Safety Annex for Architecture Analysis and Design Language

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

[1] University of Minnesota
Department of Computer Science and Engineering
dkstewar, whalen@cs.umn.edu
[2] Rockwell Collins
Advanced Technology Center
Jing.Liu, darren.cofer@rockwellcollins.com

Abstract. This paper describes a new methodology with tool support for model based safety analysis. It is implemented as a new 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 [5, 6]. 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) [9] 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 behavior of system components. The nominal model is then verified using the Assume Guarantee Reasoning Environment (AGREE) [6]. 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.

There are other tools purpose-built for safety analysis, including AltaRica [14], smartIFlow [12] and xSAP [**?**]. These notations are separate from the system development model. Other tools extend existing system models, such as HiP-HOPS [1] and the AADL Error Model Annex, Version 2 (EMV2) [7]. EMV2 uses enumeration of faults in each component and explicit propagation of faulty behavior to perform error analysis. The required propagation relationships must be manually added to the system model and can become complex, leading to mistakes in the analysis.

In contrast, the Safety Annex supports model checking and quantitative reasoning by attaching behavioral faults to components and then using the normal behavioral propagation and proof mechanisms built into the AGREE AADL annex. This allows users to reason about the evolution of faults over time, and produce counterexamples demonstrating how component faults lead to failures. Our approach adapts the work of Joshi et. al in [13] to the AADL modeling language. More information on the approach is available in [17], and the tool and relevant documentation can be found at: `https://github.com/loonwerks/AMASE/`.

2 Preliminaries

In this section, we describe the overall process flow and introduce some terminology associated with the certification context. We also describe the system architecture modeling and verification environment that we are using.

2.1 Safety Assessment Process

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

ARP4754A, the Guidelines for Development of Civil Aircraft and Systems [15], has been recognized by the Federal Aviation Administration (FAA) as an "acceptable method for establishing a development assurance process" [2]. It provides guidance on applying development assurance at each hierarchical level throughout the development lifecycle of highly-integrated/complex aircraft systems.

The safety assessment process is a starting point at each hierarchical level of the development lifecycle, and is tightly coupled with the system development and verification processes. It is used to show compliance with certification requirements, and meeting a company's internal safety standards [15]. ARP4761, the Guidelines and Methods for Conducting Safety Assessment Process on Civil Airborne Systems and Equipment [16], 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 works, primarily focusing on the integrity of the outputs and the availability of the product. The safety engineers then use the acquired understanding to construct the safety analysis artifacts, conduct safety analysis, 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 fair amount of knowledge on how system works in general, but not necessarily with the specific system. Acquiring the knowledge on the content and behavior of a specific system has shown to be time consuming to get it right. In one real case example, it took a safety engineer two solid days to understand how the software works in a Stall Warning System (a small system in comparison to a flight control computer). The primary task includes threading 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 the same amount of time, if not more, as constructing the analysis artifacts and performing the analysis itself. In another real case example, it took a safety engineer almost a year to finalize the PSSA document for a Horizontal Stabilizer Control System (a medium system in comparison to a flight control computer), involving two major revisions and multiple rounds of reviews with system, hardware, and software engineers.

Capturing failure mode in models and generating safety analysis artifacts directly from models can enhance communication and synchronization between system design process and safety assessment process, and the ability to analyze complex systems. 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 with model based safety analysis appendix is under way.

Using the same system design model to conduct both system design and safety analysis can help reduce the gap in comprehending the system behavior and transferring the knowledge between the design model to the safety analysis model. It maintains a living model that captures the latest state of the system design as the process flows per the system development lifecycle. It also allows all participants of the ARP4754A process to be able to communicate and review the system design using the “single source of truth”.

In order to allow performing system design and safety analysis on the same model, the system design model will need to be augmented to include both the system design information (e.g., system architecture, functional behavior) and safety-relevant information (e.g., failure mode, failure rate), at the same time keeping the two types of information distinguishable yet interactable from each other.

