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 [**?**, **?**, **?**, **?**, **?**], 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) [**?**]. 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 [**?**] models [**?**, **?**, **?**, **?**]. 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) [**?**], COMPASS [**?**], and AltaRica [**?**, **?**]. 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 Safety Assessment Process

ARP4754A, the Guidelines for Development of Civil Aircraft and Systems [**?**], provides guidance on applying development assurance at each hierarchical level throughout the development life cycle of highly-integrated/complex aircraft systems. It has been recognized by the Federal Aviation Administration (FAA) as an acceptable method to establish the assurance process. The safety assessment process is a starting point at each hierarchical level of the development life cycle and is tightly coupled with the system development and verification processes. It is used to show compliance with certification requirements and for meeting a company's internal safety standards.

ARP4761, the Guidelines and Methods for Conducting Safety Assessment Process on Civil Airborne Systems and Equipment [**?**], identifies a systematic means to show compliance. Among the industry accepted safety assessment processes are Preliminary System Safety Assessment (PSSA) and System Safety Assessment (SSA). PSSA evaluates the system design and defines safety requirements. SSA evaluates the implemented system to show that safety requirements defined in the PSSA are in fact satisfied.

A prerequisite of performing the safety assessment is understanding how the system is intended to work, primarily focusing on the integrity of the outputs and the availability of the system. The safety engineers then use the acquired understanding to conduct safety analysis, construct safety analysis artifacts, and compare the results with established safety objectives and requirements. 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. Figure **??** 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.

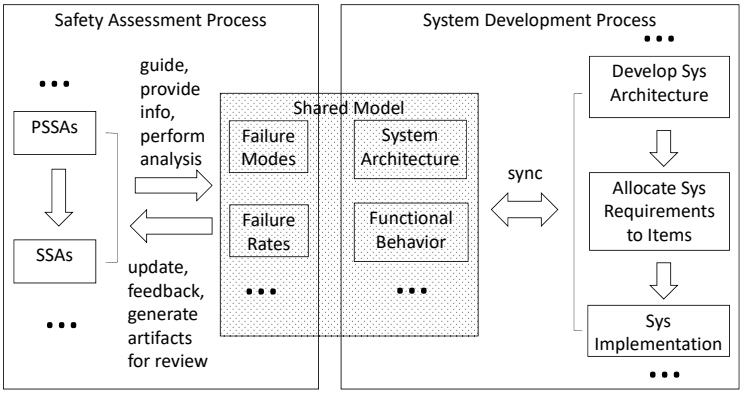

Fig. 1. Use of the Shared System/Safety Model in the ARP4754A Safety Assessment Process

2.2 Modeling Language for System Design

We are using the Architectural Analysis and Design Language (AADL) [**?**] 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" [**?**]. From its conception, AADL has been designed for the design and construction of avionics systems. Rather than being merely descriptive, AADL models can be made specific enough to support system-level code generation. Thus, results from analyses conducted, including the new safety analysis proposed here, correspond to the system that will be built from the model.

An AADL model describes a system in terms of a hierarchy of components and their interconnections, where each component can either represent a logical entity (e.g., application software functions, data) or a physical entity (e.g., buses, processors). An AADL model can be extended with language annexes to provide a richer set of modeling elements for various system design and analysis needs (e.g., performance-related characteristics, configuration settings, dynamic behaviors). The language definition is sufficiently rigorous to support formal analysis tools that allow for early phase error/-fault detection.

The Assume Guarantee Reasoning Environment (AGREE) [**?**] 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 [**?**] and then queries a user-selected model checker to conduct the back-end analysis. The analysis can be performed compositionally following the architecture hierarchy such that analysis at a higher level is based on the components at the next lower level. When compared to monolithic analysis (i.e., analysis of the flattened model composed of all

components), the compositional approach allows the analysis to scale to much larger systems [**?**].

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

3 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 [**?**]. This system is a well-known example that has been used as a case study for safety analysis, formal verification, and contract based design [**?**, **?**, **?**, **?**].

