Architecture Fault Modeling with the AADL Error-Model Annex

Julien Delange and Peter Feiler
Carnegie Mellon Software Engineering Institute
4500 5th Avenue
Pittsburgh, PA15218
Email: {jdelange,phf}@sei.cmu.edu

Abstract—Safety-Critical systems, as used in the automotive, avionics, or aerospace domains, are becoming increasingly software-reliant to the extent that the system cannot function without the software. On one hand the software system provides an integrated set of functionality to operate the system and manage failure and unsafe conditions. Current best safety engineering practices, such as DO178B/C or SAE ARP4761, are labor intensive and are only performed as part of the system engineering process. At the same time increased interaction complexity of the embedded software with the hardware platform and mechanical system has resulted in the software to be a major source of defects with potentially fatal consequences.

To address these issues, the SAE Architecture Analysis & Design Language (AADL) standard has been extended with an Error Model Annex to support architecture fault modeling and automated safety analysis. In this paper we introduce the concepts of the revised Error Model (EMV2) Annex and a fault propagation ontology to support such architecture fault models at three levels of abstraction focusing on fault propagation, failure behavior of individual components, and composite failure behavior of a system in terms of its components. Such specifications reflect fault tolerance strategies assumptions made by fault impact, fault tree and reliability analysis about the safety system component. We illustrate their use on a dual redundant flight guidance system and discuss the automation of different safety analysis methods in use by the SAE ARP4761, emphasizing on automation benefits.

I. INTRODUCTION

Safety-critical systems are becoming increasingly software-reliant and in growth in interaction complexity between embedded software components, the hardware platform, and the mechanical system being managed and controlled has resulted in increasing rework cost due to verification. This is due to 70% of the defects being introduced in requirements and architecture design, and 80% detected post unit test, making their development increasingly not affordable [1].

At the same time, current best safety engineering practices, such as DO-178B/C[2], SAE ARP-4754 and SAE ARP-4761 [3] are labor intensive, typically performed during system engineering and not repeated frequently. This results in a major gap between system and software requirements [4]. These safety practices include methods such as Functional Hazard Assessment (FHA), System Safety Assessment (SSA), Common Cause Failure Analysis (CCFA), methods such as Failure Modes and Effect Analysis (FMEA) and its variant Failure Mode, Effects, and Criticality Analysis (FMECA), Fault Tree Analysis (FTA), availability and reliability prediction via reliability block diagrams and stochastic models. The models developed by engineers become quickly outdated as system design evolves. This makes it a challenge to maintain consistency between these models and validity of the analysis results. Architecture trade-off studies are often performed without fully exercising the different safety analyses due to their high cost.

One of the objectives of the SAE Error Model EMV2 Annex standard, an extension to the SAE Architecture Analysis and Design Language (AADL) standard is to automate safety analysis methods by supporting them through analyzable architecture fault models. It allows the user to annotate system and software architectures expressed in AADL to be annotated with hazard, fault propagation, failure modes and effects due to failures, as well as compositional fault behavior specifications to facilitate incremental and scalable automated safety analysis. In that context:

- We have augmented the original Error Model Annex language for AADL [5] with safety semantics and a fault propagation ontology to support the modeling of error behaviors. The resulting revised Error-Model V2 (EMV2) Annex standard for architecture fault modeling is currently in ballot.
- We have developed new analysis tools that automatically evaluate system safety with the augmented AADL description (core language + EMV2). Such tools support recommended safety practices and produce appropriate analysis reports.

This paper presents our contributions to the standard as well as an example that illustrates its application. Existing work based on the original Error Model Annex partially addresses the automation of safety evaluation, for example by generating failure effect analysis [6] and Fault-Tree Analysis [7], [8], [9]. However, this first version of the Error-Model annex has limitation in expressing desired semantics of architectural fault behavior, such as:

- separation of error events from recover and repair events requiring separate specifications for fault tree generation and for reliability and availability analysis.
- support fault propagation contracts between components by explicit specification of propagations expected not to occur.
• specification of error types
• expression of errors through component bindings
• extension mechanisms to add additional (and potentially user-defined) safety-related information

The revision of the Error Model Annex draws on user experiences with the first version, e.g., [10], and experimental extensions, (e.g., SLIM as part of the COMPASS project [6]). The fault propagation ontology draws on previous work by Powell [11] and Walter [12]. The resulting architecture fault models support a full range of safety analysis.

