Technical Report: The Safety Annex for AADL

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, safety analysis, fault analysis, safety engineering

1 Introduction

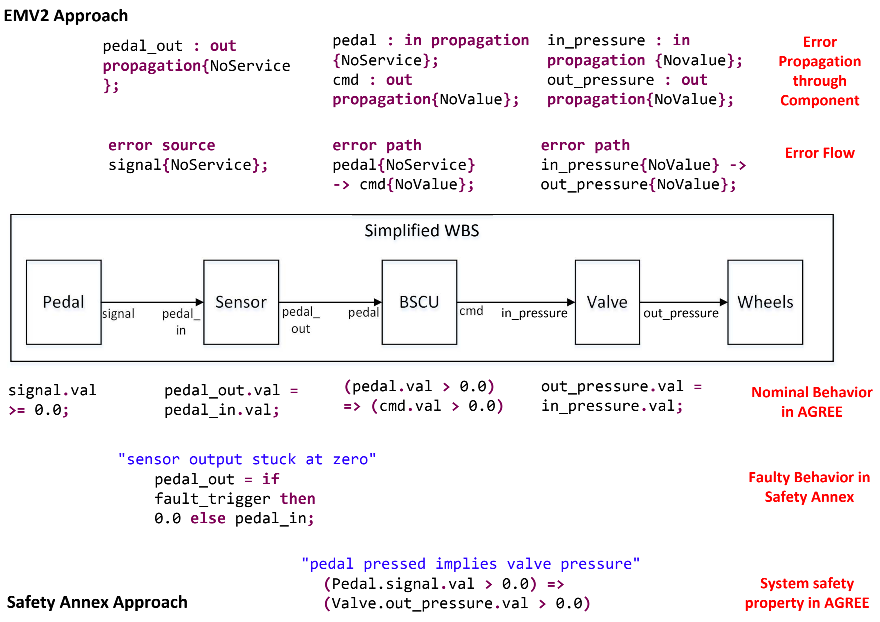
System safety analysis is crucial in the development life cycle of critical systems to ensure adequate safety as well as demonstrate compliance with applicable standards. A prerequisite for any safety analysis is a thorough understanding of the system architecture and the behavior of its components; 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.



```
pedal_out : out
propagation{NoService
};
```

```
pedal : in propagation
{NoService};
cmd : out
propagation{NoValue};
```

```
in_pressure : in
propagation {Novalue};
out_pressure : out
propagation{NoValue};
```

```
error source
signal{NoService};
```

```
error path
pedal{NoService}
-> cmd{NoValue};
```

```
error path
in_pressure{NoValue} ->
out_pressure{NoValue};
```

```
signal.val
>= 0.0;
```

```
pedal_out.val =
pedal_in.val;
```

```
(pedal.val > 0.0)
=> (cmd.val > 0.0)
```

```
out_pressure.val =
in_pressure.val;
```

```
"sensor output stuck at zero"
    pedal_out = if
    fault_trigger then
    0.0 else pedal_in;
```

```
"pedal pressed implies valve pressure"
  (Pedal.signal.val > 0.0) =>
  (Valve.out_pressure.val > 0.0)
```

Fig. 1. Differences between Safety Annex and EMV2

2 Example

The AADL language has previously been extended to provide some fault modeling and analysis capabilities using its Error Model Annex, Version 2 (EMV2) [21]. EMV2 focuses on injection and propagation of discrete faults for generation of fault trees, rather than on analysis of system behavior in the presence of faults. To illustrate some of the key differences between our approach and the EMV2 approach, Figure 1 shows a simplified example based on an aircraft Wheel Brake System (WBS). The WBS model is described in greater detail in [41] and in Section 6. The code fragments in the figure extracted from EMV2, AGREE, and the Safety Annex do not represent the complete code.

In our simplified WBS system, the physical signal from the Pedal component in 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 this example, we use the general term "fault" to denote all component errors, hardware failures, and system faults captured by both approaches.

In the EMV2 approach (top half of Figure 1), all faults 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. In the example, the "NoService" fault is explicitly allowed by the EMV2 declarations to propagate 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 fault flow and propagation information from different components to compose an overall fault flow diagram or fault tree.

In the Safety Annex approach (bottom half of Figure 1), faults are captured as faulty behaviors that augment the system behavioral model in AGREE contracts. When a fault is triggered, the output behavior of the Sensor component is modified, in this case resulting 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*. No explicit fault 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.

3 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.

