The Safety Annex for the Architecture Analysis and Design Language

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

[1] University of Minnesota
Department of Computer Science and Engineering
dkstewar, whalen, heimdahl@cs.umn.edu
[2] Collins Aerospace
Trusted Systems - Engineering and Technology
Jing.Liu, Darren.Cofer@collins.com
[3] Collins Aerospace
Flight Controls Safety Engineering - Avionics
Michael.Peterson@collins.com

Abstract. Model-based development techniques are increasingly being used in the development of critical systems software. Leveraging the artifacts from model based development in the safety analysis process would be highly desirable to provide accurate analysis and enable cost savings. In particular, architectural and behavioral models provide rich information about the system's operation. In this paper we describe an extension to the Architecture Analysis and Design Language (AADL) developed to allow a rich modeling of a system under failure conditions. This *Safety Annex* allows the independent modeling of component failure modes and allows safety engineers to weave various types of faults into the nominal system model. The accompanying tool support allows investigation of the propagation of errors from their source to their effect on top level safety properties without the need to add separate propagation specifications; it also supports describing dependent faults that are not captured through the behavioral models, e.g., failures correlated due to the physical structure of the system. We describe the Safety Annex, illustrate its use with a representative example, and discuss and demonstrate the tool support enabling an analyst to investigate the system behavior under failure conditions.

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

1 Introduction

System safety analysis is crucial in the development life cycle of critical systems to ensure adequate safety as well as demonstrate compliance with applicable standards. A prerequisite for any safety analysis is a thorough understanding of the system architecture and the behavior of its components; safety engineers use this understanding to explore the system behavior to ensure safe operation, assess the effect of failures on the overall safety objectives, and construct the accompanying safety analysis artifacts. Developing adequate understanding, especially for software components, is a difficult and

time consuming endeavor. Given the increase in model-based development in critical systems [8, 23, 25, 29, 32], leveraging the resultant models in the safety analysis process holds great promise in terms of analysis accuracy as well as efficiency.

In this paper we describe the *Safety Annex* for the system engineering language AADL (Architecture Analysis and Design Language), a SAE Standard modeling language for Model-Based Systems Engineering (MBSE) [2]. The Safety Annex allows an analyst to model the failure modes of components and then "weave" these failure modes together with the original models developed as part of MBSE. The safety analyst can then leverage the merged behavioral models to propagate errors through the system to investigate their effect on the safety requirements. Determining how errors propagate through software components is currently a costly and time-consuming element of the safety analysis process. The use of behavioral contracts to capture the error propagation characteristics of software component without the need to add separate propagation specifications (*implicit* error propagation) is a significant benefit for safety analysts. In addition, the annex allows modeling of dependent faults that are not captured through the behavioral models (*explicit* error propagation), for example, the effect of a single electrical failure on multiple software components or the effect hardware failure (e.g., an explosion) on multiple behaviorally unrelated components. Furthermore, we will describe the tool support enabling engineers to investigate the correctness of the nominal system behavior (where no failures have occurred) as well as the system's resilience to component failures. We illustrate the work with a substantial example drawn from the civil aviation domain.

Our work can be viewed as a continuation of work conducted by Joshi et al. where they explored model-based safety analysis techniques defined over Simulink/Stateflow [33] models [13, 27–29]. Our current work extends and generalizes this work and provide new modeling and analysis capabilities not previously available. For example, the Safety Annex allows modeling explicit error propagation, supports compositional verification and exploration of the nominal system behavior as well as the system's behavior under failure conditions. Our work is also closely related to the existing safety analysis approaches, in particular, the AADL Error Annex (EMV2) [21], COMPASS [9], and AltaRica [5, 36]. Our approach is significantly different from previous work in that unlike EVM2 we leverage the behavioral modeling for implicit error propagation. We provide compositional analysis capabilities not available in COMPASS. In addition, the Safety Annex is fully integrated in a model-based development process and environment unlike a stand alone language such as AltaRica.

The contributions of the Safety Annex and this paper are:

- close integration of behavioral fault analysis into the *Architecture Analysis and Design Language* AADL, which allows close connection between system and safety analysis and system generation from the model,
- support for *behavioral specification of faults* and their *implicit propagation* through behavioral relationships in the model, in contrast to existing AADL-based annexes (HiP-HOPS, EMV2) and other related toolsets (COMPASS, Cecilia, etc.),
- additional support to capture binding relationships between hardware and software and logical and physical communications, and
- guidance on integration into a traditional safety analysis process.

2 Preliminaries

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

2.1 Traditional Safety Assessment Process

The traditional safety assessment process at the system level is based on ARP4754A [39] and ARP4761 [38]. It starts with the System level Functional Hazard Assessment (SFHA) examining the functions of the system to identify potential functional failures and classifies the potential hazards associated with them.

The next step is the Preliminary System Safety Assessment (PSSA), updated throughout the system development process. A key element of the PSSA is a system level Fault Tree Analysis (FTA). The FTA is a deductive failure analysis to determine the causes of a specific undesired event in a top-down fashion. For an FTA, a safety analyst begins with a failure condition from the SFHA, and systematically examines the system design (e.g., signal flow diagrams provided by system engineers) to determine all credible faults and failure combinations that could cause the undesired event.