The paper is organized as follows. First, we summarize the concepts of the SAE Architecture Analysis and Design Language (AADL), which is the basis for architecture fault models supporting safety analysis. Next, we introduce the concepts of the revised Error Model (EMV2) Annex and a fault propagation ontology to support such architecture fault models at three levels of abstraction: focusing on fault propagation, on failure behavior of individual components, and on composite failure behavior of a system in terms of its components. Such specifications reflect fault tolerance strategies assumptions made by fault impact, fault tree and reliability analysis about the safety system component. This is followed by an illustration EMV2-based architecture fault model specification for a dual redundant flight guidance system and a discussion of the automation of safety analysis methods in use by the SAE ARP4761 safety practice. Finally, we discuss the value of automation of safety analysis based on EMV2 architecture fault models.

II. THE ARCHITECTURE ANALYSIS AND DESIGN LANGUAGE

The Architecture Analysis and Design Language (AADL) [13] is a modeling language standardized by SAE International. It defines a notation for describing system concerns and its interaction with its operating environment (i.e., processors, bus, devices).

The core language specifies several categories of components with well-defined semantics. For each component the modeler defines a component type to represent its external interface, and one or more component implementations to represent a blue print in terms of subcomponents. For example, the task and communication architecture of the embedded software is modeled with thread and process components interconnected with port connections, shared data access and remote service call. The hardware platform is modeled as an interconnected set of processor, bus, and memory components, with virtual processor representing partitions and hierarchical schedulers, and virtual bus representing virtual channels and protocol layers. A device component represents a physical subsystem with both logical and physical interfaces to the embedded software system and its hardware platform. The system component is used to organize the architecture into a multi-level hierarchy. Users model the dynamics of the architecture in terms of operational modes and different runtime configurations through the mode concept. Users further characterize components through standardized properties, e.g., by specifying the period, deadline, worst-case execution time for threads.

The language is extensible; users may adapt it to their needs using two mechanisms:

1) User-defined properties. New properties can be defined by users to extend the characteristics of the component. This is a convenient way to add specific architecture criteria into the model (for example, criticality of a subprogram or task)
2) Annex languages. Specialized languages [14] can be attached to AADL components to augment the component description and specify additional characteristics and requirements (for example, specifying the component behavior [14] by attaching a state-machine). They are referred to as annex languages, meaning that they are added as an additional piece of the component. In this paper we will discuss the Error Model Annex language.

AADL provides two views to represent models:

1) The graphical view outlines components hierarchy and dependencies (bindings, connection, bus access, etc.). While it does not provide all details, this view is very useful when using the architecture for communication and documentation purposes.
2) The textual view shows the complete model description, with component interfaces, properties and language annexes. It is appropriate for users to capture system internals details and for tools to process and analyzes the system architecture from models.

![AADL Ecosystem](image)

Fig. 1. AADL Ecosystem for Software System Design and Implementation

The AADL model, annotated with properties and annex language clauses is the basis for analysis of functional and non-functional properties along multiple dimensions from the same source, and for generating implementations, as shown in figure 1. AADL has already been successfully used to validate several quality attributes such as Security [15], [1], Performance or Latency [16]. Supporting analysis functions have been developed in the Open Source AADL Tool Environment (OSATE) [17], an AADL tool for architecture design and analysis.

III. THE ERROR-MODEL ANNEX

EMV2 supports architecture fault modeling at three levels of abstraction:

• Focus on error propagation between system components and with the environment: Modeling of fault sources in a system and their impact on other components or the operational environment through propagation. It allows for safety analysis in the form of hazard identification, fault impact analysis, and stochastic fault analysis.
• **Focus on component faults, failure modes, and fault handling**: Modeling of fault occurrences within a component, resulting fault behavior in terms of failure modes, effects on other components, the effect of incoming propagations on the component, and the ability of the component to recover or be repaired. It allows for modeling of system degradation and fail-stop behavior, specification of redundancy and recovery strategies providing an abstract error behavior specification of a system without requiring the presence of subsystem specifications.

