Written Preliminary Examination Report: Architectural Modeling and Analysis for Safety Engineering

Danielle Stewart
Department of Computer Science & Engineering
University of Minnesota
MN, USA
dkstewar@umn.edu

ABSTRACT

This paper describes a method with tool support for model-based safety analysis. The tool support 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

Safety critical systems are an increasingly important topic in our modern age. From nuclear power plants and airplanes to heart monitors and automobiles, critical systems are vitally important in our society. With a rise in technological advances also comes an increased need for reliable safety analysis methods and tools. Standardized methods of safety analysis have been used for many years and various tools and methods are currently applied to the topic in both industrial and academic settings. 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 system development process [11,23]. While model-based development methods are widely used in the aerospace industry, they are only recently being applied to system safety analysis.

This paper describes a behavioral approach to safety analysis using an architecture description language called Architecture Analysis and Design Language. We describe a *Safety Annex* for the Architecture Analysis and Design Language (AADL) [26] 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) [23]. 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 [34] to the AADL modeling language, and provides a domain specific language for the kinds of analysis performed manually in previous work [47].

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.

The organization of this paper is as follows. Section 2 provides the necessary background in safety analysis, model based safety analysis, and the safety assessment process. Section 3 outlines the Safety Annex tool and usage. Case studies are shown in Section 5 followed by Future Work in Section 6 and finally a conclusion.

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 critical systems. Therefore, we need to understand how the tools and the models will fit into the existing safety assessment and certification process. Part of this understanding involves taking a look at pertinent background information in safety analysis.

2.1 Safety Critical Systems

A safety critical system is a system whose safety cannot be shown solely by test, whose logic is difficult to comprehend without the aid of analytical tools, and whose failure can directly or indirectly cause significant loss of life or property [43]. Guaranteeing safety and reliability of safety critical systems is mandatory. The process that guides this guarantee is highly standardized and controlled [1, 43]. Due to the complexity of critical systems, the field of safety analysis has in recent decades turned to formal methods [19,37]. In practice, a systems behavior can be described in a variety of ways that include diagrams, textual descriptions, and operational procedures [44]. These descriptions must be clear and well defined in order to avoid ambiguous interpretation. The formal definition of system behavior has a unique interpretation and is therefore a good candidate for automated analysis in order to validate requirements and spot design flaws [34].

Model checking is a technique used to allow exhaustive and automatic checking of whether a system model (formal system defi-

nition) meets a set of formal requirements. As early as the '90's, using model checking for safety requirements began to surface in the critical systems literature [6, 22]. Current tools in safety analysis use model checking techniques during the development and assessment of safety critical systems, [9, 11, 14, 37].

2.2 Model Based Safety Analysis

Safety engineers traditionally perform safety analysis based on information synthesized from a variety of sources including informal design models and requirement documents. These analyses are highly subjective and dependent on the skill of the analyst. The lack of precise models requires the analyst to devote a fair amount of time to information gathering of the architecture and behavior of the system. On the other hand, in Model Based Safety Aanalysis (MBSA), the system and safety engineers share a common system model created using the model based development process. By extending the system model and relevant physical control systems, automated support can be provided for much of the safety analysis. Using a common model for both system and safety engineering and automating parts of safety analysis assists in the reduction of cost and improves the quality of the safety analysis, but this is not without disadvantages if the model itself is faulty.

In model based system development, various development activities such as simulation, verification, testing, and code generation are based on a formal model of the system under development [34]. This is called the nominal model. Model based development was extended to include model based safety analysis [11, 16, 32–34]. The goal of MBSA is to incorporate safety analysis into the model based development process to provide information on the safety of the formal model of the system under development. In this process, the nominal (non-failure) system behavior that is captured in the model based development process is augmented with the fault behavior of the system. Model based safety analysis then operates on a formal model that describes both nominal system behavior and the fault model, which describes fault behavior.

2.3 Faults, Errors, and Failures

The usage of the terms error, failure, and fault are defined in ARP4754A [44] and are described here for ease of understanding. A *failure* is an event that occurs when the delivered service of a system deviates from correct behavior. A service of a system is a sequence of the system's external states. This means that if a service failure occurs, one or more of the systems external states has deviated from the correct service state. This deviation is called an *error*. The complete definition of an error is the part of the state of the system that may cause a failure. Not all errors will affect the external state of a system and cause an failure. The cause of an error is a *fault*. Faults can be *active* or *dormant*. A fault is active when it causes an error to occur, else it is dormant.