The lack of precise models of the system architecture and its failure modes often forces safety analysts to devote significant effort to gathering architectural details about the system behavior from multiple sources. Furthermore, this investigation typically stops at system level, leaving software function details largely unexplored.

Typically equipped with the domain knowledge about the system, but not detailed knowledge of how the software applications are designed, practicing safety engineers find it a time consuming and involved process to acquire the knowledge about the behavior of the software applications hosted in a system and its impact on the overall system behavior. Industry practitioners have come to realize the benefits of using models in the safety assessment process, and a revision of the ARP4761 to include Model Based Safety Analysis (MBSA) is under way.

2.2 Modeling Language for System Design

Figure 1 presents our proposed use of a single unified model to support both system design and safety analysis. It describes both system design and safety-relevant information that are kept distinguishable and yet are able to interact with each other. The shared model maintains a living model that captures the current state of the system design as it moves through the development lifecycle, allowing all participants of the ARP4754A process to be able to communicate and review the system design. Safety analysis artifacts can be generated directly from the model, providing the capability to more accurately analyze complex systems.

We are using the Architectural Analysis and Design Language (AADL) [20] to construct system architecture models. AADL is an SAE International standard that defines a language and provides a unifying framework for describing the system architecture for "performance-critical, embedded, real-time systems" [2]. From its conception, AADL

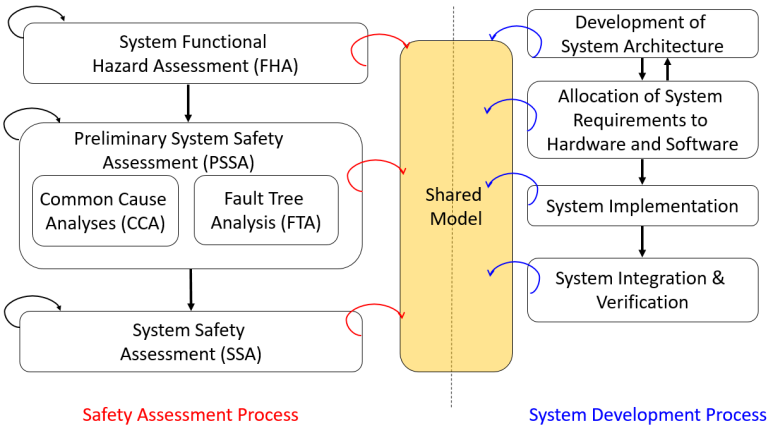


Fig. 1. Use of the Shared System/Safety Model in the ARP4754A Safety Assessment Process

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) [19] 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 [24] 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 [19].

In our prior work [42], we added an initial failure effect modeling capability to the AADL/AGREE language and tool set. We are continuing this work so that our tools and methodology can be used to satisfy system safety objectives of ARP4754A and ARP4761.

2.3 Model-Based Safety Assessment Process Backed by Formal Methods

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

1. System engineers capture the critical information in a shared AADL/AGREE model: high-level hardware and software architecture, nominal behavior at the component level, and safety requirements at the system level (e.g., inhibit throttle movement during critical takeoff phase).
2. System engineers use the backend model checker to check that the safety requirements are satisfied by the nominal design model.
3. Safety engineers use the Safety Annex to augment the nominal model with the component failure modes (e.g., processor failure, input signal corrupted). In addition, safety engineers specify the fault hypothesis for the analysis which corresponds to how many simultaneous faults the system must be able to tolerate.
4. Safety engineers use the backend model checker to analyze if the safety requirements are satisfied by the design in the presence of faults (e.g., if the system is resilient to a single failure). The tool produces counterexamples leading to safety requirement violation in the presence of faults, and also produces fault trees showing smallest set of faults that may lead to the safety requirement being violated.
5. The safety engineers examine the results to assess the validity of the fault combinations and the fault tolerance level of the system design. If a design change is warranted, the model will be updated with the latest design change and the above process is repeated.

There are other tools purpose-built for safety analysis, including AltaRica [36], smartIFlow [26] and xSAP [6]. These notations are separate from the system development model. Other tools extend existing system models, such as HiP-HOPS [16] and the AADL Error Model Annex, Version 2 (EMV2) [21]. EMV2 uses enumeration of faults in each component and explicit propagation of faulty behavior to perform error analysis. The required propagation relationships must be manually added to the system model and can become complex, leading to mistakes in the analysis.

In contrast, the Safety Annex supports model checking and quantitative reasoning by attaching behavioral faults to components and then using the normal behavioral propagation and proof mechanisms built into the AGREE AADL annex. This allows users to reason about the evolution of faults over time, and produce counterexamples demonstrating how component faults lead to failures. Our approach adapts the work of Joshi et. al in [29] to the AADL modeling language. More information on the approach is available in [42], and the tool and relevant documentation can be found at: `https://github.com/loonwerks/AMASE/`.

3 Fault Modeling with the Safety Annex

To demonstrate the fault modeling capabilities of the Safety Annex we will use the Wheel Brake System (WBS) described in AIR6110 [1]. This system is a well-known

example that has been used as a case study for safety analysis, formal verification, and contract based design [8, 12, 13, 27].