Figure 1 presents our proposed use of the shared system design and safety analysis model (to be referred as ”the shared model” in the rest of this section) inside the

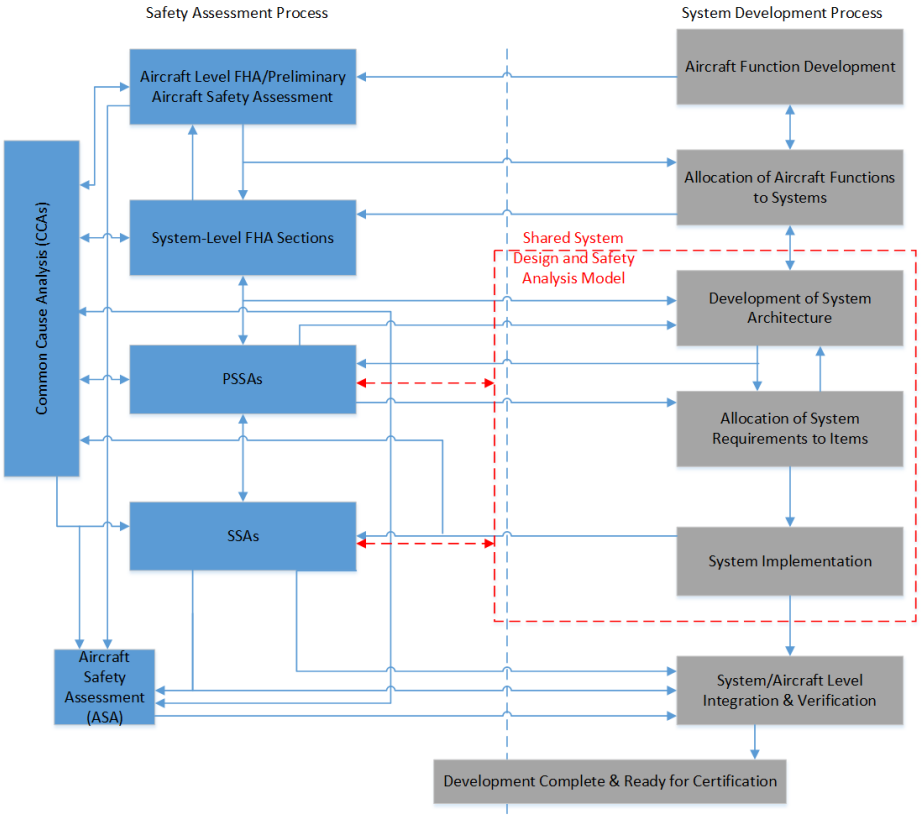

Fig. 1. Using the Shared System/Safety Model in the ARP4754A Safety Assessment Process

ARP4754A Safety Assessment Process Model (Figure 7 of [15]). As seen in the figure, the shared model represents a system development artifact 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 serve as a wrapper and interface to capture the information relevant to safety analysis from the system design and implementation.

Figure 2 shows an example how the preliminary FTAs and FTAs (artifacts from the PSSA and SSA activities in the Safety Assessment Process) can guide and be updated from the shared model. The next subsection will describe the foundation of the modeling technique in more details.

2.2 Architecture Description and Design Contracts

The Architectural Analysis and Design Language (AADL) [**?**] is a architecture modeling language for embedded, real-time, distributed systems. It was approved as an SAE Standard in 2004, and its standardization committee has active participation from many