3.1 Safety Assessment Process

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

The safety assessment process is a starting point at each hierarchical level of the development life cycle, and is tightly coupled with the system development and verification processes. It is used to show compliance with certification requirements, and for meeting a company's internal safety standards. ARP4761, the Guidelines and Methods for Conducting Safety Assessment Process on Civil Airborne Systems and Equipment [38], identifies a systematic means to show compliance. The guidelines presented in ARP4761 include industry accepted safety assessment processes (Functional Hazard Assessment (FHA), Preliminary System Safety Assessment (PSSA), and System Safety Assessment (SSA)), and safety analysis methods to conduct the safety assessment, such as Fault Tree Analysis (FTA), Failure Modes and Effect Analysis (FMEA), and Common Cause Analysis (CCA).

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

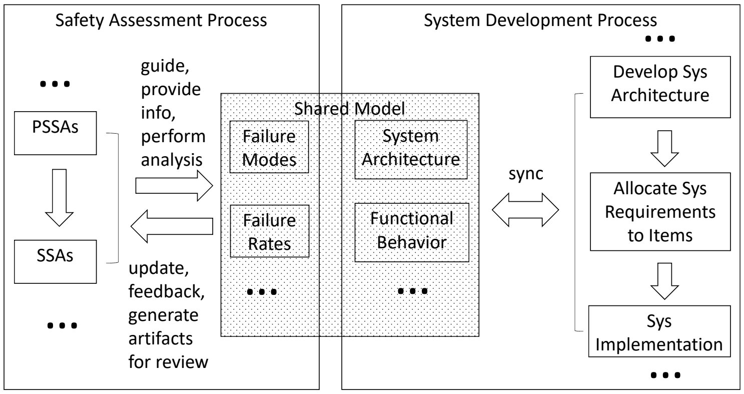

Safety Assessment Process
System Development Process
PSSAs
SSAs
guide, provide info, perform analysis
update, feedback, generate artifacts for review
Shared Model
Failure Modes
Failure Rates
System Architecture
Functional Behavior
sync
Develop Sys Architecture
Allocate Sys Requirements to Items
Sys Implementation

Fig. 2. Use of the Shared System/Safety Model in the ARP4754A Safety Assessment Process

In practice, prior to performing the safety assessment of a system, the safety engineers are often equipped with the domain knowledge about the system, but do not necessarily have detailed knowledge of how the software functions are designed. Acquiring the required knowledge about the behavior and implementation of each software function in a system can be time-consuming.

For example, in a recent project it took one of our safety engineers two days to understand how the software in a Stall Warning System was intended to work. The primary task includes connecting the signal and function flows to relate the input and output signals from end-to-end and understanding the causal effect between them. This is at least as much time as was required to construct the safety analysis artifacts and perform the safety analysis itself. In another instance, it took a safety engineer several months to finalize the PSSA document for a Horizontal Stabilizer Control System, because of two major design revisions requiring multiple rounds of reviews with system, hardware, and software engineers to establish complete understanding of the design details.

Industry practitioners have come to realize the benefits and importance of using models to assist the safety assessment process (either by augmenting the existing system design model, or by building a separate safety model), and a revision of the ARP4761 to include *model based safety analysis* is under way. Capturing failure modes in models and generating safety analysis artifacts directly from models could greatly improve communication and synchronization between system designer and safety engineers, and provide the ability to more accurately analyze complex systems.

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

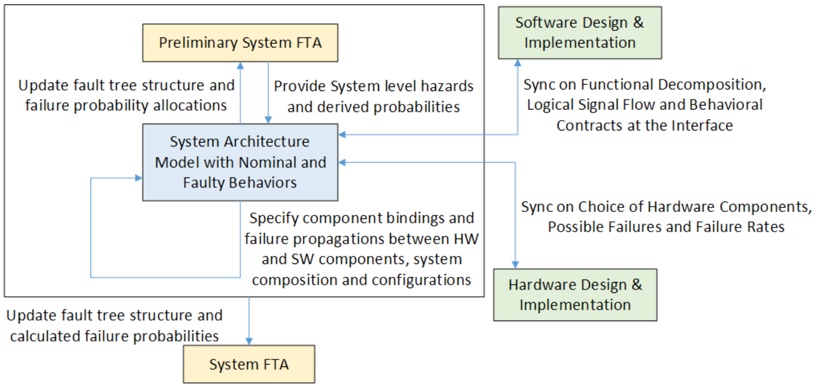

Preliminary System FTA
Update fault tree structure and failure probability allocations
Provide System level hazards and derived probabilities
System Architecture Model with Nominal and Faulty Behaviors
Specify component bindings and failure propagations between HW and SW components, system composition and configurations
Software Design & Implementation
Sync on Functional Decomposition, Logical Signal Flow and Behavioral Contracts at the Interface
Sync on Choice of Hardware Components, Possible Failures and Failure Rates
Hardware Design & Implementation
Update fault tree structure and calculated failure probabilities
System FTA