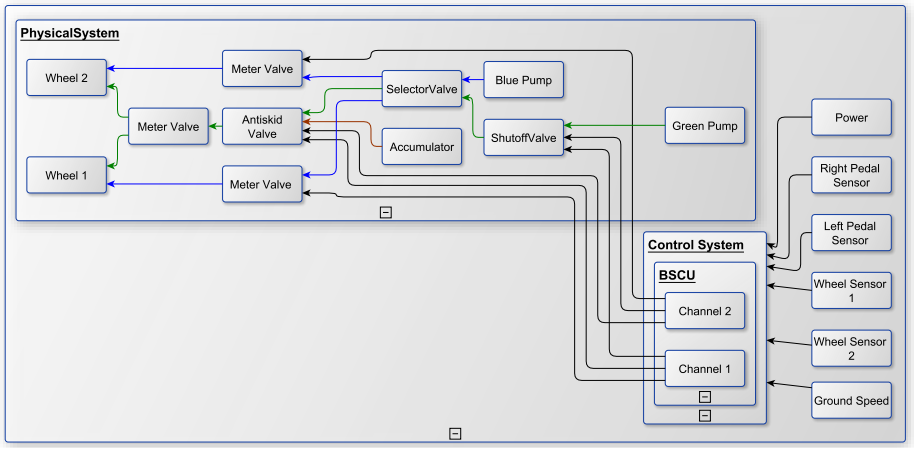


Fig. 2. Wheel Brake System

The WBS is composed of two main parts: the Line Replaceable Unit control system and the electro-mechanical physical system. The control system electronically controls the physical system and contains a redundant channel of the Braking System Control Unit (BSCU) in case a detectable fault occurs in the active channel. It also commands antiskid braking.The physical system consists of the hydraulic circuits running from hydraulic pumps to wheel brakes as well as valves that control the hydraulic fluid flow. This system provides braking force to each of the eight wheels of the aircraft. The wheels are all mechanically braked in pairs (one pair per landing gear). For simplicity, Figure 2 displays only two of the eight wheels.

There are three operating modes in the WBS model:

- In *normal* mode, the system is composed of a *green* hydraulic pump and one meter valve per each of the eight wheels. Each of the meter valves are controlled through electronic commands coming from the active channel of the BSCU. These signals provide braking and antiskid commands for each wheel. The braking command is determined through a sensor on the pedal and the antiskid command is determined by the *Wheel Sensors*.
- In *alternate* mode, the system is composed of a *blue* hydraulic pump, four meter valves, and four antiskid shutoff valves, one for each landing gear. The meter valves are mechanically commanded through the pilot pedal corresponding to each landing gear. If the system detects lack of pressure in the green circuit, the BSCU channel commands the selector valve to switch to the blue circuit.
- In *emergency* mode, the system mode is entered if the *blue* hydraulic pump fails. The accumulator pump has a reserve of pressurized hydraulic fluid and will supply this to the blue circuit in emergency mode.

The WBS architecture model in AADL contains 30 different kinds of components, 169 component instances, and a model depth of 5 hierarchical levels.

The behavioral model is encoded using the AGREE annex and the behavior is based on descriptions found in AIR6110. The top level system properties are given by the requirements and safety objectives in AIR6110. All of the subcomponent contracts support these system safety objectives through the use of assumptions on component input and guarantees on the output. The WBS behavioral model in AGREE annex includes one top-level assumption and 11 top-level system properties, with 113 guarantees allocated to subsystems.

An example system safety property is to ensure that there is no inadvertent braking of any of the wheels. This is based on a failure condition described in AIR6110 is *Inadvertent wheel braking on one wheel during takeoff shall be less than 1E-9 per takeoff*. Inadvertent braking means that braking force is applied at the wheel but the pilot has not pressed the brake pedal. In addition, the inadvertent braking requires that power and hydraulic pressure are both present, the plane is not stopped, and the wheel is rolling (not skidding). The property is stated in AGREE such that inadvertent braking does *not* occur, as shown in Figure 3.

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

Fig. 3. AGREE Contract for Top Level Property: Inadvertent Braking

3.1 Component Fault Modeling

The usage of the terms error, failure, and fault are defined in ARP4754A and are described here for ease of understanding [39]. An *error* is a mistake made in implementation, design, or requirements. A *fault* is the manifestation of an error and a *failure* is an event that occurs when the delivered service of a system deviates from correct behavior. If a fault is activated under the right circumstances, that fault can lead to a failure. The terminology used in EMV2 differs slightly for an error: an error is a corrupted state caused by a fault. The error propagates through a system and can manifest as a failure. In this paper, we use the ARP4754A terminology with the added definition of *error propagation* as used in EMV2. An error is a mistake made in design or code and an error propagation is the corrupted state caused by an active fault.

The Safety Annex is used to add potential faulty behaviors to a component model. Within the AADL component instance model, an annex is added which contain the fault definitions for the given component. The flexibility of the fault definitions allow the user to define numerous types of fault *nodes* by utilizing the AGREE node syntax. A library of common fault nodes has been written and is available at (Where should we put this file for easy accessibility?). Examples of such faults include valves being stuck open or closed, output of a software component being nondeterministic, or power being

cut off. When the fault analysis requires fault definitions that are more complex, these nodes can easily be written and used in the model.