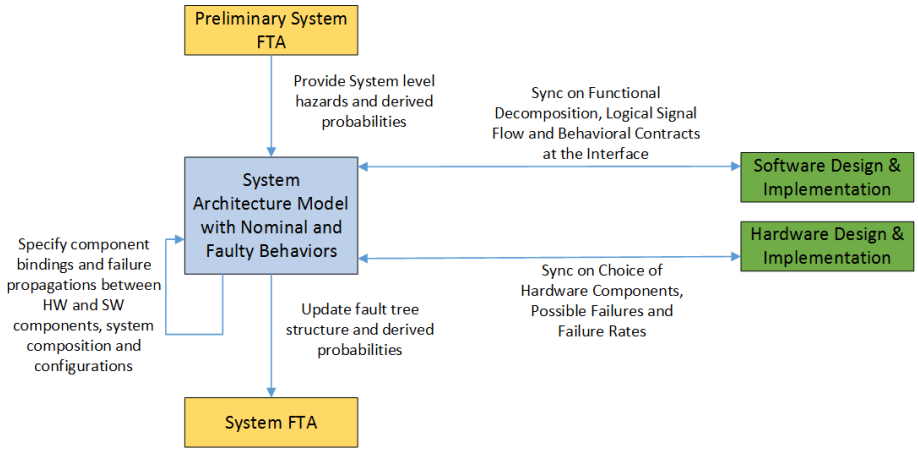

Fig. 2. Example Interactions between the Shared System/Safety Model and the FTAs

academic and industrial partners in the aerospace industry. It provides the constructs needed to model both hardware and software in embedded systems such as threads, processes, processors, buses, and memory. It is sufficiently formal for our purposes, and is extensible through the use of language annexes that can initiate calls to separately developed analysis tools.

The Assume Guarantee Reasoning Environment (AGREE) [**?**] is a language and tool for compositional verification of AADL models. It is implemented as an AADL annex that allows AADL models to be annotated with assume-guarantee behavioral contracts. A contract contains a set of assumptions about the component's inputs and a set of guarantees about the component's outputs. The assumptions and guarantees may also contain predicates that reason about how the state of a component evolves over time.

AGREE uses a syntax similar to Lustre [**?**] to express a contract's assumptions and guarantees. AGREE translates an AADL model and its contract annotations into Lustre and then queries a user-selected model checker to perform verification. The goal of the analysis is to prove that each component's contract is satisfied by the interaction of its direct subcomponents as described by their respective contracts. Verification is performed at each layer of the architecture hierarchy and details of lower level components are abstracted away during verification of higher level component contracts. This compositional approach allows large systems to be analyzed efficiently.

Component contracts at the lowest level of the architecture are assumed to be true by AGREE. Verification of these component contracts must be performed outside of the AADL/AGREE environment. In a traditional software development process, components will be developed to meet the high-level requirements corresponding to these contracts and verified by testing or code review. However, there are two problems with this approach:

1. Verification methodologies like test and code review are not exhaustive. Errors in these activities can cause the compositional verification that AGREE performs to be incorrect.
2. Manual translation of an AGREE contract into a property for verification at the component level can be time-consuming and error-prone.

Our solution to these problems is to automatically translate AGREE contracts of software components into expressions in the development language of the component software. A formal verification tool that reasons about artifacts expressed in this language can then be used to verify that the contracts hold. The remainder of the paper describes this solution in detail.

3 Detailed Approach

In this section, we describe the main features and functionality of the Safety Annex.

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 [6] for AADL. The nominal model requirements are verified using compositional verification techniques based on k-induction model checking [10].

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 [3] and used in our previous work [17]. 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 non-deterministic fault in the pump component is shown below:

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

```
annex safety {**
  fault "In pump: pressure_output failed to non-deterministic value.":   faults.fail_to {
    eq alt_value :real;
    inputs: val_in <- pressure_output,
            alt_val <- alt_value;
    outputs: pressure_output.val <- val_out ;
    probability: 1.0E-5 ;
    duration: permanent;
}
**};
```

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 nondeterministic value defined within the Safety Annex. 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 Fault and Dependent Fault

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

Faults propagate in AGREE as part of a systems nominal behavior. This means that any fault 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 HW dependent faults, 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 faults, nor data ports to apply the fault. An example of a HW 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:

```
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 [6]. 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 [11]. 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.

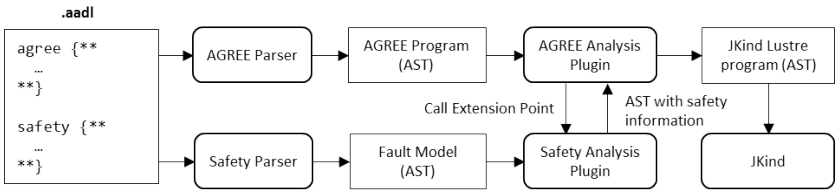

Fig. 3. Safety Annex Plug-in Architecture

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

```
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. 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). 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 AGREE translation path to the model checker JKind [10], 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 evaluate the effectiveness of the Safety Annex, we updated the WBS model [17] to specify faulty component behaviors. The components' nominal and faulty behaviors are modeled separately. At the top-level AADL component, the fault hypothesis was specified as the maximum number of faults that can be active at any time. The AGREE contracts at the top-level component were verified using AGREE, with the "Perform Safety Analysis" option selected. This signals the tool to weave the nominal and faulty behaviors into one augmented AGREE model before feeding to the model checker.