• **Focus on compositional abstraction of system error behavior in terms of its subsystems**: It allows for scalable compositional safety analysis.

In addition, EMV2 introduces the concept of error type to characterize faults, failures, and propagations. It includes a set of predefined error types as starting point for systematic identification of different types of error propagations providing an error propagation ontology. Users can adapt and extend this ontology to specific domains.

**Focus on error propagation**: For each component the user can specify outgoing and incoming error propagations of error types being propagated and of error types expected to be contained. The error propagation paths between components are determined by connections and deployment bindings. In addition, each component includes a specification of whether it is the source of an error propagation, the sink of an error propagation, or a path that passes on incoming error propagations, possibly transforming the error type into a different one.

Error sources, propagations, paths, and sinks are similar to the Fault Propagation and Transformation Calculus (FPTC) [18]. When annotated with properties that indicate a hazard characterization they are the basis for FHA and FMEA.

<table>
<thead>
<tr>
<th>error behavior</th>
<th>Simple</th>
</tr>
</thead>
<tbody>
<tr>
<td>events</td>
<td></td>
</tr>
<tr>
<td>failure</td>
<td>error event</td>
</tr>
<tr>
<td>recov</td>
<td>error event</td>
</tr>
<tr>
<td>states</td>
<td></td>
</tr>
<tr>
<td>Operational</td>
<td>initial state</td>
</tr>
<tr>
<td>Failed</td>
<td>state</td>
</tr>
<tr>
<td>transitions</td>
<td></td>
</tr>
<tr>
<td>t1</td>
<td>Operational $\rightarrow$ Failed</td>
</tr>
<tr>
<td>t2</td>
<td>Failed $\rightarrow$ recov $\rightarrow$ Operational</td>
</tr>
<tr>
<td>end behavior</td>
<td></td>
</tr>
</tbody>
</table>

Listing 1. Error Behavior State Machine

**Focus on component error behavior**: For each component the user can specify an error event, i.e., activation of component-specific faults, recover and repair events, their occurrence probability, how they together with incoming error propagations affect the error state of the component, under what conditions outgoing error propagations occur, and when error behavior is detected and addressed by the component.

Error behavior is specified by reusable error behavior state machines. Listing 1 illustrates this for a state machine with an Operational and a Failed state. An error event triggers transition to the Failed state, while a recover event triggers transition back to Operational.

Component error behavior is specified by

• identifying an error behavior state machine from a reusable an error model library,
• specifying component-specific transitions in terms of component-specific error event types and in terms of how the component responds to incoming error propagations from other components,
• specifying conditions under which an outgoing error propagation occurs, i.e., whether and when error states become observable by other components, and in which error states incoming propagations are masked (the component becomes a sink) or passed on to other components in the same form or recognized and converted into a different error type (such as an incoming out of range value results in a skipped output,
• specifying conditions under which an error state or error propagation is detected by the actual system and which component is expected to perform the detection, reporting, and appropriate recovery. The completion of recovery by the actual system is mapped back into EMV2 via the recover and repair events.

In addition, users can specify the fact that the interaction between components, i.e., connections, can be an error source, and how errors propagated from a network that the connection is bound to manifests itself for the recipient.

**Focus on the compositional error behavior**: For each component with subcomponents the user can specify under what conditions in terms of subcomponent error states the component is in a particular error state. This mapping of subcomponent error state into a component error state abstraction reflects fault tree logic and allows for architecture fault analysis at different levels of the architecture hierarchy. This notation is the basis for FTA.

Listing 2 shows an example system that operates with two sensors. It remains operational as long as one sensor is operational, but fails when both sensors have failed.