When the fault is activated by its specified triggering conditions, it modifies the output of the component. This faulty behavior may violate the contracts of other components in the system, including assumptions of downstream components. The impact of a fault is computed by the AGREE model checker when the safety analysis is run on the fault model.

The majority of faults that are connected to outputs of components are known as *symmetric*. That is; whatever component receives this faulty output will receive the same faulty output value. Thus this output is seen symmetrically. An alternative fault type is *asymmetric*. This pertains to a component with a fanned output: one which is sent to many receiving components. This fault can present itself differently to the receiving components. For instance, in a boolean setting, one component might see a true value and the rest may see false. This is also possible to model using the keyword *asymmetric*. For more information on fault definitions and modeling possibilities, see the Safety Annex Users Guide [40]. (We should collect the users guide and fault library and any other file of this type and put in a location that is not as hard to look through as the repo.)

As an illustration to fault modeling in the Safety Annex, we look at one of the components important to the inadvertent braking property: the brake pedal. When the mechanical pedal is pressed, a sensor reads this information and passes an electronic signal to the BSCU which then commands hydraulic pressure to the wheels.

Below shows the AADL pedal sensor component with a contract for its nominal behavior. The sensor has only one input, the mechanical pedal position, and one output, the electrical pedal position. A property that governs the behavior of the component is that the mechanical position should always equal the electronic position.

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

  -- Behavioral contracts for subcomponent
  annex agree {**

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

One possible failure for this sensor is inversion of its output value. This fault can be triggered with probability 5.0×10^{-6} as described in AIR6110 (in reality, the component failure probability is collected from hardware specification sheets). The Safety Annex definition for this fault is shown below. Fault behavior is defined through the use of a fault node called *inverted_fail*. When the fault is triggered, the nominal output of the component (*elec_pedal_position*) is replaced with its failure value (*val_out*).

```
annex safety {**
  fault SensorPedalPosition_ErroneousData "Inverted boolean fault" : faults.inverted_fail {
       inputs: val_in <- elec_pedal_position;
       outputs: elec_pedal_position <- val_out;
       probability: 5.0E-6 ;
       duration: permanent;
  }
};
```

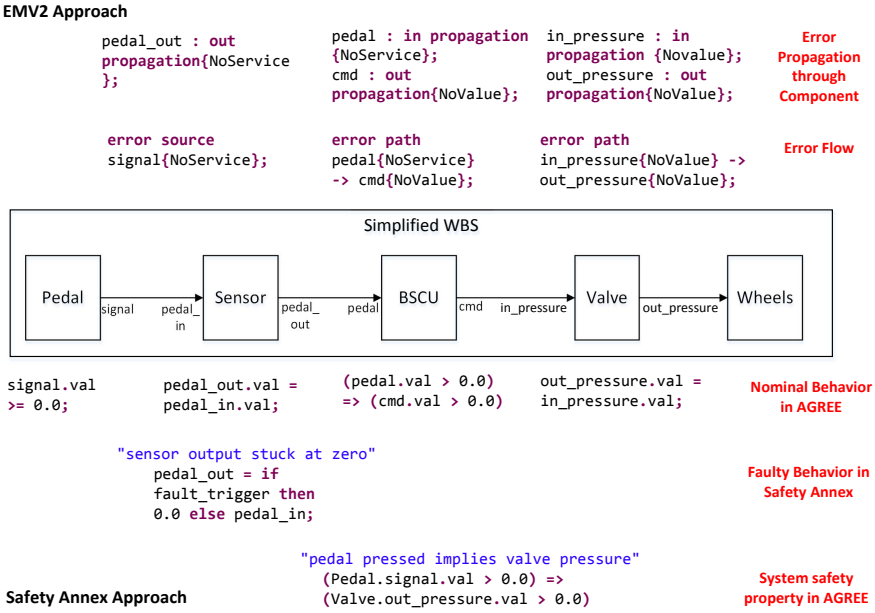


Fig. 4. Differences between Safety Annex and EMV2

The WBS fault model in Safety Annex contains a total of 33 different fault types and 141 fault instances. The large number of fault instances is due to the redundancy in the system design and its replication to control 8 wheels.

3.2 Implicit Error Propagation

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

On the contrary, in the AADL Error Model Annex, Version 2 (EMV2) [21] approach, all errors must be explicitly propagated through each component (by applying fault types on each of the output ports) in order for a component to have an impact on the rest of the system. To illustrate the key differences between implicit error propagation provided in Safety Annex and the explicit error propagation provided in EMV2, we use a simplified behavioral flow from the WBS example using code fragments from EMV2, AGREE, and the Safety Annex.

In this simplified WBS system, the physical signal from the Pedal component is detected by the Sensor and the pedal position value is passed to the Braking System Control Unit (BSCU) components. The BSCU generates a pressure command to the Valve component which applies hydraulic brake pressure to the Wheels.

In the EMV2 approach (top half of Figure 4), the “NoService” fault is explicitly propagated through all of the components. These fault types are essentially tokens that

do not capture any analyzable behavior. At the system level, analysis tools supporting the EMV2 annex can aggregate the propagation information from different components to compose an overall fault flow diagram or fault tree.

