Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 18-007

Safety Annex for the Architecture Analysis and Design Language

Danielle Stewart, Jing (Janet) Liu, Michael W. Whalen, Darren Cofer, Michael Peterson

March 9, 2018
Abstract. This paper describes a new methodology with tool support for model based safety analysis. It is implemented as a Safety Annex for the Architecture Analysis and Design Language (AADL). The Safety Annex provides the ability to describe faults and faulty component behaviors in AADL models. In contrast to previous AADL-based approaches, the Safety Annex leverages a formal description of the nominal system behavior to propagate faults in the system. This approach ensures consistency with the rest of the system development process and simplifies the work of safety engineers. The language for describing faults is extensible and allows safety engineers to weave various types of faults into the nominal system model. The Safety Annex supports the injection of faults into component level outputs, and the resulting behavior of the system can be analyzed using model checking through the Assume-Guarantee Reasoning Environment (AGREE).

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

1 Introduction

System safety analysis techniques are well-established and are a required activity in the development of safety-critical systems. While model based development methods are widely used in the aerospace industry, these methods are only recently being applied to system safety analysis. Model-based systems engineering (MBSE) methods and tools based on formal methods now permit system-level requirements to be specified and analyzed early in the development process [7, 17]. These tools can also be used to perform safety analysis based on the system architecture and initial functional decomposition. Design models can be integrated into the safety analysis process to help guarantee accurate and consistent results. This integration is especially important as the amount of
safety-critical hardware and software in various domains has drastically increased due to the demand for greater autonomy, capability, and connectedness.

We have developed a Safety Annex for the Architecture Analysis and Design Language (AADL) [19] that provides the ability to reason about faults and faulty component behaviors in AADL models. In the Safety Annex approach, we use formal assume-guarantee contracts to define the nominal behavior of system components. The nominal model is then verified using the Assume Guarantee Reasoning Environment (AGREE) [17]. The Safety Annex provides a way to weave faults into the nominal system model and analyze the behavior of the system in the presence of faults. The Safety Annex also provides a library of common fault node definitions that is customizable to the needs of system and safety engineers. Our approach adapts the work of Joshi et. al in [26] to the AADL modeling language. More information on the approach is available in [34], and the tool and relevant documentation can be found at: https://github.com/loonwerks/AMASE/.

There are other tools purpose-built for safety analysis, including AltaRica [31], smartIFlow [24] and xSAP [6]. These notations are separate from the system development model. Other tools extend existing system models, such as HiP-HOPS [14] and the AADL Error Model Annex, Version 2 (EMV2) [20]. EMV2 uses enumeration of faults in each component and explicit propagation of faulty behavior to perform safety analysis. The required propagation relationships must be manually added to the system model and can become complex, leading to potential omissions and inconsistencies.

In contrast, the Safety Annex supports model checking and quantitative reasoning by attaching behavioral faults to components and then using the normal behavioral propagation and proof mechanisms built into the AGREE AADL annex. This allows users to reason about the evolution of faults over time, and produce counterexamples demonstrating how component faults lead to system failures. It can serve as the shared model to capture system design and safety-relevant information, and produce both qualitative and quantitative description of the causal relationship between faults/failures and system safety requirements. Thus, the contributions of the Safety Annex and this paper are:

- Close integration of behavioral fault analysis into an architectural design language AADL, which allows close connection between system and safety analysis and system generation from the model (unlike AltaRica, smartIFlow, and xSAP),
- support for behavioral specification of faults and their implicit propagation through behavioral relationships in the model, in contrast to existing AADL-based annexes (HiP-HOPS and EMV2),
- additional support for explicit propagation of faults to capture binding relationships between hardware and software and logical and physical communications beyond what is supported in xSAP, and
- guidance on integration into a traditional safety analysis process.

2 Preliminaries

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

2.1 Safety Assessment Process

ARP4754A, the Guidelines for Development of Civil Aircraft and Systems [33], has been recognized by the Federal Aviation Administration (FAA) as an “acceptable method for establishing a development assurance process” [1]. It provides guidance on applying development assurance at each hierarchical level throughout the development life cycle of highly-integrated/complex aircraft systems.