Fig. 3. Example Interactions between the Shared System/Safety Model and the FTAs

A model that supports both system design and safety analysis must describe both the system design information (e.g., system architecture, functional behavior) and safety-relevant information (e.g., failure modes, failure rates). It must do this in a way that keeps the two types of information distinguishable, yet allows them to interact with each other.

Figure 2 presents our proposed use of a single unified model to support both system design and safety analysis. The shared model can serve as an interface to capture the information from the system design and implementation that is relevant for the safety analysis.

Figure 3 shows how the preliminary FTAs and final system FTAs (artifacts from the PSSA and SSA activities in the Safety Assessment Process) can guide and be updated from the shared model. The shared model is expected to be created and maintained in sync with the software and hardware design and implementation, and guided by the hazard and probability information from the preliminary system FTA. The analysis results from checking the system level properties on the shared model are then used to update the preliminary system FTA. This process continues iteratively until the system safety property is satisfied with the desired fault tolerance and failure probability achieved. The effort needed to update the final system FTA from the preliminary system FTA would be greatly reduced.

3.2 Modeling Language for System Design

We are using the Architectural Analysis and Design Language (AADL) [20] to construct system architecture models. AADL is an SAE International standard [2] that defines a language and provides a unifying framework for describing the system architecture for “performance-critical, embedded, real-time systems” [2]. From its conception, AADL has been designed for the design and construction of avionics systems. Rather than being merely descriptive, AADL models can be made specific enough to support system-level code generation. Thus, results from analyses conducted, including the new safety analysis proposed here, correspond to the system that will be built from the model.

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

The Assume Guarantee Reasoning Environment (AGREE) [18] 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.

In our prior work [41], we added an initial failure effect modeling capability to the AADL/AGREE language and tool set. We are continuing this work so that our tools and methodology can be used to satisfy system safety objectives of ARP4754A and ARP4761.

4 The Safety Annex

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

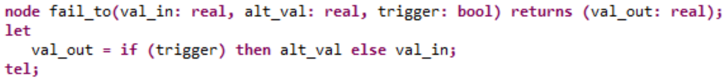
The Safety Annex Users Guide can be found in the GitHub repository along with the tool plugins and examples described in this technical report [40].

4.1 Basic Functionality

An AADL model of the nominal system behavior specifies the hardware and software components of the system and their interconnections. This nominal model is then annotated with assume-guarantee contracts using the AGREE annex [18] for AADL. The nominal model requirements are verified using compositional verification techniques based on inductive model checking [22].

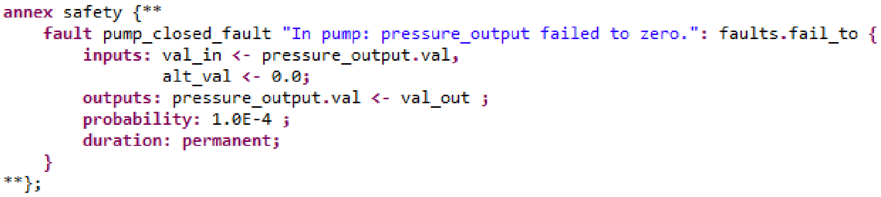
Once the nominal model behavior is defined and verified, the Safety Annex can be used to specify possible faulty behaviors for each component. The faults are defined on each of the relevant components using a customizable library of fault nodes and the faults are assigned a probability of occurrence.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

To illustrate the syntax of the Safety Annex, we use an example based on the Wheel Brake System (WBS) described in [1] and used in our previous work [41]. The fault library contains commonly used fault node definitions. An example of a fault node is shown below:



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

The *fail_to* node provides a way to inject a faulty input value. When the *trigger* condition is satisfied, the nominal component output value is overridden by the *fail_to* failure value. In the WBS, the pump component generates an expected amount of pressure to a hydraulic line. Declaration of a fail to zero fault in the pump component is shown below:



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

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

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

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

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

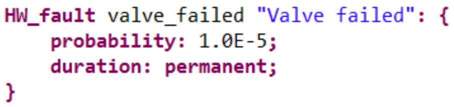
Duration describes the duration of the fault; currently the Safety Annex supports transient and permanent faults.

