided by the hazard and probability information from the preliminary system FTA. The analysis results from checking the system level properties on the shared model are then used to update the preliminary system FTA. This process continues iteratively until the system safety property is satisfied with the desired fault tolerance and failure probability achieved. The effort needed to update the final system FTA from the preliminary system FTA would be greatly reduced.

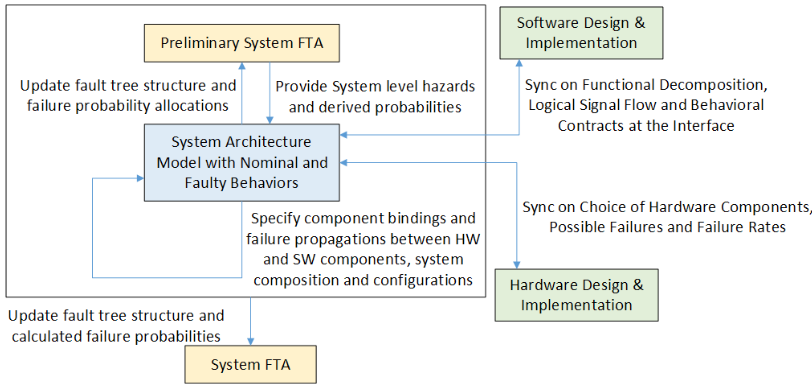

Fig. 2. Example Interactions between the Shared System/Safety Model and the FTAs

2.2 Modeling Language for System Design

We are using the Architectural Analysis and Design Language (AADL) [23] to construct system architecture models. AADL is an SAE International standard [2] 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 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) [20] is a tool for formal analysis of behaviors in AADL models. It is implemented as an AADL annex and annotates AADL components with formal behavioral contracts. Each component's contracts can include assumptions and guarantees about the component's inputs and outputs respectively, as well as predicates describing how the state of the component evolves over time.

AGREE translates an AADL model and the behavioral contracts into Lustre [27] 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.

In our prior work [41], 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.

3 The Safety Annex

In this section, we describe the main features and functionality of the Safety Annex. The usage of the terms error, failure, and fault follow their definitions in ARP4754A [39]. We use *fault* as the generic modeling keyword throughout the AADL model hierarchy.

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 [20] for AADL. The nominal model requirements are verified using compositional verification techniques based on inductive model checking [25].

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 [1] and used in our previous work [41]. 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* failure value. In the WBS, the pump component generates an expected amount of pressure to a hydraulic line. Declaration of a fail to zero fault in the pump component is shown below:

```
annex safety {**
    fault pump_closed_fault "In pump: pressure_output failed to zero.": 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;
    }
**};
```

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 fail to value of zero. 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 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 in AGREE AADL annex and associated tools [20]. 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 [27]. 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.

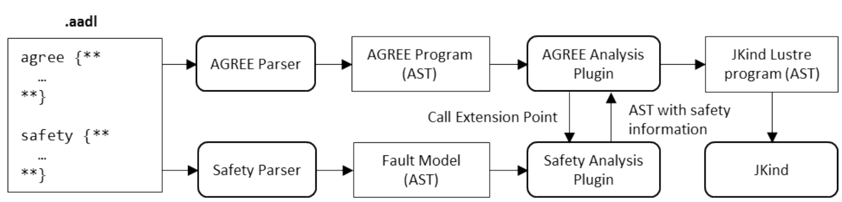

Fig. 3. Safety Annex Plug-in Architecture

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). 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).

Once augmented with fault information, the AGREE model follows the standard translation path to the model checker JKind [25], 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.

3.3 Comparing with EMV2 Modeling

In this section, we use a few examples from the WBS system to illustrate the differences in Safety Annex and EMV2 modeling. The AADL and EMV2 code examples are adapted from [42].