In this example, the top level contract "Pedal pressed and no skid implies brake pressure applied" was verified in the presence of at most one fault active during execution. However, it was shown to be invalid when more than one fault was allowed. The counterexample indicated that both Selector's outputs failed to non-deterministic values due to the faults introduced.

We also applied the Safety Annex to the Quad-Redundant Flight Control System (QFCS) model [4]. We introduced faulty behaviors 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 [8] 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. A system-level property failed due to the fault on the baseline model, but did not fail on the model with Byzantine fault handling protocol added. Using the Safety Annex like this can test a system's vulnerability to Byzantine faults and verify mitigation mechanisms.
- Dependent faults in hardware were simulated by injecting faults to hardware components (physical layer) to affect their data outputs (logical layer), and consequently failing the software components bound to the hardware components. The relationship between the hardware components' outputs and the software components' inputs were specified in AGREE as part of the system's nominal behavior.

4.1 Safety Process Related Case Study

value added to traditional safety assessment

4.2 Scalability Related Case Study

Talk about the findings from translating WBS LTL properties into AGREE/Safety Annex, e.g., missing including power in the system validity assessment that caused the compositional verification to fail?

5 Related Work

Address NFM review comments

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 the ability to specify dependent faults.

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

References

1. Systems Modeling with EAST-ADL for Fault Tree Analysis through HiP-HOPS*. *IFAC Proceedings Volumes*, 46(22):91 – 96, 2013.
2. AC 20-174. Development of Civil Aircraft and Systems, September 2011.
3. AIR 6110. Contiguous Aircraft/System Development Process Example, Dec. 2011.
4. J. Backes, D. Cofer, S. Miller, and M. W. Whalen. Requirements Analysis of a Quad-Redundant Flight Control System. In K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 9058 of *Lecture Notes in Computer Science*, pages 82–96. Springer International Publishing, 2015.
5. M. Bozzano, A. Cimatti, A. Griggio, and C. Mattarei. Efficient Anytime Techniques for Model-Based Safety Analysis. In *Computer Aided Verification (CAV '11*, 2015.
6. D. D. Cofer, A. Gacek, S. P. Miller, M. W. Whalen, B. LaValley, and L. Sha. Compositional Verification of Architectural Models. In A. E. Goodloe and S. Person, editors, *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, volume 7226, pages 126–140, Berlin, Heidelberg, April 2012. Springer-Verlag.
7. J. Delange, P. Feiler, J. Hudak, and D. Gluch. Architecture fault modeling and analysis with the error model annex, version 2, 06 2016.
8. K. Driscoll, H. Sivencrona, and P. Zumsteg. Byzantine Fault Tolerance, from Theory to Reality. In *SAFECOMP*, volume 2788 of *LNCS*, pages 235–248, 2003.
9. P. Feiler and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley Professional, 2012.
10. A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani. The JKind Model Checker. *ArXiv e-prints*, Dec. 2017.
11. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Dataflow Programming Language Lustre. In *In Proceedings of the IEEE*, volume 79(9), pages 1305–1320, 1991.
12. P. Hönig, R. Lunde, and F. Holzapfel. Model Based Safety Analysis with smartIflow. *Information*, 8(1), 2017.
13. 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 (Awarded Best Paper of Track)*, 2005.
14. 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 Proceedings Volumes*, 46(22):127 – 132, 2013.
15. SAE ARP4754A. Guidelines for Development of Civil Aircraft and Systems, December 2010.
16. SAE ARP4761. Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

17. D. Stewart, M. Whalen, D. Cofer, and M. P. Heimdahl. Architectural Modeling and Analysis for Safety Engineering. In *In Proceedings of IMBSA2017: 5th International Symposium on Model Based Safety and Assessment*, 2017.