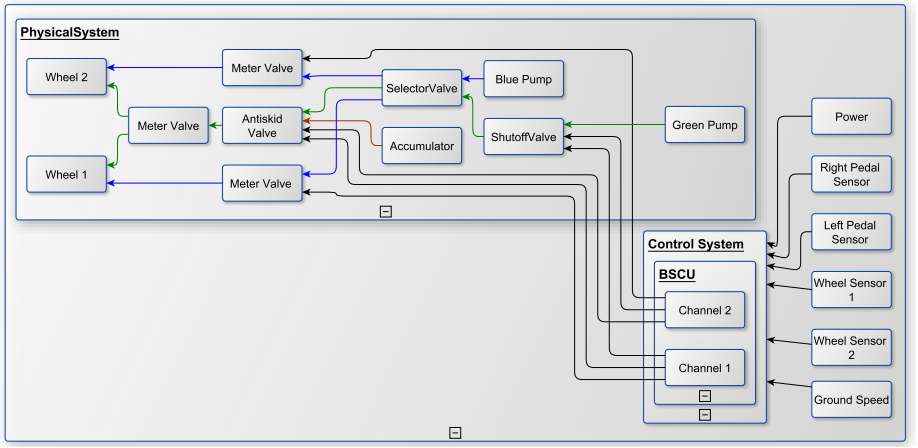

Fig. 2. Wheel Brake System

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 **??** 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 below.

```
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)));
```

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 [**?**]. 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. 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. Examples of such faults include valves being stuck open or closed, output of a software component being nondeterministic, or power being cut off.