Below is the AADL component for the command function unit of the Braking and Steering Control Unit (BSCU) in the WBS system, and the error propagations and component error behavior specified in EMV2:

```
process command
    features
        pedalvalue: in data port common::command.pedal;
        brake: out data port common::command.brake;
        skid: out data port common::command.skid;
end command;

process implementation command.i
    annex EMV2 {**
        use types     error_library;
        use behavior  error_library::simple;
        error propagations
            pedalvalue      : in propagation {NoService};
            brake           : out propagation{NoValue};
            skid            : out propagation{NoValue};
        flows
            nopedal         : error sink pedalvalue{NoService};
            noskid          : error source skid{NoValue};
            nobrake         : error source brake{NoValue};
        end propagations;

        component error behavior
            transitions
                terrfrompedal : Operational -[pedalvalue{NoService}]-> Failed;
            propagations
                p1 : Failed -[]-> brake{NoValue};
                p2 : Failed -[]-> skid{NoValue};
        end component;
    **};
end command.i;
```

As seen in the EMV2 example, the error propagations are specified for the input and output signals, and the component error behavior describes how the component transitions to a failed state and what output errors can be propagated in that state.

Different from EMV2, the Safety Annex approach utilizes AGREE for the nominal behavior, as well as the propagations (no matter nominal or faulty). Therefore, error propagations are specified for the output signals in the Safety Annex; no error propagations for the input signals need to be explicitly specified. Furthermore, all error behaviors have to be concretely represented in terms of how they affect the output data flow, so that they can be analyzed by the back-end analysis engine. For example, adding the *service_mode* element to the *pedalvalue* output signal of the *pedal* component in AADL to model the "No Service" error, and adding the *presence* element to the *skid* and *brake* output signals of the *command* component in AADL to model the "No Value" error.

Below shows an example in AGREE capturing the relationship between the *service_mode* element of the *pedalvalue* input signal and the *presence* element of the *skid* and *brake* output signals:

```
annex agree{**
    guarantee "False service mode on pedalvalue causes false presence for brake and skid":
    not(pedalvalue.service_mode) => (not(brake.presence) and not(skid.presence));
**};
```

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 AIR6110 [1] is a well-known example that has been used as a case study for safety analysis, formal verification, and contract based design [8, 12, 13, 29]. The preliminary work for the safety annex used a simplified model of the WBS [41]. In order to demonstrate scalability of our tools and compare results with other studies, we constructed a functionally and structurally equivalent AADL version of one of the most complex WBS xSAP models (arch4wbs) described in [13].

WBS architecture description The WBS is composed of two main parts: 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.

There are three operating modes in the WBS model. 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 channel 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.

The model contains 30 different kinds of components, 169 component instances, a model depth of 5 hierarchical levels. The model includes one top-level assumption and 11 top-level system properties, with 113 guarantees allocated to subsystems. There are a total of 33 different fault types and 141 fault instances within the model. The large number of fault instances is due to the redundancy in the system design and its replication to control 8 wheels.

An example property is to ensure no inadvertent braking of each of the 8 wheels. This means that if all power and hydraulic pressure is supplied (i.e., braking is commanded), then either the aircraft is stopped (ground speed is zero), or the mechanical pedal is pressed, or brake force is zero, or the wheel is not rolling.

Fault Analysis of WBS using Safety Annex Fault analysis on the top level WBS system was performed on the 11 top-level properties using two fault hypotheses. In the first case, we allow at most one fault, and in the second we allow combinations of faults that exceed the acceptable probability for the top-level hazard defined in AIR6110.

We use this model to demonstrate the benefits of formal fault analysis and to show the scalability of our tools. We applied both *monolithic* analysis, in which the entire model is flattened and analyzed at once, and also *compositional* analysis, where each architectural layer is analyzed hierarchically. For the fault-free "nominal" system model, monolithic analysis requires 21 seconds, whereas compositional analysis requires 1 minute and 53 seconds. Although the compositional time is longer, each sub-problem completes in less than 5 seconds. The additional time for compositional analysis is due to the start-up overhead to invoke the JKind model checker many times for individual layers. On the other hand, when examining the model under a single-fault hypothesis, compositional analysis requires 2 minutes 6 seconds, while monolithic analysis did not terminate after 60 minutes.