The relationship between faults, failures, and errors were summarized by Aviyienis, et. al. [4] and are described here once again.

- A fault is active when it causes an error. An active fault can be an internal fault that was previously dormant but has been activated or it can be an external fault. Fault activation is what causes a dormant fault to become active.

- Error propagation in a component is caused by the computation process of that component. In this process, an error is successfully transformed into other errors. Error propagation from component A to component B occurs when an error reaches the service interface of component A. The service delivered from component A to component B is no longer correct and thus the error is propagated into component B through the interface between the two components.

- A service failure occurs when that error is propagated to the service interface and causes the service of the system to be incorrect. The failure of a component causes a fault in the system that contains the component. Service failure of a system will cause external fault(s) for the other systems that recieve service from the given system.

2.4 Safety Artifacts

The fault behavior of the model is commonly analyzed using techniques described and formalized in guidelines such as ARP4761 and AIR6110 and the most relevant of these techniques are Fault Tree Analysis (FTA) and Fault Mode and Effects Analysis (FMEA) [2, 19, 42, 43].

2.4.1 Fault Tree Analysis

A Fault Tree (FT) is a directed acyclic graph whose leaves model component failures and whose gates model failure propagation. The system failure under examination is the root of the tree and is called the Top Level Event (TLE). The Basic Events (BE) are the events that can occur in the system which lead to the TLE and in the graphical model, these correspond to the leaves. 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 1, the AND gate has three inputs and one output. The leaves of the tree represent the basic events of the system and in the case of this fault tree, these three events are also the *Minimum Cut Set* for this top level event. The Minimal Cut Set (MCS) is the minimal set of basic events that must occur together in order to cause the TLE to occur. Finding these sets 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 [25, 42].

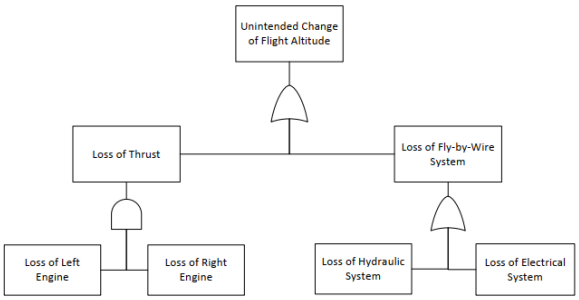

Figure 1: A Simple Fault Tree

Figure 1 shows a simple example of a fault tree. In this example, the top level event corresponds to an aircraft having an unintended change of altitude. In order for this event to occur, there must be either a loss of thrust or the loss of a Fly-by-Wire system. This is seen through the use of the OR gate below the top level event. The malfunction of both the left and right engines will cause the loss of thrust to occur and the Fly-by-Wire system can be lost if either the hydraulic system or the electrical system were to malfunction. The MCSs for this example are {Loss of Left Engine, Loss of Right Engine}, {Loss of Hydraulic System}, and {Loss of Electrical System}.

There are two main types of FTA that we differentiate here as *qualitative* analysis and *quantitative* analysis. In qualitative analysis, the structure of the fault tree is considered and the cut sets 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 occurance of the basic events.

2.4.2 *Fault Mode and Effects Analysis*

FMEA is represented as a table that shows the relationships between sets of faults, intermediate events, and properties that represent undesirable states. Even though FMEA tables are different than fault trees, the generation of MCSs are often used to build the FMEA tables [19]. For instance, the FMEA table in Figure 2 represents the MCSs also found in the FT from Figure 1.

Fault Configuration	Intermediate Events	Top Level Events
Loss of Hydraulic System	Loss of Fly-by-Wire System	Unintended Change of Flight Altitude
Loss of Electrical System	Loss of Fly-by-Wire System	Unintended Change of Flight Altitude
Loss of Left Engine and Loss of Right Engine	Loss of Thrust	Unintended Change of Flight Altitude

Figure 2: A Simple FMEA Table

2.4.3 *Safety Artifacts in the MBSA Process*

The artifacts developed from the MBSA process can be updated iteratively from the shared nominal and fault model of the system. It is assumed that the model is created and maintained in sync with the hardware and software design and implementation and guided by the hazard and probability information from a preliminary system FTA. The analysis results for each iteration of the 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.