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 [32], identifies a systematic means to show compliance. The guidelines presented in ARP4761 include industry accepted safety assessment processes (Functional Hazard Assessment (FHA), Preliminary System Safety Assessment (PSSA), and System Safety Assessment (SSA)), and safety analysis methods to conduct the safety assessment, such as Fault Tree Analysis (FTA), Failure Modes and Effect Analysis (FMEA), and Common Cause Analysis (CCA).

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

In practice, prior to performing the safety assessment of a system, the safety engineers are often equipped with the domain knowledge about the system, but do not necessarily have detailed knowledge of the current system is implemented. 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 implementation details.

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

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

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

Figure 1 presents our proposed use of this shared system design and safety analysis model in the context of the ARP4754A Safety Assessment Process Model (derived from Figure 7 of ARP4754A). The shared model is one of the system development artifacts from the “Development of System Architecture” and “Allocation of System Requirements to Item” activities in the System Development Process, which interacts with the PSSAs and SSAs activities in the Safety Assessment Process. The shared model can
Fig. 2. Example Interactions between the Shared System/Safety Model and the FTAs

serve as an interface to capture the information from the system design and implementation that is relevant for the safety analysis.

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

2.2 Modeling Language for System Design

We are using the Architectural Analysis and Design Language (AADL) [19] to construct system architecture models. AADL is an SAE International standard [4] that defines a language and provides a unifying framework for describing the system architecture for "performance-critical, embedded, real-time systems" [4]. 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) [17] 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 [23] and then queries a user-selected model checker to conduct the back-end analysis. The analysis is 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 [34], 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. In particular, our goals are to:

- Provide a comprehensive, qualitative description of the causal relationship between basic failure events and system level safety requirements.
- Provide an accurate, quantitative description of the contribution relationship between failure rates of the fault tree basic events and numerical probability requirements at the system level.

3 The Safety Annex

In this section, we describe the main features and functionality of the Safety Annex. The usage of the terms error, failure, and fault follow their definitions in ARP4754A [33]. We use fault as the generic modeling keyword throughout the AADL model hierarchy.

3.1 Basic Functionality

An AADL model of the nominal system behavior specifies the hardware and software components of the system and their interconnections. This nominal model is then annotated with assume-guarantee contracts using the AGREE annex [17] for AADL. The nominal model requirements are verified using compositional verification techniques based on inductive model checking [21].

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 [2] and used in our previous work [34]. The fault library contains commonly used fault node definitions. An example of a fault node is shown below:

```plaintext
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;
```

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:

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

The `fault statement` consists of a unique description string, the fault node definition name, and a series of `fault subcomponent` statements. **Inputs** in a fault statement are the parameters of the fault node definition. In the example above, `val_in` and `alt_val` are the two input parameters of the fault node. These are linked to the output from the Pump component (`pressure_output.val`), and `alt_value`, a fail to value of zero. When the analysis is run, these values are passed into the fault node definition. **Outputs** of the fault definition correspond to the outputs of the fault node. The fault output statement links the component output (`pressure_output.val`) with the fault node output (`val_out`). If the fault is triggered, the nominal value of `pressure_output.val` is overridden by the failure value output by the fault node. Faulty outputs can take deterministic or non-deterministic values. **Probability** (optional) describes the probability of a fault occurrence. **Duration** describes the duration of the fault; currently the Safety Annex supports transient and permanent faults.

### 3.2 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 systems 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 specify fault dependencies/propagations outside of fault statements and inside safety annex, typically in the system implementation where the system configuration that causes the dependencies (e.g., binding between SW and HW components, co-location of HW components) becomes clear. 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 components 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};
}
```

### 3.3 Architecture and Implementation

The architecture of the Safety Annex is shown in Figure 3. It is written in Java as a plug-in for the OSATE AADL toolset, which is built on Eclipse. It is not designed as a stand-alone extension of the language, but works with behavioral contracts specified AGREE AADL annex and associated tools [17]. 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 [23]. 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 Eclipse extension point interface that allows other plug-ins to modify the generated abstract syntax tree (AST) prior to its submission to the solver. If the Safety Annex is
enabled, these faults are added to the AGREE contract and, when triggered, override the nominal guarantees provided by the component. An example of a portion of an initial AGREE node and its extended contract is shown in Figure 4. The _fault variables and declarations are added to allow the contract to override the nominal behavioral constraints (provided by guarantees) on outputs. In the Lustre language, assertions are constraints that are assumed to hold in the transition system.