For probabilistic fault hypotheses, we are currently developing a sound approach for composition with respect to the top-level fault probability, but our current tool requires monolithic analysis. In this case, given a probabilistic fault hypothesis of $5 * 10^{-7}$, monolithic analysis requires 3 minutes 25 seconds.

During our analysis, we discovered that most properties were verified, but the *Inadvertent braking at the wheel* properties are not resilient to a single fault nor do they meet the desired 10^{-9} fault threshold for probabilistic analysis. In our model (as in the NuSMV model [13]), there is a single pedal position sensor for the brake pedal. If this sensor fails, it can command braking without a pilot request. Given the *counterexample* returned by the tools, it is straightforward to diagnose the fault conditions that lead to property failure.

This counterexample can be used to further analyze the system design. For our model, there are several possible reasons for failure: it could be that redundant sensors are required on the pedals (here we note that the architecture of the pedal assembly is not discussed in AIR6110), or that the phase of flight is sufficiently short that we need to adjust our pedal failure rate to match this phase of flight, rather than normalizing the failure rate to per-flight-hour. It is straightforward and computationally inexpensive to run the analysis, allowing quick iterations between systems and safety engineers. As indicated in Figure 2, the sync and update between the preliminary system FTA and the architecture/analysis model continues until the system safety property is satisfied with the desired fault tolerance and failure probability achieved.

4.2 Quad-Redundant Flight Control System

We have also used the Safety Annex to examine more complex fault types, such as asymmetric (or *Byzantine*) faults. A Byzantine fault presents different symptoms to different observers, so that they may disagree regarding whether a fault is present. We extended the Quad-Redundant Flight Control System (QFCS) example [3] to model and analyze various types of faulty behaviors. Faulty behaviors were introduced to analyze the response of the system to multiple faults, and to evaluate fault mitigation logic in the model. As expected, 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 [22] 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. The system level property "at most one flight control computer in command" was falsified in one second in the presence of Byzantine faults on the baseline model. The same property was verified in three seconds on an extended model with a Byzantine fault handling protocol added. System designers can use this approach to verify if a system design is resilient to Byzantine faults, examine vulnerabilities, and determine if a mitigation mechanism works.
- Dependent faults were modeled by first injecting failures to the cross-channel data link (CCDL) bus (physical layer), and faults to the flight control computer (FCC) outputs (logical layer), then specifying fault propagations in the top level system implementation (where the data connections between FCC outputs were bound to the CCDL bus subcomponents). The fault propagation indicates that one CCDL bus failure can trigger all FCC output faults. With the fault hypothesis that allows a maximum of one fault active during execution, the system level property "not all FCCs fail at the same time" was falsified in one second.

5 Related Work

Formal model based systems engineering (MBSE) methods and tools now permit system level requirements to be specified and analyzed early in the development process [3, 19, 20, 34]. Design models from which aircraft systems are developed can be integrated into the safety analysis process to help guarantee accurate and consistent results. Integration of MBSA into safety analysis process is described by Bozzano and Villafiorita [16]. There are tools that currently support reasoning about faults in architecture description languages such as SysML and AADL. We provide here a brief overview of the most relevant safety analysis tools.

Tools such as the AADL Error Model Annex, Version 2 (EMV2) [24] and HiP-HOPS for EAST-ADL [17] 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. In our approach, faults are injected into the system and behaviorally propagated through the use of assume-guarantee statements in AGREE. This avoids the difficulties inherent with explicit fault enumeration and propagation.

SmartIFlow [28] is a purpose-built 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 face of failures. The mechanism to keep the search space size under control during model checking relies on expert knowledge from engineers. This limits the number of failures and removes the possibility of certain failure conditions. Due to this drawback, scalability to industrial sized systems is difficult. The safety annex described in this research is not a standalone model, but is made to be incorporated into the system safety assessment process as described in section 2.1. Scalability: in terms of the safety annex, just reference the case study described in this work?