2.5 Safety Assessment Process

ARP4761, the Guidelines and Methods for Conducting Safety Assessment Process on Civil Airborne Systems and Equipment, 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 [43].

The safety assessment process is part 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. The guidelines presented in ARP4761 include various industry accepted safety assessment practices. They are summarized here for convenience.

- Functional Hazard Assessment (FHA): This process examines aircraft and system functions to identify potential functional failures and classifies the hazards associated with specific failure conditions. This is usually developed early in the development process and is updated throughout.
- Preliminary System Safety Assessment (PSSA): This will establish the system safety requirements and provide some indication that the system architecture can meet those safety requirements. This is also updated throughout the development process.
- System Safety Assessment (SSA): This process collects, analyzes, and documents verification that the system, as implemented, meets the safety requirements established by the PSSA.
- Common Cause Analysis (CCA): The CCA establishes physical and functional separation, isolation, and independence requirements between systems and verifies that these requirements have been met.

As shown in Figure 3, these processes occur during the development process and are continually updated throughout. 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. 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.

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. This maintains a living model that captures the current state of the system design as it moves through the system development lifecycle.

A single unified model also allows all participants of the ARP-4754 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 keeps the two types of information distinguishable, yet allows them to interact with each other.

Figure 3 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. This is seen as a box labeled "Shared System Design and Safety model" on the right column of the figure. 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.

3. THE SAFETY ANNEX

The usage of the terms error, failure, and fault follow their definitions in Section 2.3. We use *fault* as the generic modeling keyword throughout the AADL model hierarchy.

The Safety Annex Users Guide can be found at https://github.com/loonwerks/AMASE/tree/develop along with the tool plugins and examples described in this report.

3.1 Modeling Language for System Design

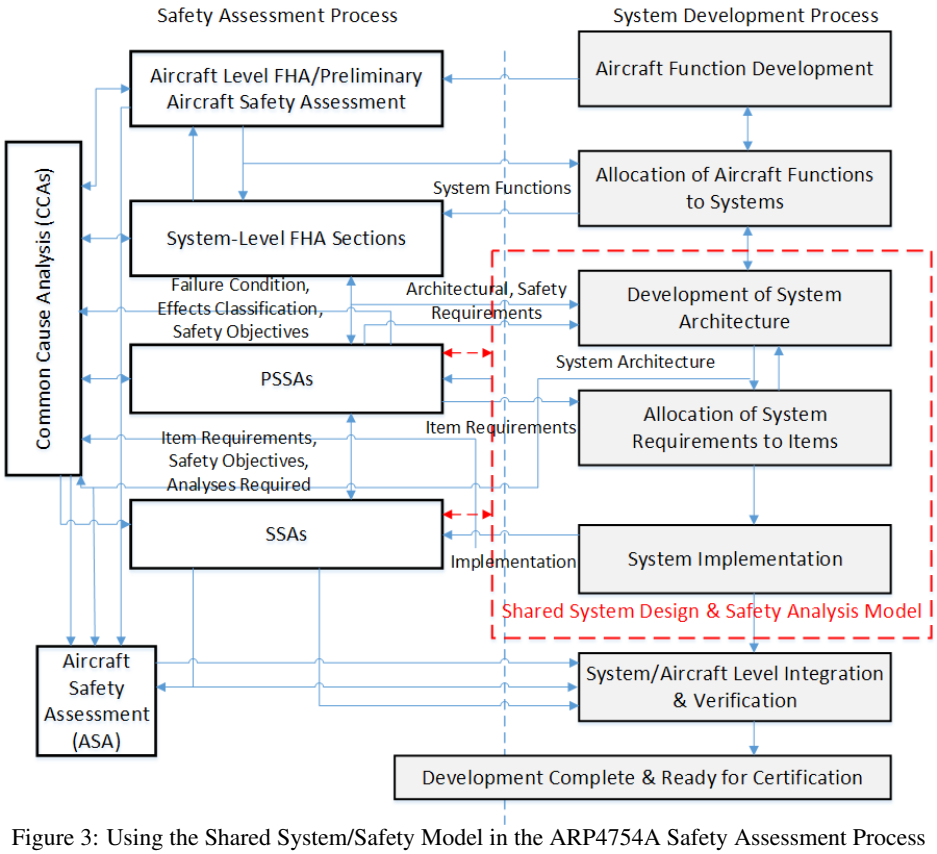

Figure 3: Using the Shared System/Safety Model in the ARP4754A Safety Assessment Process

We are using the Architectural Analysis and Design Language (AADL) to construct system architecture models [26]. AADL is an SAE International standard that defines a language and provides a unifying framework for describing the system architecture for "performance-critical, embedded, real-time systems" [3]. From its conception, AADL has been 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.