4.2 Hardware Failures and Dependent Faults

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

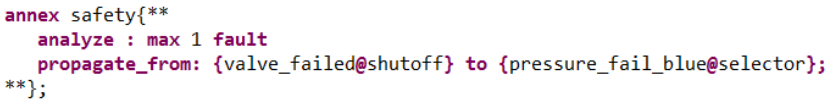
Faults propagate in AGREE as part of a system's nominal behavior. This means that any propagation in the HW portion of an AADL model would have to be artificially modeled using data ports and AGREE behaviors in SW. This is less than ideal as there may not be concrete behaviors associated with HW components.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

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



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

In addition, users can specify fault dependencies outside of fault statements, typically 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). This is because fault propagations are typically tied to the way components are connected or bound together; this information may not be available when faults are being specified for individual components. Having fault propagations specified outside of a component's fault statements also makes it easier to reuse the component in different systems. An example of a fault dependency specification is shown below, showing that the valve_failed fault at the shutoff subcomponent triggers the pressure_fail_blue fault at the selector subcomponent.



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

4.3 Architecture and Implementation

The architecture of the Safety Annex is shown in Figure 4. It is written in Java as a plug-in for the OSATE AADL toolset, which is built on Eclipse. It is not designed as a stand-alone extension of the language, but works with behavioral contracts specified in AGREE AADL annex and associated tools [18]. AGREE allows *assume-guarantee* behavioral contracts to be added to AADL components. The language used for contract specification is based on the Lustre dataflow language [24]. AGREE improves scalability of formal verification to large systems by decomposing the analysis of a complex system architecture into a collection of smaller verification tasks that correspond to the structure of the architecture.

AGREE contracts are used to define the nominal behaviors of system components as *guarantees* that hold when *assumptions* about the values the component's environment are met. The Safety Annex extends these contracts to allow faults to modify the behavior of component inputs and outputs. To support these extensions, AGREE implements an

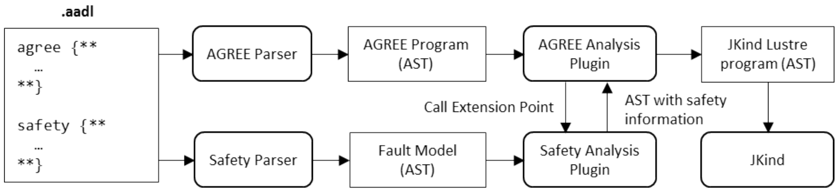
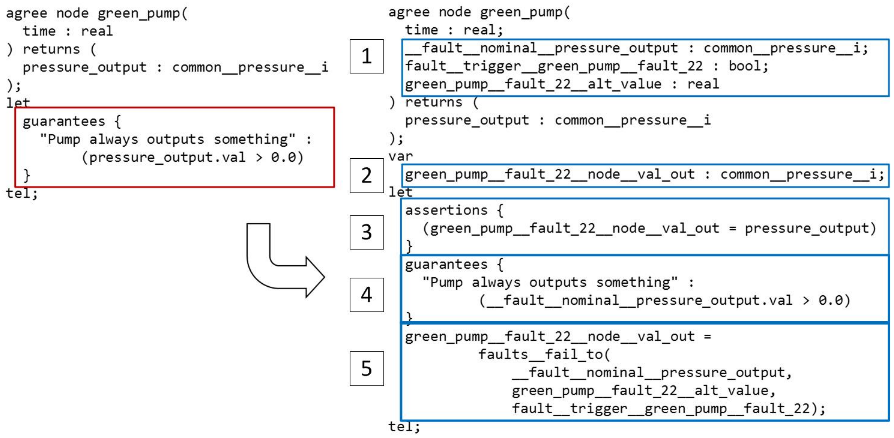


Fig. 4. Safety Annex Plug-in Architecture

Eclipse extension point interface that allows other plug-ins to modify the generated abstract syntax tree (AST) prior to its submission to the solver. If the Safety Annex is enabled, these faults are added to the AGREE contract and, when triggered, override the nominal guarantees provided by the component. An example of a portion of an initial AGREE node and its extended contract is shown in Figure 5. 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.