When a fault is triggered in the Safety Annex (bottom half of Figure 4), the output behavior of the Sensor component is modified. In this case the result is a "stuck at zero" error. The behavior of the BSCU receives a zero input and proceeds as if the pedal has not been pressed. This will cause the top level system contract to fail: *pedal pressed implies brake pressure output is positive*.

3.3 Explicit Error Propagation

Failures in hardware (HW) components can trigger behavioral faults in the system components that depend on them. For example, a CPU Failure may trigger faulty behavior in the threads bound to that CPU. In addition, a failure in one HW component may trigger failure in other HW components located nearby, such as overheating, fire, or explosion in the containment location. The Safety Annex provides the capability to explicitly model the impact of hardware failures on other faults, behavioral or non behavioral. The explicit propagation to non behavioral faults is similar to that provided in EMV2.

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

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

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

In the WBS example, assume that both the green and blue hydraulic pumps are located in the same compartment in the aircraft and an explosion in this compartment rendered both pumps inoperable. The HW fault definition can be modeled first in the green hydraulic pump component as shown in Figure 3.3. The activation of this fault triggers the activation of related faults as seen in the *propagate_to* statement shown below. Notice that these pumps need not be connected through a data port in order to specify this propagation.

```
annex safety{**

   analyze : probability 1.0E-7
   propagate_from:
     {Pump_HW_Fault@phys_sys.green_hyd_pump} to {HydPump_FailedOff@phys_sys.blue_hyd_pump};

**};
```

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

4 Architecture and Implementation

The Safety Annex is written in Java as a plug-in for the OSATE AADL toolset, which is built on Eclipse. It is not designed as a stand-alone extension of the language, but works with behavioral contracts specified in AGREE AADL annex [19]. The architecture of the Safety Annex is shown in Figure 5.

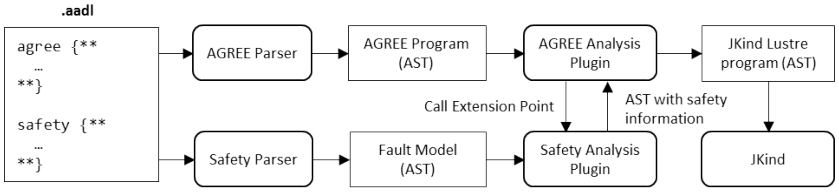


Fig. 5. Safety Annex Plug-in Architecture

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. An example of a portion of an initial AGREE node and its extended contract is shown in Figure 6. The left column of the figure shows the nominal Lustre pump definition is shown with an AGREE contract on the output; and the right column shows the additional local variables for the fault (boxes 1 and 2), the assertion binding the fault value to the nominal value (boxes 3 and 4), and the fault node definition (box 5).

Once augmented with fault information, the AGREE model follows the standard translation path to the model checker JKind [22], 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.

5 Nominal and Fault Analysis of the Model

5.1 Nominal Model Analysis

To illustrate nominal model analysis on the WBS, we first address the types of analysis that AGREE can perform. Using monolithic analysis, the contracts at the lower levels of the architecture are flattened and used in the proof of the top level safety properties of the system. Compositional analysis, on the other hand, will perform the proof layer by layer top down, essentially breaking the larger proof into subsets of smaller problems. For a more comprehensive description of these types of proofs and analyses, see research by Backus, et. al. [3, 18]

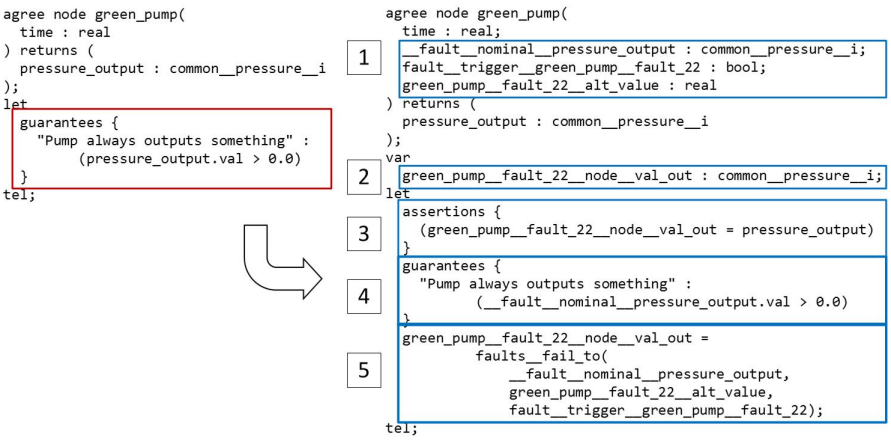


Fig. 6. Nominal AGREE Node and Extension with Faults

The WBS has a total of 29 safety properties at the top level that are supported by subcomponent assumptions and guarantees. These are shown in Table 1. I will adjust this table later to make it prettier and fit in the margins.

S18-WBS-R-0321	Loss of all wheel braking during landing or RTO shall be less than 5E-7 per flight.
S18-WBS-R/L-0322	Asymmetrical loss of wheel braking (Left/Right) shall be less than 5E-7 per flight.
S18-WBS-0323	Never inadvertent braking with all wheels locked.
S18-WBS-0324	Never inadvertent braking with all wheels.
S18-WBS-0325-wheelX	Never inadvertent braking of wheel X.
Braking implies cmd: wheel X	Braking force at wheel X implies braking command was given.
Cmd implies braking: wheel X	Braking command implies that braking force at wheel X.