One of the components important to the inadvertent braking property is 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 behaviors may be defined by the user or by using a library of fault nodes, in this case, *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;
  }
};
```

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.

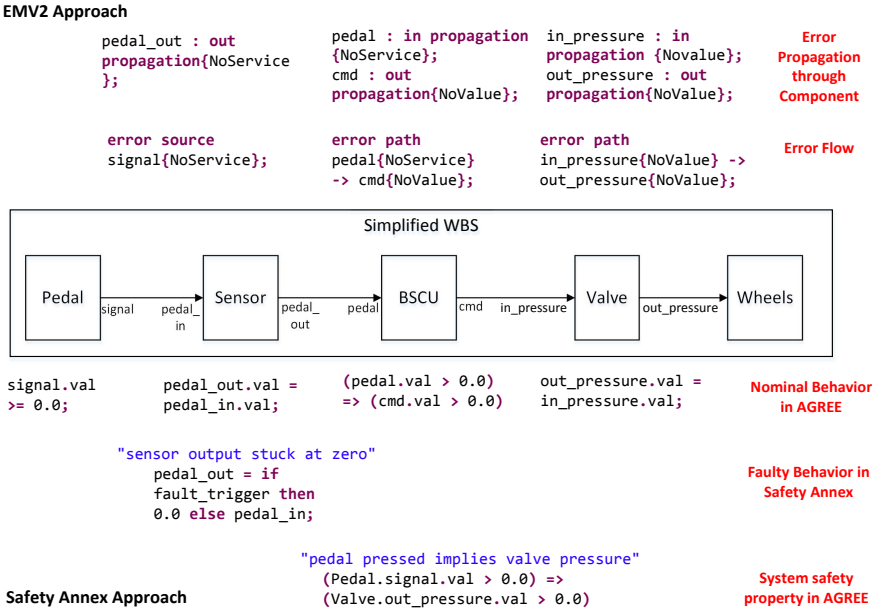

Fig. 3. Differences between Safety Annex and EMV2

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) [**?**] 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 **??**), 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 **??**), 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 **??**. 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 [**?**]. The architecture of the Safety Annex is shown in Figure **??**.

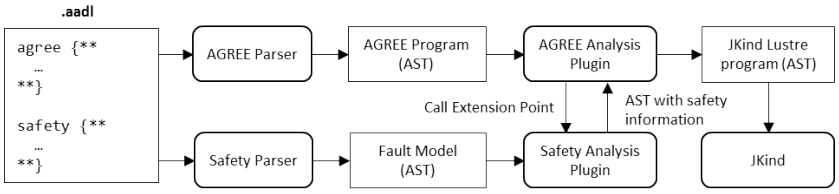

Fig. 4. Safety Annex Plug-in Architecture

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 **??**. 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 [**?**], 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.

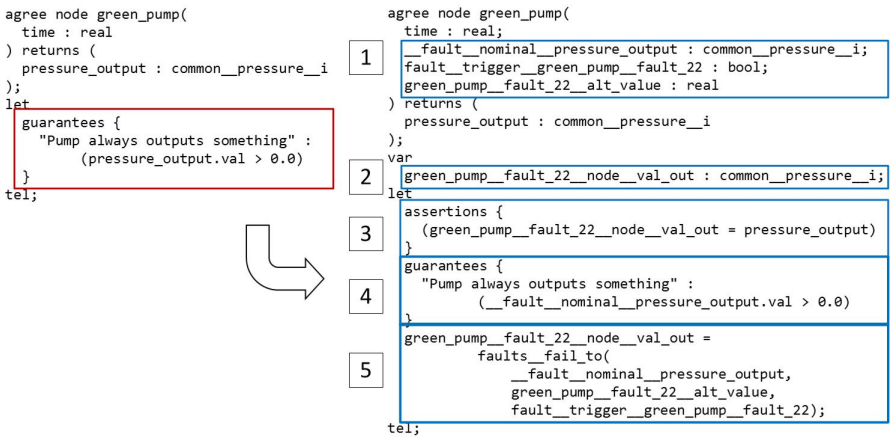

Fig. 5. Nominal AGREE Node and Extension with Faults

5 Analysis of the Fault Model

When the Safety Annex is enabled, users can invoke either the monolithic analysis or compositional analysis in AGREE to check if the top level safety properties of the system hold in the presence of faults under the fault hypothesis given for the system. If an active fault causes the violation of a contract, a counterexample is provided by the model checker. The counterexample can be used to further analyze the system design and make necessary updates to the shared model between safety assessment and system development processes. This iterations continues until the system safety property is satisfied with the desired fault tolerance and failure probability achieved.

5.1 Fault Hypothesis

As the number of component faults increases, the different fault combinations can grow exponentially, making model checking infeasible. Therefore, a fault hypothesis needs to be specified for the system under verification to limit the simultaneous fault activations that are considered by the model checker.

A Safety Annex annotation in the system implementation of the AADL model determines the fault hypothesis. There are two types of fault hypothesis:

The *max fault hypothesis* specifies a maximum number of faults that can be active at any point in execution. This is analogous to restricting the cutsets to a specified maximum number of terms in the fault tree analysis in traditional safety analysis. In implementation (i.e., the translated Lustre model feeding into the model checker), we assert that the sum of the true *fault_trigger* variables is below some integer threshold. Each layer of the model needs to have a max fault hypothesis statement specified in order to consider fault activation in that layer in the analysis.

The *probabilistic fault hypothesis* specifies that only faults whose probability of simultaneous occurrence is above some probability threshold should be considered (see Figure **??**). This is analogous to restricting the cutsets to only those whose probability is above some set value. In implementation, we determine all combinations of faults whose probabilities are above the specified probability threshold and describe this as a proposition over *fault_trigger* variables. Each subcomponent fault needs to specify a probability of occurrence in order to be considered in the analysis.

With the introduction of dependent faults, active faults are divided into two categories: independently active (activated by its own triggering event) and dependently active (activated when the faults they depend on become active). The top level fault hypothesis applies to independently active faults. Faulty behaviors augment nominal behaviors whenever their corresponding faults are active (either independently active or dependently active).

5.2 Monolithic Analysis

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.

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);
```

5.3 Compositional 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.

5.4 Analysis of the WBS Example

The fault analysis on the top level WBS system was performed on the 11 top-level properties applying the max fault hypothesis and probabilistic fault hypothesis separately. It requires between 2 and 4 minutes to run either compositional analysis with max fault hypothesis or monolithic analysis with probabilistic fault hypothesis on the model. The analysis is computationally inexpensive, allowing quick iterations between systems and safety engineers.