```plaintext
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
);
let 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 tel;
    __fault_nominal_pressure_output,
    green_pump_fault_22_alt_value,
    fault_trigger_green_pump_fault_22);
tel;
```

Fig. 4. Nominal AGREE node and its extension with faults

An annotation in the AADL model determines the fault hypothesis. This may specify either a maximum number of faults that can be active at any point in execution (typically one or two), or that only faults whose probability of simultaneous occurrence is above some probability threshold should be considered. In the former case, we assert that the sum of the true fault_trigger variables is below some integer threshold. In the latter, we determine all combinations of faults whose probabilities are above the specified probability threshold, and describe this as a proposition over fault_trigger variables. With the introduction of dependent faults, active faults are divided into two categories: independently active (activated by its own triggering event) and dependently active (activated when the faults they depend on become active). The top level fault hypothesis applies to independently active faults. Faulty behaviors augment nominal
behaviors whenever their corresponding faults are active (either independently active or dependently active).

Once augmented with fault information, the AGREE model follows the standard translation path to the model checker JKind [21], 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.

4 Case Studies

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

4.1 Wheel Brake System

The Wheel Brake System (WBS) described in ARP4761 [32] has been used in the past as a case study for safety analysis, formal verification, and contract based design [7, 10, 11, 15, 25, 27, 34]. The preliminary work for the safety annex used a simplified model of the WBS [34]. In order to demonstrate scalability of our tools and compare results with other studies, we constructed a functionally and structurally equivalent AADL version of the most complex WBS xSAP model (arch4wbs) described in previous work [11]. We describe the elaborations of this model to the ARP4761 WBS below.

WBS architecture description The WBS is composed of two main systems: the control system and the physical system. The control system electronically controls the physical system and contains a redundant Braking System Control Unit (BSCU) in case of failure. The physical system consists of the hydraulic circuits running from hydraulic pumps to wheel brakes. This is what provides braking force to each of the 8 wheels of the aircraft.

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

In alternate mode, the system uses the blue hydraulic circuit. The wheels are all mechanically braked in pairs (one pair per landing gear). The alternate system is composed of the blue hydraulic pump, four meter valves, and four antiskid shutoff valves. The meter valves are mechanically commanded through the pilot pedal corresponding to each landing gear. If the system detects lack of pressure in the green circuit, the selector valve switches to the blue circuit. This can occur if there is a lack of pressure from the green hydraulic pump, if the green hydraulic pump circuit fails, or if pressure is cut off by a shutoff valve. If the BSCU unit becomes invalid, the shutoff valve is closed.
The last mode of operation of the WBS is the emergency mode. This is supported by the blue circuit but operates if the blue hydraulic pump fails. The accumulator pump has a reserve of pressurized hydraulic fluid and will supply this to the blue circuit in emergency mode.

The model, which is available at [3], contains 30 different kinds of components, 169 component instances, a model nesting depth of 5 levels (an AADL system is decomposed into other systems, which are further decomposed, up to a depth of 5 levels). The total model involves 17 assumptions and 113 guarantees, with 11 top-level system properties. 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 model and the use of 8 wheels.

An example property is to ensure no inadvertent braking of each of the 8 wheels, such that if all power and hydraulic pressure is supplied, then either the system has no ground speed, or mechanical pedal is pressed, or brake force is zero, or wheel is not rolling.

**Fault Analysis of WBS using Safety Annex** Fault analysis on the top level WBS system was performed on 11 top-level properties using two fault hypotheses: the first allows at most one fault and the second allows combinations of faults that exceed the acceptable probabilities for the top-level hazard defined in ARP4761 [32].

We examine scalability by looking at analysis times for both compositional verification, where the verification task is split into layers to be analyzed separately, and monolithic verification, where the entire system is analyzed in one pass. When considering the nominal system behavior (no faults), the total time required for analysis using compositional verification is 2 minutes 38 seconds, and the time for monolithic analysis is 30 seconds. This nominal model is too small to gain significant benefit from compositional analysis. However, when we consider faulty behavior, when given a single-fault hypothesis, the total time for compositional analysis is 2 minutes 37 seconds, and monolithic analysis did not finish after 42 minutes (ended due to out of memory error), clearly demonstrating the value of compositional analysis for more complex models. Similarly, for probabilistic analysis, the compositional time is 2 minutes 38 seconds and the monolithic time is 3 minutes 31 seconds.