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

```
agree node green_pump(
  time : real;
  __fault__nominal__pressure_output : common__pressure__i;
  fault__trigger__green_pump__fault_22 : bool;
  green_pump__fault_22__alt_value : real
) returns (
  pressure_output : common__pressure__i
);
var
  green_pump__fault_22__node__val_out : common__pressure__i;
let
  assertions {
    (green_pump__fault_22__node__val_out = pressure_output)
  }
  guarantees {
    "Pump always outputs something" :
          (__fault__nominal__pressure_output.val > 0.0)
  }
  green_pump__fault_22__node__val_out =
          faults__fail_to(
              __fault__nominal__pressure_output,
              green_pump__fault_22__alt_value,
              fault__trigger__green_pump__fault_22);
tel;
```

Fig. 5. Nominal AGREE Node and Extension with Faults

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

In the latter, we determine all combinations of faults whose probabilities are above the specified probability threshold, and describe this as a proposition over *fault_trigger* variables. With the introduction of dependent faults, active faults are divided into two categories: independently active (activated by its own triggering event) and dependently active (activated when the faults they depend on become active). The top level fault hypothesis applies to independently active faults. Faulty behaviors augment nominal behaviors whenever their corresponding faults are active (either independently active or dependently active).

Once augmented with fault information, the AGREE model follows the standard translation path to the model checker JKind [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 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. 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 $\mathcal{R}$. The reason we can do this is because of the arrangement in priority queue from highest to lowest value. If this combination is below threshold, certainly any other combination of these faults with one of lesser value in the priority queue will also be below threshold.

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

```
$\mathcal{F} = \{\}$ : fault combinations above threshold ;
$\mathcal{Q}$ : faults, $q_i$, arranged with probability high to low ;
$\mathcal{R} = \mathcal{Q}$ , with $r \in \mathcal{R}$;
while $\mathcal{Q} \neq \{\} \wedge \mathcal{R} \neq \{\}$ do
    $q$ = removePriorityElement($\mathcal{Q}$) ;
    for $i = 0 : |\mathcal{R}|$ do
        $prob = q \times r_i$ ;
        if $prob < threshold$ then
            removeTail($\mathcal{R}, j = i : |\mathcal{R}|$);
        else
            add($\{q, r_i\}, \mathcal{Q}$);
            add($\{q, r_i\}, \mathcal{F}$);
```

After all possible fault combinations are computed from Algorithm 1, we look at the collection of propagation statements used in HW fault definitions and add additional faults into the possible fault combinations if a fault that triggers the fault can become active, as computed from Algorithm 1.

At the end of Algorithm 1, the possible fault combinations reside in the list $\mathcal{F}$. We then look at the collection of propagation statements used in HW fault definitions. These have a source (HW fault) and destination (faults triggered by HW fault).

Let $\mathcal{P}$ be the collection of propagation statements. For all $S \subset \mathcal{F}$, check to see if for $f \in S$, $f \in \mathcal{P}$ as a source. If so, add the corresponding destinations to the set S. This set $\mathcal{F}$ of allowed fault combinations is then added as a constraint to the Lustre model and thus they become active. If an active fault causes the violation of a contract, this is seen in a counterexample provided by the model checker.

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.

6 Case Studies

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

6.1 Wheel Brake System

The Wheel Brake System (WBS) described in AIR6110 [1] is a well-known example that has been used as a case study for safety analysis, formal verification, and contract based design [8, 12, 13, 27]. The preliminary work for the safety annex used a simplified model of the WBS [41]. In order to demonstrate scalability of our tools and compare results with other studies, we constructed a functionally and structurally equivalent AADL version of one of the most complex WBS xSAP models (arch4wbs) described in [13].

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

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

The WBS is composed of two main parts: the control system and the physical system. The control system electronically controls the physical system and contains a redundant Braking System Control Unit (BSCU) in case of failure. In addition to the redundant BSCU channel, the control system is composed of a number of logical components including sensors for the wheels and brake pedal position, a monitor system that checks validity of the BSCU channel, and the command system which commands braking for each of the 8 wheels. The control system is primarily used in the normal mode of operation to command brake pressure.

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

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

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

The last mode of operation of the WBS is the *emergency* mode. This is supported by the blue circuit but operates if the blue hydraulic pump fails. The accumulator pump has a reserve of pressurized hydraulic fluid and will supply this to the blue circuit in emergency mode.

The wheel brake system architecture is shown in Figure 6 and for simplicity, the diagram displays only two of the eight wheels.