Table 1. Safety Properties of WBS

As a whole, the model consists of 36 assumptions and 246 supporting guarantees. The nominal AGREE analysis proves the safety properties at the top level in approx 3 minutes for monolithic analysis and 3.5 minutes for compositional analysis.

5.2 Monolithic Fault ModelAnalysis

When monolithic analysis is performed on the nominal system model, the architectural model is flattened in order to perform the analysis. All of the contracts in the lower levels are used for the analysis. Two types of fault analysis can be performed using the monolithic option.

Probabilistic Hypothesis with Monolithic Analysis Given a probabilistic fault hypothesis, this corresponds to performing a analysis on which combinations of faults

have a probability less than the threshold and then inserting assertions into the Lustre code accordingly. If the probability of such combination of faults is in fact less than the designated top level threshold, these faults may be activated and the behavioral effects can be seen through a counterexample.

To perform this analysis, it is assumed that the non-hardware faults occur independently and possible combinations of faults are computed and passed to the Lustre model to be checked by the model checker. As seen in Algorithm 1, the computation first removes all faults from consideration that are too unlikely given the probability threshold. The remaining faults are arranged in a priority queue $\mathcal{Q}$ from high to low. Assuming independence in the set of faults, we take a fault with highest probability from the queue (step 5) and attempt to combine the remainder of the faults in $\mathcal{R}$ (step 7). If this combination is lower than the threshold (step 8), then we do not take into consideration this set of faults and instead remove the tail of the remaining faults in $\mathcal{R}$.

In this calculation, we assume independence among the faults, but in the Safety Annex it is possible to define dependence between faults using a fault propagation statement. After fault combinations are computed using Algorithm 1, the triggered dependent HW faults are added to the combination as appropriate.

Algorithm 1: Monolithic Probability Analysis

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

Max n Faults Hypothesis with Monolithic Analysis Using a max number of faults for the hypothesis (see Section X reference whatever section describes fault hypotheses), the user can constrain the number of active faults in the model. The model is flattened per the monolithic process and given the constraint on the number of possible active faults, the model checker attempts to prove the top level properties given these constraints. If this cannot be done, the counterexample provided will show which of the n faults are active and which contracts are violated. Because the model is flattened, this shows if n active fault(s) can violate a top level property. The counterexample provides one such example.

5.3 Compositional Fault Model Analysis

In compositional analysis, the analysis proceeds in a top down fashion. To prove the top level properties, the properties in the layer directly beneath the top level are used to perform the proof. The analysis proceeds in this manner.

The compositional analysis currently works with the max fault hypothesis. Users can constrain the maximum number of faults within each layer of the model by specifying the maximum fault hypothesis statement to that layer. If any lower level property failed due to activation of faults, the property verification at the higher level can no longer be trusted because the higher level properties were proved based on the assumption that the direct sublevel contracts are valid.

The compositional analysis is helpful to see weaknesses in a given layer of the system. In future work, we plan to reflect lower layer property violations in the verification results of higher layers in the architecture and enable the display or constraint active faults system wide instead of layer wide.

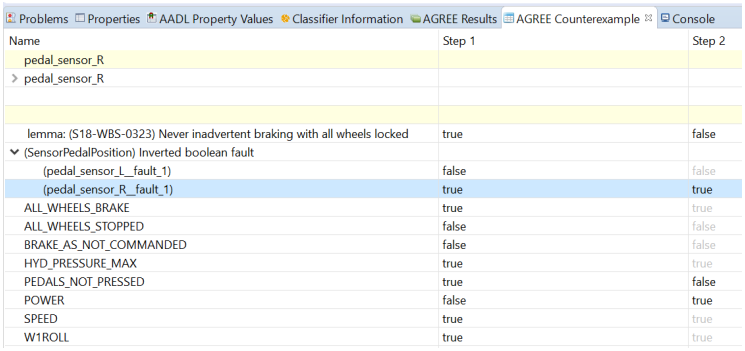
Max n Faults Hypothesis with Compositional Analysis Using a max number of faults for the hypothesis, the user can constrain the number of active faults in *each layer* of the model. In a top down approach, the faults are injected into the Lustre model and JKind attempts to find a satisfiable assignment to all variables *in the contracts of that layer* given the constraint on faults. If this cannot be done, the counterexample provided will show which of the n faults are active in a given layer and which contracts are violated. This is useful to see which subsystems or subcomponents are not resiliant to a certain number of faults and where unexpected behavior in the model can occur. In addition, the combination of all sets of n active faults which cause the violation of a safety property can be collected from the model checker. This allows the user to see all such fault combinations thorughout the model.

Probabilistic Hypothesis with Compositional Analysis Probabilistic compositional analysis collects all combinations of faults in the model that cause violation of safety properties and eliminates the sets in which the combined fault probabilities exceed the threshold given in the probability hypothesis. Thus if a set of independently occurring faults can occur with combined probability greater than or equal to the threshold, and the top level properties are violated, these faults are displayed to the user.