3.2 Assume-Guarantee Reasoning

The Assume Guarantee Reasoning Environment (AGREE) is a tool for formal analysis of behaviors in AADL models [23]. It is implemented as an AADL annex and annotates AADL components with formal behavioral contracts. Each component's contracts can include assumptions about the component's inputs and guarantees about the component's outputs as well as predicates describing how the state of the component evolves over time.

AGREE contracts can be used to capture the functional requirements at each level of the hierarchy. Once the model has been reviewed and the requirements captured have been validated, the back-end analysis can be conducted to verify if each level of the model implements its higher level requirements correctly.

AGREE translates an AADL model and the behavioral contracts into Lustre [30] and then queries a user-selected model checker to conduct the back-end analysis. This analysis can be performed compositionally or monolithically. In monolithic analysis, the model is flattened and the top level property is proved using the leaf level component properties. Due to the state space explosion problem inherent in such large scale verification problems, this method does not scale well. On the other hand, compositional analysis begins with the lowest level components and seeks to prove the properties one level above. The analysis continues breaking the large problem into many smaller problems. This approach allows the analysis to scale to much larger systems [5, 23].

In prior work, an initial failure effect modeling capability was added to the AADL/AGREE language and tool set [47]. This work is an extension in hopes that this tool can be used to satisfy system safety objectives of ARP4754A and ARP4761.

3.3 Basic Functionality of the Safety Annex

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 inductive model checking [28].

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

```
system pump
features
    pressure_output : out data port common::pressure.i;

    annex AGREE {**
        guarantee "Pump always outputs something" : pressure_output.val > 0.0 ;
    **} ;

end pump;
```

Figure 4: Nominal Pump Model with AGREE contract

```
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;
    }
```

Figure 5: Pump Fault Definition

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. An example of this is shown in Figure 4 as a pump component. In this simple example, the pump has an output and the contract is that the pump output is always positive.

To illustrate the syntax of the Safety Annex, we use an example based on the Wheel Brake System (WBS) described in AIR6110 standards document and used in previous work [2, 47].

The *fault library* contains commonly used fault node definitions. Commonly used faults in various systems include *fail_to* fault, where the output of a component fails to a particular value, for instance a pump may fail to produce any output and hence the fail to value is zero. Another example of a commonly occuring fault is an *inverted_boolean*. For example, a sensor may have an inverted value fault present on its boolean output. The fault library is simply a collection of fault nodes with these commonly used fault definitions which are then easily referenced within the fault definitions in the safety annex.

A *fault declaration* is shown in Figure 5. This describes a fail to zero fault in a pump. The *fail_to* node provides the mechanism in which a faulty value is injected into the fault model. When the *trigger* condition for this pump fault is satisfied, the nominal component output value is overridden by the failure value. In this particular example, the pump will fail to produce any pressure and the hydraulic fluid ceases being pumped through the lines.

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

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 permanent faults.

3.4 Hardware Failures and Dependent Faults

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, a failure 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 the nominal behavior of a system. 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 software components that depend on the hardware components.

To better model faults at the system level dependent on HW failures, there is a fault definition specifically designed 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 hardware component fault declaration is shown in Figure 6.

When analyzing hardware specific faults, it is often of interest to see how these faults may propagate to other components of the system, specifically the software components. As an example, con-

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

Figure 6: Hardware Fault in the Safety Annex

sider software systems that run on a particular CPU. This CPU has a hardware fault pertaining to overheating. An important question is how this hardware specific fault may propagate to components that rely on the CPU. In the nominal model, there is a binding between hardware and software components in AADL that is specified within the system implementation. In order to model this type of fault, users can specify fault dependencies in the system implementation where these dependencies are clearly defined. This is the only case of explicit fault propagation within the Safety Annex and in this case the propagation only is from the hardware to the software that runs on it.