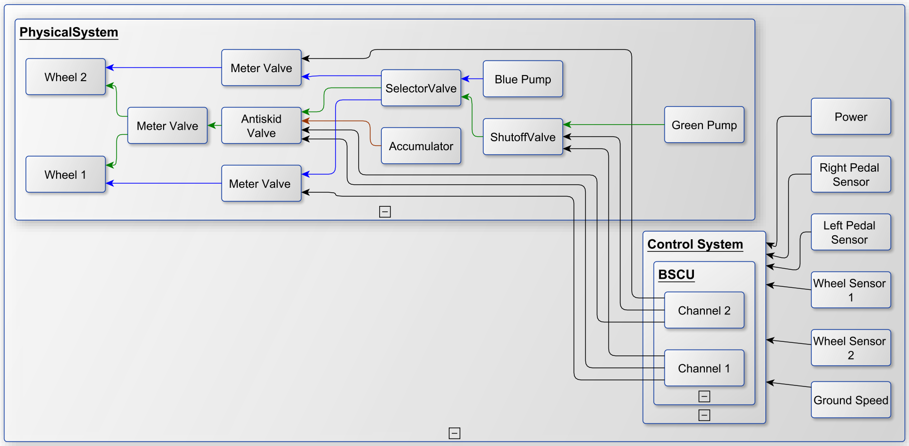
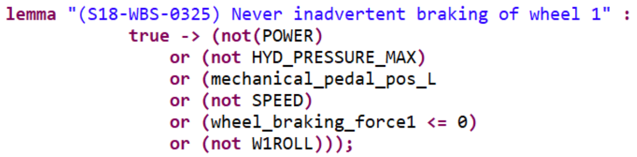


Fig. 6. Wheel Brake System

The model contains 30 different kinds of components, 169 component instances, a model depth of 5 hierarchical levels. The model includes one top-level assumption and 27 top-level system properties, with 113 guarantees allocated to subsystems. There are a total of 33 different fault types and 141 fault instances within the model. The large number of fault instances is due to the redundancy in the system design and its replication to control 8 wheels.

An example 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)));
```

Fault Analysis of WBS using Safety Annex 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.

6.2 Quad-Redundant Flight Control System

We have also used the Safety Annex to examine more complex fault types, such as asymmetric (or *Byzantine*) faults. A Byzantine fault presents different symptoms to different observers, so that they may disagree regarding whether a fault is present. We extended the Quad-Redundant Flight Control System (QFCS) example [3] to model and analyze various types of faulty behaviors. Faulty behaviors were introduced to analyze the response of the system to multiple faults, and to evaluate fault mitigation logic in the model. As expected, the QFCS system-level properties failed when unhandled faulty behaviors were introduced.

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

– Byzantine faults [19] were simulated by creating one-to-one connections from the source to multiple observers so that disagreements could be introduced by injecting faults on individual outputs. The system level property "at most one flight control computer in command" was falsified in one second in the presence of Byzantine faults on the baseline model. The same property was verified in three seconds on an extended model with a Byzantine fault handling protocol added. System designers can use this approach to verify if a system design is resilient to Byzantine faults, examine vulnerabilities, and determine if a mitigation mechanism works.
– Dependent faults were modeled by first injecting failures to the cross-channel data link (CCDL) bus (physical layer), and faults to the flight control computer (FCC) outputs (logical layer), then specifying fault propagations in the top level system implementation (where the data connections between FCC outputs were bound to the CCDL bus subcomponents). The fault propagation indicates that one CCDL bus failure can trigger all FCC output faults. With the fault hypothesis that allows a maximum of one fault active during execution, the system level property "not all FCCs fail at the same time" was falsified in one second.

7 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 failure propagations, which allow more natural expression of some failure types (e.g. reverse flow within segments of a pipe), but are more difficult to analyze. A final dimension involves whether analysis is *compositional* across layers of hierarchically-composed systems or *monolithic*. Our approach is an extension of AADL (ESM), causal, compositional, mixed FLM/FEM approach.

Tools such as the AADL Error Model Annex, Version 2 (EMV2) [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 Systems) [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 failure 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].

8 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. 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. 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.
19. K. Driscoll, H. Sivencrona, and P. Zumsteg. Byzantine Fault Tolerance, from Theory to Reality. In *SAFECOMP*, LNCS, 2003.
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. *ArXiv e-prints*, Dec. 2017.
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 Realtime 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, M. Whalen, D. Cofer, and M. P. Heimdahl. Architectural Modeling and Analysis for Safety Engineering. In *IMBSA 2017*, pages 97–111, 2017.