To illustrate how the Safety Annex can be used to determine single points of failure, flaws in system design, or how faults can affect the behavior of the system, we focus on a single top level requirement of the WBS: S18-WBS-0323 (Never indadvertent braking with all wheels locked). This safety property description can be found detaiiled in Section 3. Upon running max n compositional fault analysis with $n = 1$, this particular fault was shown to be a single point of failure for this safety property. A counterexample is shown in Figure 7 showing the active fault on the pedal sensor.

In order to mitigate this problem, various strategies are possible.

- Monitor system for the sensor: A monitor subcomponent can be modelled in which it accesses the mechanical pedal as well as the signal from the sensor. If the monitor



Name	Step 1	Step 2
pedal_sensor_R		
pedal_sensor_R		
lemma: (S18-WBS-0323) Never inadvertent braking with all wheels locked	true	false
(SensorPedalPosition) Inverted boolean fault		
(pedal_sensor_L_fault_1)	false	false
(pedal_sensor_R_fault_1)	true	true
ALL_WHEELS_BRAKE	true	true
ALL_WHEELS_STOPPED	false	false
BRAKE_AS_NOT_COMMANDED	false	false
HYD_PRESSURE_MAX	true	true
PEDALS_NOT_PRESSED	true	false
POWER	false	true
SPEED	true	true
W1ROLL	true	true

Fig. 7. AGREE counterexample for inadvertent braking safety property

finds discrepencies between these values, it can report an invalid monitor to the top level of the system. In terms of the modelling, this would require a change to the contracts which use the sensor value. This validity would be taken into account through the means of $valid \wedge pedal\_sensor\_value$. In the real system however, this mitigation would need to be taken into account. Whether this is a flag to the pilot who can then override the electrical system and switch to a different mode or perform some other action to mitigate the failed sensor must be discussed and implemented.

– Redundancy in the sensor: A sensor subsystem can be modelled which contains 3 or more sensors. The overall output from the sensor system may utilize a voting scheme to determine validity of sensor reading. There are multiple voting schemes that are possible, one of which is a majority voting. If three sensors are present, this mitigates the single point of failure problem (e.g. one sensor fails, the other two take majority vote and the correct value is passed). In this case, the whole of the system sees the sensor system as a single sensor. No contracts need to be updated or changed and the architectural connections remain in tact. The only new contracts of the system are located in the sensor system and model the behavior of redundancy and voting.

Depending on the goals of the system, the architecture currently modelled, and the mitigation strategies that are desired, solutions will vary. But due to the information gathered in the fault analysis phase of design, the information is readily available that a mitigation strategy is required.

In the case of the pedal sensor in the WBS, the latter of the two strategies outlined above was implemented. A sensor system was added to the model which held three pedal sensors. The output of this subsystem was constrained using a majority voting scheme. Upon subsequent runs of the analysis (regardless which type of run was used), there was resiliance in the system regarding the failure of a single pedal sensor. Figure 8 outlines these architectural changes that were made in the model.

As can be seen through this single example, a system as large as the WBS would benefit from many iterations of this process. Furthermore, if the model is changed even slightly on the system development side, it would automatically be seen from the safety analysis perspective and any negative outcomes would be shown upon subsequent anal-

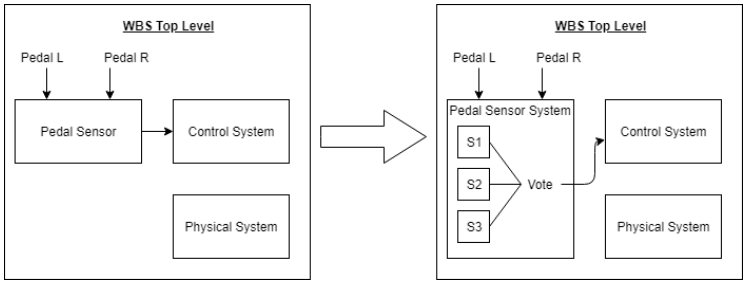


Fig. 8. Changes in architectural model for fault mitigation

ysis runs. This effectively eliminates any miscommunications between the development and analysis teams and creates a new safeguard regarding model changes.

For more information on types of fault models that can be created as well as details on analysis results, see the users guide located in the GitHub repository [40]. This repository also contains all models used in this case study.

6 Related Work

A model-based approach for safety analysis was proposed by Joshi et. al in [27–29]. In this approach, a safety analysis system model (SASM) is the central artifact in the safety analysis process, and traditional safety analysis artifacts, such as fault trees, are automatically generated by tools that analyze the SASM.

The contents and structure of the SASM differ significantly across different conceptions of MBSA. We can draw distinctions between approaches along several different axes. The first is whether they propagate faults explicitly through user-defined propagations, which we call *failure logic modeling* (FLM) or through existing behavioral modeling, which we call *failure effect modeling* (FEM). The next is whether models and notations are *purpose-built* for safety analysis vs. those that extend *existing system models* (ESM).

For FEM approaches, there are several additional dimensions. One dimension involves whether *causal* or *non-causal* models are allowed. Non-causal models allow simultaneous (in time) bi-directional error propagations, which allow more natural expression of some failure types (e.g. reverse flow within segments of a pipe), but are more difficult to analyze. A final dimension involves whether analysis is *compositional* across layers of hierarchically-composed systems or *monolithic*. Our approach is an extension of AADL (ESM), causal, compositional, mixed FLM/FEM approach.