<table>
<thead>
<tr>
<th>composite error behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>states</td>
</tr>
<tr>
<td>[sensor1.Failed and sensor2.Failed] $\rightarrow$ Failed ;</td>
</tr>
<tr>
<td>[sensor1.Operational or sensor2.Operational] $\rightarrow$ Operational ; end composite ;</td>
</tr>
</tbody>
</table>

Listing 2. Example of a Composite Error Behavior

The composite error behavior specification must be consistent with the component error behavior specification for the same component. For that reason, our tool includes a consistency checker to ensure that both specifications (composite error behavior and component error behavior) are consistent. The more abstract component error behavior specification is referenced when specifying the composite error behavior of the enclosing system.
This allows us to specify and analyze error behavior one architecture hierarchy level at a time.

**Error types and error propagation ontology:** The error type system allows users to efficiently characterize different types of error events, error propagations, and error states. Users can define their own error type libraries, or use a predeclared set of error types that represents a common error propagation ontology. Error types can be placed into type hierarchies. Figure 2 shows `TimingError` to be defined as consisting of `LateDelivery` and `EarlyDelivery`, but they cannot occur at the same time.

Error types can also be grouped into type set as illustrated in Listing 3. In this example, we see that errors can be of type `TimingError`, `InvalidValue`, or a combination of the two error types. For example, a sensor may send an invalid sensor reading and send it late.

```
Mset: type set {TimingError, InvalidValue, TimingError + InvalidValue};
```

Listing 3. Error-Type Set Example

The Error Model Annex includes a standard set of error types to represent an ontology of commonly propagated effects. The ontology draws on previous work on formally specifying error propagation behavior [11], [12]. The ontology helps modelers to ensure that they have considered different types of error propagation in the interaction between system components. The ontology views the interaction between two components as one component providing a service to the other with the service consisting of a sequence of service items, e.g., a sensor provides a stream of sensor readings. The ontology includes the following error types:

- **Omission and commission errors** in the service provided by a component (ServiceOmission, ServiceCommission) or of individual service items (ItemOmission, ItemCommission) to represent loss or unexpected provision of electrical power, loss of a message or command, failure to provide sensor readings,

- **Timing errors** in the form of individual messages arriving early or late (EarlyDelivery, LateDelivery), a sequence of sensor readings being sent at the wrong rate (HighRate, LowRate, RateJitter), or a name lookup service starting up too early or late (EarlyService, DelayedService),

- **Value errors** in individual service items, such asOutOfRange sensor readings,OutOfRange control state, or errors in a sequence of values such as BoundedValueChange, e.g., difference between two successive setpoint values beyond an acceptable delta, StuckValue, NonMonotonic, or OutOfCalibration,

- **Replication errors** dealing with replicates of service or states being communicated, such as AsymmetricValue error in one of the replicated communication channels due to data corruption, AsymmetricTiming error when one of the replicated sensors produces the sensor reading to late, or AsymmetricOmission error when one of the replicates fails, and

- **Concurrency errors** when accessing shared logical or physical resources, such as RaceCondition or MutexError resulting in inconsistencies of shared data between several concurrent tasks.

Modelers can define aliases for these error types, e.g. `NoPower` as alias for `ServiceOmission`, and extend the ontology, e.g., by introducing a non-monotonic value change error on sequences. Modelers also use the error type system to introduce domain specific error types to characterize different types of faults in a component. For example, error types to characterize errors in software components, such as stack overflow, array out of bound, or divide by zero, or failures in mechanical components, such as stuck value, missed reading, corrupt reading of a sensor.

While this section provides an overview of the main EMV2 concepts, the complete specification of the EMV2 language [19] with its syntax is available as an annex document of the AADL standard [13]. Interested readers can get it through the official SAE website.

**IV. AN ARCHITECTURE FAULT MODEL EXAMPLE**

In this section we illustrate architecture fault modeling with EMV2 at three levels of abstraction with an example system, a dual redundant flight guidance system (FGS). FGS consists of a flight guidance (FG) component, an auto pilot (AP) component, and an actuator subsystem (AC). The FG and AP are dual redundant pairs as illustrated in figure 3. We proceed by first focusing on error propagation between system components, then by introducing error state to the components and specifying compositional fault behavior to reflect the expected fault tree logic, and finally by elaborating the error behavior of each component with the response of each component to its failures and incoming propagations reflecting expected fault management by a safety system component of the system.
A. Error Source and Propagation Specification

First we specify the error sources and propagations for each of the components in FGS. We do this as EMV2 subclause in the component type of the FG, AP, and AC components. In our example, FG is a source of NoData due to an internal failure, declared as error source for the outgoing error propagation NoValue on OutPort. (shown in listing 4). This also shows a component error behavior declaration associating the Failed state with the outgoing propagation - consistent with the error source declaration.