In our analysis, we discovered that most properties pass the model, but the Inadvertent braking at the wheel properties were not resilient to a single fault nor did they meet the desired $10^{-9}$ fault threshold for probabilistic analysis. In our model (as in the NuSMV model [11]), there is a single pedal position sensor for the brake pedal. If this sensor fails, it can command braking without a pilot request. It was straightforward for us to diagnose the cause of the property failure using an automatically generated counterexample, which is a test case that demonstrates the failure.

This counterexample can be used to further iterate system design. For our model, it could be that the time in the V1 phase of flight is short enough that we need to adjust our model failure rates for the V1 time scale, or that redundant sensors are required on the pedals (here we note that the architecture of the pedal assembly is not discussed in ARP4761). It is straightforward and computationally inexpensive to run the analysis, allowing quick iterations between systems and safety engineers. As indicated in Figure 2,
the sync and update between the preliminary system FTA and the architecture/analysis model continues until the system safety property is satisfied with the desired fault tolerance and failure probability achieved.

4.2 Quad-Redundant Flight Control System

In order to discuss Byzantine faults and hardware failures with their propagations, we applied the Safety Annex to the Quad-Redundant Flight Control System (QFCS) model [5]. Faulty behaviors were introduced in order to see the response of the system to several faults, and to evaluate fault mitigation logic in the model. 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 [18] 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 verified false 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. System designers can use this approach to verify if a system design is resilient to Byzantine faults, examine vulnerabilities, and test 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 maximum one fault active during execution, the system level property “not all FCCs fail at the same time” was verified false in one second.

5 Related Work

In recent years, there has been an increase in the interest of Model Based Safety Analysis (MBSA) [13].

Formal model based systems engineering (MBSE) methods and tools now permit system level requirements to be specified and analyzed early in the development process [5,16,17,29,30]. Design models from which aircraft systems are developed can be integrated into the safety analysis process to help guarantee accurate and consistent results. This integration is especially important as the amount of safety critical hardware and software has drastically increased in safety critical domains such as aerospace, automotive, and medical fields [34].

There are tools that currently support reasoning about faults in architecture description languages such as SysML and AADL. These tools include the AADL Error Model.
Annex, Version 2 (EMV2) [20] and HiP-HOPS for EAST-ADL [14]. These approaches primarily utilize qualitative reasoning. Faults are enumerated and the propagations through system components are explicitly described. Given many possible faults, these propagation relationships increase in complexity and understandability. Interactions are easily overlooked by analysts and thus not explicitly described. This is also the case with tools like SAML that incorporate both qualitative and quantitative reasoning [22].

In earlier work, an approach to MBSA was demonstrated using the Simulink notation [25–28]. In this approach, a behavioral model of system dynamics was used to reason about the effects of faults in the system. This approach allows an implicit and natural notion of fault propagation through the system. However, non-functional architectural properties were not captured as Simulink is not designed as an architecture description language. In our approach, we are applying quantitative reasoning and implicit fault propagation to a more rich architecture language.

Formal verification tools based on model checking have been used to automate the generation of safety artifacts [6, 8, 9, 12, 13]. 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 [10].

6 Conclusion

We have developed an extension to the AADL language with tool support for formal analysis of system safety properties in the presence of faults. Faulty behavior is specified as an extension of the nominal model, allowing safety analysis and system implementation to be driven from a single common model. This new Safety Annex leverages the AADL structural model and nominal behavioral specification (using the AGREE annex) to propagate faulty component behaviors without the need to add separate propagation specifications to the model. Next steps will include extensions to automate injection of Byzantine faults as well as automatic generation of fault trees.

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

References