3.5 Implementation Details

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 AGREE 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 [30].

The organization of these interactions is shown in Figure 7. The AADL file is annotated with AGREE contracts which creates the nominal model. This nominal model is parsed and sent to JKind where formal verification is performed. If the Safety Annex is also annotated within this model, this creates the extended (fault) model. When desired, the user can run "Perform Safety Analysis" within the Osate environment. This creates an extension point where the Safety Analysis informaiton is added into the AGREE model before getting passed to JKind for analysis. In either case, the JKind verification result is returned and displayed to the user in the Osate environment. For more information about this process or figures depicting the environment and analysis results, see the Safety Annex Users Guide or the Safety Annex Technical Report [45, 46].

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 [2] is a well-known example that has been used as a case study for safety analysis, formal verification, and contract based design [11, 15, 16, 32,37]. The preliminary work for the safety annex used a simplified model of the WBS [47]. In order to demonstrate scalability of our tools and compare results with other studies, a functionally and structurally equivalent AADL version was constructed of one of the most complex WBS xSAP models (arch4wbs) [16].

The Aerospace Information Report 6110 (AIR6110) document provides an example of a single aircraft system, namely the braking system, for the hypothetical passenger aircraft model S18. The two engine passenger aircraft is designated to carry up to 350 passengers for an average flight time of 5 hours. The purpose of the system is to provide a clear example of systems development and its analysis using the methods and tools described in ARP4754A/ED-79A. This brake system implements the aircraft function *"Decelerate aircraft on the ground (stopping on the runway)"*. The wheel brake system architecture is shown in Figure 8 and is taken from the work of Mattarei [37].

4.1.1 WBS overview and architecture description

The WBS is a hydraulic braking system that provides braking of left and right landing gears, each of which have four wheels. Each landing gear can be individually controlled by the pilot through left/right brake pedals.

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. In addition to the redundant BSCU channel, the control system is composed of a number of logical components including sensors for the wheels and brake pedal position, a monitor system that checks validity of the BSCU channel, and the command system which commands braking for each of the 8 wheels. The control system is primarily used in the normal mode of operation to command brake pressure.

The physical system consists of the hydraulic circuits running from hydraulic pumps to wheel brakes. This circuit contains the pumps for both normal and alternate modes of operation (named green and blue lines respectively), a selector valve which selects the circuit depending on input from the BSCU, meter valves at each wheel. These are the physical components that provide braking force to 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. In the normal mode of operation, the selector valve uses the green hydraulic pump to supply fluid to the wheels. Each of the 8 wheels has one meter valve which 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,

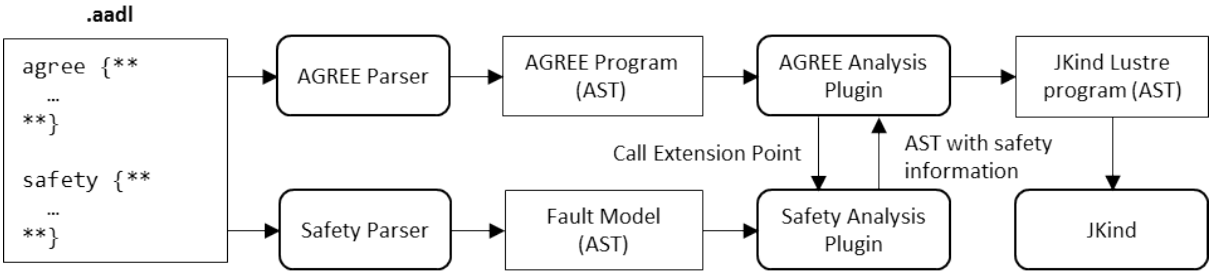

Figure 7: Safety Annex Plug-in Design

or brake force is zero, or the wheel is not rolling.

4.1.2 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 illustrate the use 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 (maximum one fault present in the system), compositional analysis requires 2 minutes 6 seconds, while monolithic analysis did not terminate after 60 minutes. The reason for this lies in the difference between compositional analysis and monolithic analysis as described in Section 3.2.