Tools such as the AADL Error Model Annex, Version 2 (EMV2) [21] and HiP-HOPS for EAST-ADL [16] are *FLM*-based *ESM* approaches. As previously discussed, given many possible faults, these propagation relationships require substantial user effort and become more complex. In addition, it becomes the analyst's responsibility to determine whether faults can propagate; missing propagations lead to unsound analyses. In our Safety Annex, propagations occur through system behaviors (defined by the nominal contracts) with no additional user effort.

Closely related to our work is the model-based safety assessment toolset called COMPASS (Correctness, Modeling project and Performance of Aerospace Sys-

tems) [9]. 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 [10, 14]. 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 [17, 35], MRMC (Markov Reward Model Checker) [30, 34], 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]. 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 [26] is a *FEM*-based, *purpose-built*, *monolithic non-causal* safety analysis tool that describes components and their interactions using finite state machines and events. Verification is done through an explicit state model checker which returns sets of counterexamples for safety requirements in the presence of failures. SmartIFlow allows *non-causal* models containing simultaneous (in time) bi-directional error propagations. On the other hand, the tools do not yet appear to scale to industrial-sized problems, as mentioned by the authors [26]: "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) [23] 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 [17], PRISM (Probabilistic Symbolic Model Checker) [31], or the MRMC probabilistic model checker [30].

AltaRica [5, 36] 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 [11]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [4]. 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 [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].

7 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. The support to the implicit error propagation enables safety engineers to inject failures/faults at component level and assess the effect of behavioral propagation at the system level. It also supports explicit error propagation that allows safety engineers to describe dependent faults that are not easily captured using implicit error propagation. Next steps will include adding full support for compositional fault analysis, automatic generation of fault trees, and extensions to automate injection of Byzantine faults. For more details on the tool, models, and approach, see the technical report [41]. To access the tool plugin, users manual, or models, see the repository [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. 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.
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*, 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 *Computer Safety, Reliability, and Security*. Springer Berlin Heidelberg, 2009.
10. M. Bozzano, A. Cimatti, J.-P. Katoen, V. Yen Nguyen, T. Noll, and M. Roveri. Model-based codesign of critical embedded systems. 507, 2009.
11. M. Bozzano, A. Cimatti, O. Lisagor, C. Mattarei, S. Mover, M. Roveri, and S. Tonetta. Symbolic Model Checking and Safety Assessment of Altarica Models. In *Science of Computer Programming*, volume 98, 2011.
12. M. Bozzano, A. Cimatti, C. Mattarei, and S. Tonetta. Formal safety assessment via contract-based design. In *Automated Technology for Verification and Analysis*, 2014.

13. M. Bozzano, A. Cimatti, A. F. Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta. Formal Design and Safety Analysis of AIR6110 Wheel Brake System. In *CAV 2015, Proceedings, Part I*, pages 518–535, 2015.
14. M. Bozzano, A. Cimatti, M. Roveri, J. P. Katoen, V. Y. Nguyen, and T. Noll. Codesign of dependable systems: A component-based modeling language. In *2009 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design*, 2009.
15. M. Bozzano, A. Cimatti, and F. Tapparo. Symbolic fault tree analysis for reactive systems. In *ATVA*, 2007.
16. D. Chen, N. Mahmud, M. Walker, L. Feng, H. Lönn, and Y. Papadopoulos. Systems Modeling with EAST-ADL for Fault Tree Analysis through HiP-HOPS*. *IFAC Proceedings Volumes*, 46(22):91 – 96, 2013.
17. A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2000.
18. D. Cofer, A. Gacek, S. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional verification of architectural models. In *NASA Formal Methods Symposium*, pages 126–140. Springer, 2012.
19. 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.
20. P. Feiler and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.
21. 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.
22. A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The JKind Model Checker. *CAV 2018*, 10982, 2018.
23. M. Gudemann and F. Ortmeier. A framework for qualitative and quantitative formal model-based safety analysis. In *HASE 2010*, 2010.
24. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. In *IEEE*, volume 79(9), pages 1305–1320, 1991.
25. P. Hönig, R. Lunde, and F. Holzapfel. Model Based Safety Analysis with smartIflow. *Information*, 8(1), 2017.
26. P. Hönig, R. Lunde, and F. Holzapfel. Model Based Safety Analysis with smartIflow. *Information*, 8(1), 2017.
27. 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.
28. 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.
29. 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.
30. 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.
31. 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.
32. O. Lisagor, T. Kelly, and R. Niu. Model-based safety assessment: Review of the discipline and its challenges. In *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, 2011.

33. MathWorks. The MathWorks Inc. Simulink Product Web Site. http://www.mathworks.com/products/simulink, 2004.
34. MRMC: Markov Rewards Model Checker. http://wwwhome.cs.utwente.nl/ zapreevis/mrmc/.
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), 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 aadl repository. `https://github.com/loonwerks/AMASE`, 2018.
41. 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.
42. D. Stewart, M. Whalen, D. Cofer, and M. P. Heimdahl. Architectural Modeling and Analysis for Safety Engineering. In *IMBSA 2017*, pages 97–111, 2017.