Listing 5. Textual declaration of error propagations for AC

```plaintext
system FG
features
  InPort: in data port;
  OutPort: out data port;
annex emv2 {**
  use types ErrorModelLibrary;
  use behavior ErrorModelLibrary::Simple:
error propagations
  OutPort: out propagation {NoValue};
flows
    FGGFail: error source outport{NoValue};
end propagations;

component error behavior
propositions
  Failed [0] -> Outport(NoValue);
end component;
**};
end FG;
```

AP is both a source of NoValue due to failure and passes on incoming NoValue from the FG as indicated by the error path declaration. In addition, AP shows an incoming propagation of BadValue. This accommodates potential data corruption in the communication between FG and AP. These specifications are shown in listing 5. The propagation specification of AC is slightly more complicated in that it passes on incoming NoValue from both APs.

Listing 6. Compositional Error Behavior Specification

```plaintext
system AP
features
  InPort: in data port;
  OutPort: out data port;
annex emv2 {**
  use types ErrorModelLibrary;
  use behavior ErrorModelLibrary::Simple:
error propagations
  InPort: in propagation {NoValue, BadValue};
  OutPort: out propagation {NoValue, BadValue};
flows
  APFail: error source outport{NoValue};
  ThruFGFail: error path import{NoValue} -> outport{NoValue};
  ThruFGBad: error path import{BadValue} -> outport{BadValue};
end propagations;
**};
end AP;
```

These specification automatically apply to all instances of each component type. In our FGS example they apply to the two instances of FG and AP, as well as the single instance of AC as illustrated in figure 4. The outgoing NoValue propagation from AC becomes an outgoing propagation for

B. Compositional Error Behavior Specification

The purpose of the compositional error behavior specification is to define a mapping of error states of sub-components of a system into error states of the system itself. For that purpose we associate an error behavior state machine with each sub-component through the use behavior declaration (see listing 5). We will use the Simple state machine defined in listing 1.

For FGS we have defined a separate error state machine with three states that distinguishes between a single channel failure and failure of both channels.

Listing 7. Compositional Error Behavior Specification

```plaintext
composite error behavior
states
  [AP. Operational and AP2. Operational and
  FG1. Operational and FG2. Operational and
  AC. Operational] -> Operational;

  [AC. Operational and 1 ormore
  (FG1. Failed , AP1. Failed) and
  FG2. Operational and AP2. Operational or
  1 ormore (FG2. Failed , AP2. Failed) and

  [in powersupply[NoPower] or AC. Failed or
  1 ormore (AP1. Failed , FG1. Failed) and
  1 ormore (AP2. Failed , FG2. Failed)] -> TwoChannelFailure;
end composite;
```

The error source, path, and sink specifications corresponds to the Fault Propagation and Transformation Calculus (FPTC) of York University. When combined with the error propagation paths inherent in the AADL model, namely, any port, shared data, subprogram call, or bus access connection, as well as any binding relationships, such as thread to processor bindings, data and code to memory bindings, and connection to bus, processor, and device bindings, they represent a fault propagation dependency graph that is the basis for fault impact analysis, such as Failure Mode and Effect Analysis (FMEA) or Common Cause Analysis (CCA). Note that EMV2 allows users to declare propagation paths that are not explicit found in the AADL model, e.g., the propagation of heat between two processor closely co-located on the same board.
external power supply on the FGS. The logical expressions used in that mapping effectively represent the logic in a fault tree. In other words, such a specification is typically produced when engineers make initial decisions regarding redundancy strategies to compensate for failures in system components.

This compositional abstraction of error behavior plays two important roles. First, it provides a specification of expected redundancy management that must be satisfied by the health monitoring/fault management capabilities of the system. The expected realization of this health monitoring/fault management capability will be expressed using component error behavior specifications discussed in the next section.