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 [16]), 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 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. The sync and update between the preliminary system artifact analysis (FTA, etc.) and the architecture 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 *Byzantine* faults. A Byzantine fault presents different symptoms to different observers, so that they may disagree regarding whether a fault is present. For example, a sensor may appear to be both failed and functional to different components to which it provides signal.

The Quad-Redundant Flight Control System (QFCS) was first developed in Simulink and then manually translated in AADL by Backes, et. al. This system consists of four cross-checking flight control computers (FCC). The work done by Backus formalized the requirements for components of the system and for more detail on the requirements specification, see the original paper [5].

The QFCS nominal system model was extended 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 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 [24]. 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.

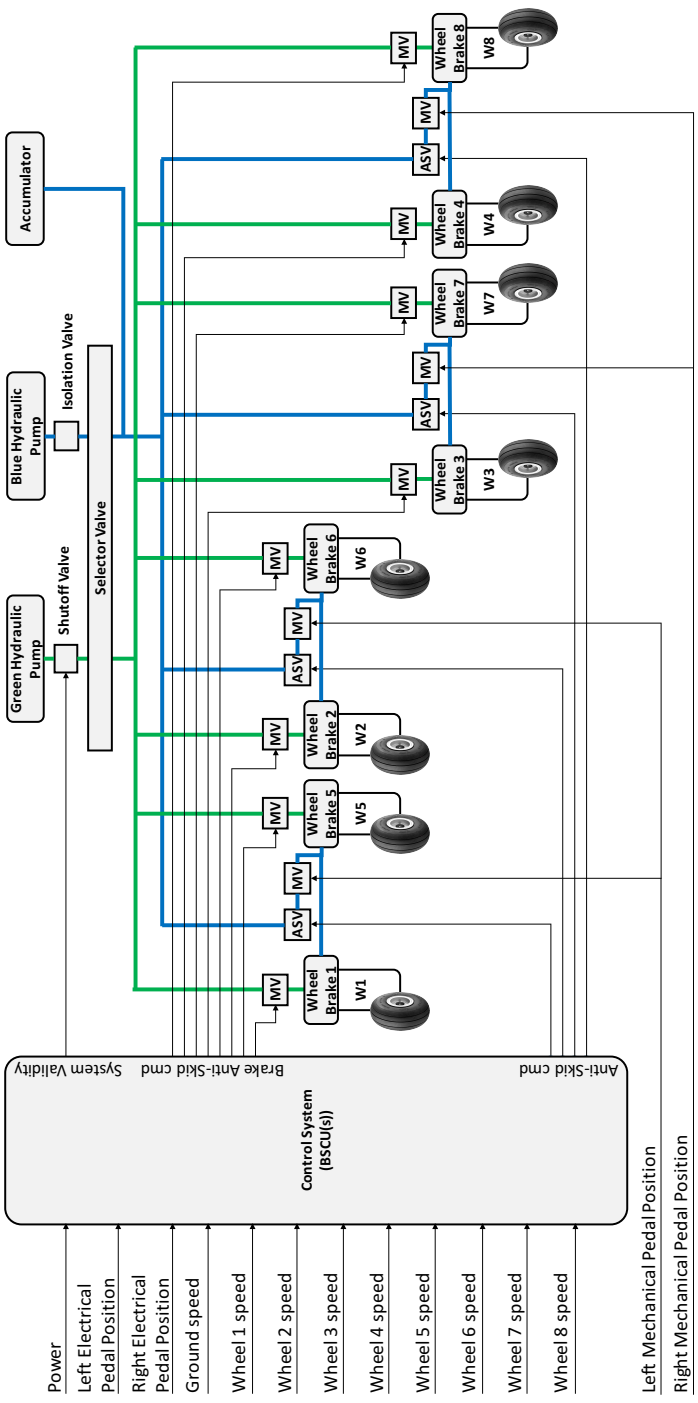

Figure 8: High Level Wheel Brake System

5. RELATED WORK

A model-based approach for safety analysis was proposed by Joshi et. al in [32–34]. 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) bidirectional failure 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) [27] and HiP-HOPS for EAST-ADL [20] are *FLM*-based *ESM* approaches. As previously discussed, given many possible faults, these propagation relationships require substantial user effort and become more complex. In addition, it becomes the analyst's responsibility to determine whether faults can propagate; missing propagations lead to unsound analyses. In the Safety Annex, most propagations occur through system behaviors defined by the nominal contracts with no additional user effort. As described in Section 3.4, the initial propagations of certain faults are specified by the user, in particular the faults that originate in hardware and propagate to the components that rely on the correct service of the hardware.