Another approach has been introduced by Güdemann et. al. [26]. System models are constructed in SAML (Safety Analysis and Modeling Language) which can be imported into several analysis tools like NuSMV [18] or PRISM [32] (Probabilistic Symbolic Model Checker) or MRMC probabilistic model checker [31]. SAML is used for both qualitative and quantitative analyses and allows for the combination of discrete probability distributions and non-determinism. The framework is used to create a system model comprised of software control, hardware components, environment and failure mode modeling.

In earlier work, an approach to MBSA was demonstrated using the Simulink® notation [29, 30]. In this approach, a behavioral model of system dynamics was used to reason about the effects of faults in the system. This approach allows an implicit and natural notion of fault propagation through the system. However, non-functional architectural properties were not captured as Simulink is not designed as an architecture description language. In our approach, we are applying quantitative reasoning and implicit fault propagation to a more rich architecture language.

Similarly, AltaRica [36] has been incorporated into Cecilia OCAS as a model based safety analysis tool [5]. Safety assessment, fault tree generation, and functional verification can be performed with the aid of NuSMV model checking [11]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [4]. A limitation of this is that Linear Temporal Logic operators are required in some of the failure definitions. This is a downfall to the safety community/engineers who are not familiar with LTL [4].

Closely related to our work is the model-based safety assessment toolset called COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [9]. COMPASS uses the SLIM language which is based on AADL, for its input models. The SLIM (System Level Integrated Modeling Language) language was developed by the COMPASS project for modeling hardware and software systems for safety-related tasks [10, 14]. The 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 [18, 35], MRMC (Markov Reward Model Checker) [31, 33], and RAT (Requirements Analysis Tool) [37]. The safety analysis tool xSAP [6] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [7]. While it is clear that behavioral contracts can be specified in the SLIM model through the use of assume-guarantee statements, the focus of the tool and examples provided is on explicit fault propagation much like EMV2 AADL error annex [21]. Our approach is different in that the focus is not on explicit fault propagation, but instead leveraging the nominal model behavior in order to view the system behavior in the presence of faults.

Formal verification tools based on model checking have been used to automate the generation of safety artifacts [6,11,15]. This approach has limitations in terms of scalability and readability of the fault trees generated. Work has been done towards mitigating these limitations by the scalable generation of readable fault trees [12].

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. For more details on the tool, models, and approach, see the technical report [40].

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. P. Bieber, C. Bougnol, C. Castel, J. pierre Heckmann, C. Kehren, S. Metge, C. Seguin, P. Bieber, C. Bougnol, C. Castel, J. p. Heckmann, C. Kehren, and C. Seguin. Safety assessment with altarica - lessons learnt based on two aircraft system studies. In *In 18th IFIP World Computer Congress, Topical Day on New Methods for Avionics Certification*, page 26, 2004.
5. 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.
6. 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*, pages 533–539, 2016.