We first applied compositional analysis with max one fault hypothesis at all layers of the model. Most properties were verified except for the *Inadvertent braking at the wheel* properties. The failure of the verification for those properties shows that they are not resilient to a single fault. We then applied monolithic analysis with probabilistic fault hypothesis of 10^{-9} fault threshold. The *Inadvertent braking at the wheel* properties also failed. Results from the first round of checks indicate that the WBS design is not fault tolerant to the inadvertent braking properties and is not meeting the Probability of Failure goal of 10^{-9} on those properties.

The counterexample returned by the tools allowed us to straightforwardly diagnose the fault conditions that lead to property failure: in this model, there is a single pedal position sensor for the brake pedal. If this sensor fails, it can command braking without a pilot request. The counterexample can be used to further analyze the system design and explore a solution to the problem. There are several ways to proceed (here we note that the architecture of the pedal assembly is not discussed in AIR6110):

- Decrease the probability of failures of the brake pedal sensor. The failure probabilities are estimated from the failure rates and exposure times of the events. We may adjust the exposure time to match the phase of flight, rather than normalizing it per-flight-hour, if the phase of flight is sufficiently short. We may also find a brake pedal sensor with a lower failure rate. For example, if the failure probability for the sensor component is lower than the 10^{-9} fault threshold, it will not be considered in the possible fault combinations for the analysis. Decreasing the probability of failure of the components could help increase the reliability of the system. However, in this case the system still has a single point of failure.
- Create redundancy in the sensor component and model a voting procedure. In order for braking to occur, both (or all) sensors must agree. This would eliminate a single point of failure and would also cause the model to meet the top level probabilistic threshold. However, by introducing redundancy, it could affect the probability of meeting of other properties in the system that command braking, for example, when braking is commanded, braking pressure is provided at the wheel. Introducing the redundancy can increase the integrity of the system with respect to inadvertent braking, but decrease the availability of the system with respect to braking when needed.

Providing a way to quickly and effectively run analysis on the model for these different modes of failure can be of great benefit to assist system engineers to make design decisions and safety engineers to assess the effect.

6 Related Work

A model-based approach for safety analysis was proposed by Joshi et. al in [**?**, **?**, **?**]. 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) [**?**] and HiP-HOPS for EAST-ADL [**?**] are *FLM*-based *ESM* approaches. As previously discussed, given many possible faults, these propagation relationships require substantial user effort and become more complex. In addition, it becomes the analyst's responsibility to determine whether faults can propagate; missing propagations lead to unsound analyses. In our Safety Annex, propagations occur through system behaviors (defined by the nominal contracts) with no additional user effort.

Closely related to our work is the model-based safety assessment toolset called COMPASS (Correctness, Modeling project and Performance of Aerospace Systems) [**?**]. 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 [**?**,**?**]. 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 [**?**, **?**], MRMC (Markov Reward Model Checker) [**?**, **?**], and RAT (Requirements Analysis Tool) [**?**]. The safety analysis tool xSAP [**?**] can be invoked in order to generate safety analysis artifacts such as fault trees and FMEA tables [**?**]. 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 [**?**] 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 [**?**]: "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) [**?**] 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 [**?**], PRISM (Probabilistic Symbolic Model Checker) [**?**], or the MRMC probabilistic model checker [**?**].

AltaRica [**?**, **?**] 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 [**?**]. Failure states are defined throughout the system and flow variables are updated through the use of assertions [**?**]. 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 [**?**, **?**, **?**]. 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 [**?**].

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 [**?**]. To access the tool plugin, users manual, or models, see the repository [**?**].

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