Closely related to our work is the model-based safety assessment toolset called COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [12]. 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 [13, 17]. 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 [21, 39], MRMC (Markov Reward Model Checker) [35, 38], and RAT (Requirements Analysis Tool) [41]. The safety analysis tool xSAP [9] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [10]. 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 [31] 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 failure propagations. On the other hand, the tools do not yet appear to scale to industrial-sized problems, as mentioned by the authors [31]: "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) [29] 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 [21], PRISM (Probabilistic Symbolic Model Checker) [36], or the MRMC probabilistic model checker [35].

AltaRica [8,40] 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. For this dialect the Safety assessment, fault tree generation, and functional verification can be performed with the aid of NuSMV model checking [14]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [7]. 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 [9, 14, 18]. 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 [15].

6. FUTURE WORK

As mentioned in Section 4.2, the Byzantine fault analysis is currently manually performed. Part of the future work entails automating this process in order to have easy injection of Byzantine faults in a system model.

Considering that fault trees are commonly used in all major fields of safety engineering, and due to the importance of MCSs in safety engineering, it is important for the Safety Annex to automatically provide such artifacts. A difficulty in automatically generating such artifacts is finding the MCSs in large models [11, 18, 42]. At this time, the monolithic method of analysis that is used in order to generate MCSs tends to generate fault trees that are quite shallow but very wide. When the model is large enough, it becomes very difficult to generate all MCSs [37]. Compositional probabilistic analysis is a topic that needs further exploration in this research before fault trees can be generated from compositional safety analysis approaches.

When MCSs are generated in an efficient way, safety analysis artifacts such as FTs and FMEA tables can also be generated. Throughout this process, it is important to find out what is the desired structure of fault trees from the perspective of safety engineers for certification purposes. The goal is to make this tool usable for safety analysts and pertinant to the safety assessment process.

7. CONCLUSION

An extension to the AADL language has been developed 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 and the repository [45, 46].

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

8. REFERENCES

[1] RTCA/DO-178C Software Considerations in Airborne Systems and Equipment Certification, 2011.

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

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

[4] A. Aviyienis, J. Laprie, B. Randall, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[5] 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.

[6] C. Bernardeschi, A. Fantechi, S. Gnesi, and G. Mongardi. Proving safety properties for embedded control systems. In *Dependable Computing - EDCC-2, Second European Dependable Computing Conference, Taormina, Italy, October 2-4, 1996, Proceedings*, pages 321–332, 1996.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic fault tree analysis for reactive systems. In *ATVA*, pages 162–176, 2007.

[19] M. Bozzano and A. Villafiorita. *Design and Safety Assessment of Critical Systems*. Auerbach Publications, Boston, MA, USA, 1st edition, 2010.

[20] 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.

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

[22] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, and P. Traverso. Model checking safety critical software with SPIN: an application to a railway interlocking system. In *Computer Safety, Reliability and Security, 17th International Conference, SAFECOMP'98, Heidelberg, Germany, October 5-7, 1998, Proceedings*, pages 284–295, 1998.

[23] 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.

[24] K. Driscoll, H. Sivencrona, and P. Zumsteg. Byzantine Fault Tolerance, from Theory to Reality. In *SAFECOMP*, volume 2788 of *LNCS*, pages 235–248, 2003.

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

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

[27] 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.

[28] A. Gacek, J. Backes, M. Whalen, and E. Wagner, L. andGhassabani. The JKind Model Checker. *ArXiv e-prints*, Dec. 2017.

[29] M. Gudemann and F. Ortmeier. A framework for qualitative and quantitative formal model-based safety analysis. In *HASE 2010*, pages 132–141, 2010.

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

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] Mattarei. Scalable safety and reliability analysis via symbolic model checking: theory and applications, Feb. 2016.

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

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

[40] 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.

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

[42] 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.

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

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

[45] D. Stewart, J. Liu, M. Whalen, D. Cofer, and M. Peterson. Safety annex for aadl repository. https://github.com/loonwerks/AMASE, 2017.

[46] 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.

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