7. 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.
8. M. Bozzano, A. Cimatti, A. Griggio, and C. Mattarei. Efficient Anytime Techniques for Model-Based Safety Analysis. In *Computer Aided Verification*, 2015.
9. M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The compass approach: Correctness, modelling and performability of aerospace systems. In B. Buth, G. Rabe, and T. Seyfarth, editors, *Computer Safety, Reliability, and Security*, pages 173–186, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
10. M. Bozzano, A. Cimatti, J.-P. Katoen, V. Yen Nguyen, T. Noll, and M. Roveri. Model-based codesign of critical embedded systems. 507, 01 2009.
11. M. Bozzano, A. Cimatti, O. Lisagor, C. Mattarei, S. Mover, M. Roveri, and S. Tonetta. Symbolic Model Checking and Safety Assessment of Altarica Models. In *Science of Computer Programming*, volume 98, 2011.
12. M. Bozzano, A. Cimatti, C. Mattarei, and S. Tonetta. Formal safety assessment via contract-based design. In *Automated Technology for Verification and Analysis*, pages 81–97, 2014.
13. M. Bozzano, A. Cimatti, A. F. Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta. Formal Design and Safety Analysis of AIR6110 Wheel Brake System. In *CAV 2015, Proceedings, Part I*, pages 518–535, 2015.
14. M. Bozzano, A. Cimatti, M. Roveri, J. P. Katoen, V. Y. Nguyen, and T. Noll. Codesign of dependable systems: A component-based modeling language. In *2009 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, pages 121–130, July 2009.
15. M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic fault tree analysis for reactive systems. In *ATVA*, pages 162–176, 2007.
16. M. Bozzano and A. Villafiorita. *Design and Safety Assessment of Critical Systems*. Auerbach Publications, Boston, MA, USA, 1st edition, 2010.
17. 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.
18. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, Mar 2000.
19. A. Cimatti and S. Tonetta. Contracts-refinement proof system for component-based embedded systems. *SCP*, 97:333 – 348, 2015. SEAA 12.
20. D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional Verification of Architectural Models. In *NFM 2012*, volume 7226, pages 126–140, April 2012.
21. COMPASS Users Manual. http://www.compass-toolset.org/docs/compass-manual.pdf.
22. K. Driscoll, H. Sivencrona, and P. Zumsteg. Byzantine Fault Tolerance, from Theory to Reality. In *SAFECOMP*, volume 2788 of *LNCS*, pages 235–248, 2003.
23. P. Feiler and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.
24. P. Feiler, J. Hudak, J. Delange, and D. Gluch. Architecture fault modeling and analysis with the error model annex, version 2. Technical Report CMU/SEI-2016-TR-009, Software Engineering Institute, 06 2016.
25. A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The JKind Model Checker. *ArXiv e-prints*, Dec. 2017.
26. M. Gudemann and F. Ortmeier. A framework for qualitative and quantitative formal model-based safety analysis. In *HASE 2010*, pages 132–141, 2010.
27. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. In *IEEE*, volume 79(9), pages 1305–1320, 1991.

28. P. Hönig, R. Lunde, and F. Holzapfel. Model Based Safety Analysis with smartIflow. *Information*, 8(1), 2017.
29. A. Joshi and M. P. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In *SAFECOMP*, volume 3688 of *LNCS*, page 122, 2005.
30. A. Joshi, 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.
31. 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, pages 243–, Washington, DC, USA, 2005. IEEE Computer Society.
32. 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.
33. MRMC: Markov Rewards Model Checker. http://wwwhome.cs.utwente.nl/ zapreevis/m-rmc/.
34. A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. Heimdahl. Compositional Verification of a Medical Device System. In *HILT 2013*. ACM, November.
35. NuSMV Model Checker. http://nusmv.itc.it.
36. T. Prosvirnova, M. Batteux, P.-A. Brameret, A. Cherfi, T. Friedlhuber, J.-M. Roussel, and A. Rauzy. The AltaRica 3.0 Project for Model-Based Safety Assessment. *IFAC*, 46(22):127 – 132, 2013.
37. RAT: Requirements Analysis Tool. http://rat.itc.it.
38. SAE ARP 4761. Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.
39. SAE ARP4754A. Guidelines for Development of Civil Aircraft and Systems, December 2010.
40. 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.
41. D. Stewart, M. Whalen, D. Cofer, and M. P. Heimdahl. Architectural Modeling and Analysis for Safety Engineering. In *IMBSA 2017*, pages 97–111, 2017.
42. The Software Engineering Institute (SEI). ARP4761 - Wheel Brake System (WBS) Example. `https://wiki.sei.cmu.edu/aadl/index.php/ARP4761_-_Wheel_Brake_System_%28WBS%29_Example`.