Second, it allows for scalable solutions for reliability and availability analysis involving multiple levels of a system architecture hierarchy. Reliability and availability figures are associated with the Operational and error states of the component error behavior specification at each level of the architecture hierarchy. The values of the lower level components are aggregated according to the compositional logic expressions directly (stochastic fault tree analysis, reliability block diagram analysis) or based on Markov Chains or Stochastic Petri Nets. Such an analysis would otherwise not be feasible for large-scale systems, in particular when the size of Stochastic Petri nets or Markov Chains grows quickly with increasing number of system components.

C. Component Error Behavior Specification

The component error behavior specification for a component allows the modeler to what kind of component internal failures and anomalous conditions result in failure models that may impact other components. For example, the autopilot may encounter multiple types of failures that manifest themselves in a failure mode Failed with the effect that no data is provided as output. Each of those is represented by a separate error event declaration, or by one error event declaration using a set of error types to distinguish different types of failures.

Error states are used to represent the failure modes, again modeled as separate states, or as a state with different error types. In other words, the use of error types is not required for error events and error states, but allows for a more compact specification of error behavior, in particular when combinations of error types are involved.

The component error behavior specification of AC illustrates how to deal with redundant input from two sources. Listing 7 shows the component error behavior specification. The Failed error state results in a NoValue out propagation. NoValue on both incoming ports results in a NoValue outgoing propagation. An incoming NoValue propagation on one port with NoError (a built-in keyword to indicate no propagation) on the other port results in good output that cannot be confirmed by replicate comparison. This is indicated by the error type SingleSource. An incoming BadValue propagation on one port with NoError on the other port results in good output that can be detected by replicate comparison, but not corrected since we do not know which one is the correct value. Such a case is either treated as NoValue or the previous value is used as approximation. Finally, BadValue on both incoming ports is only detectable if they represent two separate corruptions with different bad values. Effectively, we have specified an expected fault management strategy to be provided by a health monitoring component of FGS, in our example functionality embedded within AC.

Note that the EMV2 consistency checker will inform us that the original abstract propagation specification for FGS has to be refined to include BadValue and SingleSource as types. Note also that incoming NoValue error propagations to AP do not trigger a transition of AP to the Failed error state. We have chosen to do so because the component itself has not failed, but responds to faulty or missing input.

The EMV2 modeling tool checks for consistency between the component error behavior specifications and the error propagation specification of the same component. Furthermore, the EMV2 analysis capability takes into account the explicitly specified redundancy strategy and calculate appropriate reliability and availability number for critical and non-critical operational modes of FGS based on occurrence probabilities of component internal error events and occurrence probabilities of incoming error propagations from external sources using Reliability Block Diagram or Markov Chain based analysis.

Note that the actual system, as specified in the core AADL model, must be refined to include a safety system component that implements the fault management capability assumed by the EMV2 specification and resulting safety analyses.

D. External Error Sources

In our example, all subcomponents of FGS draw from the same power source. Any loss of externally supplied power results in the failure of all subcomponents to operate. Therefore, we have chosen to reflect that error behavior at the FGS level rather than passing the error propagation of NoPower to each subcomponent and have it respond by transitioning into Failed state. We specify a transition to a Failed state and a return to an Operational state when incoming data is error free (i.e., the incoming propagation indicates NoError). The effect of the external power supply on FGS is also reflected in the composite error behavior specification. As listing 6 shows power loss contributes to the TwoChannelFailure error state of FGS. This reduces the size of auto-generated fault trees, Markov Chains, or Petri nets.

<table>
<thead>
<tr>
<th>component error behavior propagations</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1 : Failed -&gt; NoError</td>
</tr>
<tr>
<td>p2 : Operational FromAP1Port NoError</td>
</tr>
<tr>
<td>p3 : Operational FromAP2Port NoError</td>
</tr>
<tr>
<td>p4 : Operational FromAP1Port NoError</td>
</tr>
<tr>
<td>p5 : Operational FromAP2Port NoError</td>
</tr>
<tr>
<td>p6 : Operational FromAP1Port BadValue</td>
</tr>
<tr>
<td>end_component</td>
</tr>
</tbody>
</table>

Listing 7. AC Component Behavior
V. AUTOMATED SAFETY ANALYSIS

We have implemented a number of safety analysis tools that process the architecture fault model, i.e., the core AADL models enhanced with EMV2 annotations. They produce reports to support safety and reliability evaluation process of the ARP4761 standard [3] (for the Preliminary System Safety Assessment - PSSA - and the System Safety Assessment - SSA). Table I shows the artifacts from the Error-Model Annex of particular importance to generate each certification materials. It also shows that fewer documents are required by the PSSA than for the SSA. Also, the FTA can be used during both phases (PSSA and SSA) and when generated from a detailed architecture, the generated document will when include an enhanced tree with a wider depth, showing more potential error events.

<table>
<thead>
<tr>
<th>Document</th>
<th>Phase</th>
<th>EMV2 Artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td>FHA</td>
<td>SSA</td>
<td>error source, error events</td>
</tr>
<tr>
<td>FTA</td>
<td>PSSA</td>
<td>error composite, error events</td>
</tr>
<tr>
<td>MA</td>
<td>SSA</td>
<td>error source, component error behavior, error probabilities, error events, error events</td>
</tr>
<tr>
<td>FMEA</td>
<td>SSA</td>
<td>error source, error path, error sink</td>
</tr>
</tbody>
</table>

TABLE I. ASSOCIATION OF MODELING ARTIFACTS WITH CERTIFICATION DOCUMENTS

The automatic production of certification documents is a feature built-in in the Open Source AADL Toolkit Environment (OSATE) [17], our Eclipse-based AADL modeling framework. It is freely available under an Open-Source license (the Eclipse Public License) and the safety analysis tools can be interfaced with Open-Source tools as well (as OpenFTA [20] for the FTA and PRISM [21] for Formal Analysis methods).

A. Functional Hazard Assessment (FHA)

The Functional Hazard Assessment (FHA) document is described in section 3 of the ARP4761 standard [3] and consists in an examination of system functions and a list of all potential failure. It identifies and classifies failure conditions according to their severity. For each identify failure, the FHA report would report design constraints, annunciation of failure condition and other relevant information.

In terms of implementation, this report is a spreadsheet that enumerates faults/failure, its potential contributors and their associated information (description, condition, operational phases, effects, etc.).

Generating the FHA from the AADL model is done by processing the model and extracting information (properties) related to elements that may generate an error (error event, error propagation, etc.). Then, the tool retrieves relevant association and builds a document summarizing and constituting the FHA. This is actually implemented by a generator of excel spreadsheets, each row representing a fault or an error propagation.

B. Fault Tree Analysis (FTA)

The ARP4761 standard [3] describes the FTA as a failure analysis that focuses on one particular undesired event and provides a method for determining its causes. The FTA shows the hierarchical errors occurrences that lead to a top event. For example, the FTA for the loss of portable and self-powered device can be the loss of power (an error event) that can be decomposed into other error events such as loss of primary and redundant power sources (e.g. batteries).

Our tool interprets the composite error behavior specification to automatically generate the fault tree from a given state to generate a fault tree representation. Given a specific error state of a component, the tool analyzes all contributors and adds them into the tree. The generated fault tree can be imported into Open-Source tools such as OpenFTA [20] or commercial/proprietary programs such as CAFTA [22].

C. Failure Mode and Effect Analysis (FMEA)

The ARP4761 standard [3] describes the FMEA as a systematic, bottom-up method of identifying the failure modes of a system, item or function and determining the effects on the next higher level. An FMEA address failure effects resulting from single failures on other parts of the system.

For each error, an FMEA identify the failing component (e.g. a battery) and its effect (e.g no power on connected the electronic units). It might also include other information such as failure rates, detectability of phase of flight in which the failure occurs.

To generate the FMEA from the AADL model, our tool retrieves all error source and error event and captures error flows in the architecture model. To do so, it browses all connections from the flow source, records all error path to the final error sink that handles the error. It captures this information in a list that can be viewed using Commercial or Open-Source productivity tools.

D. Benefits of Safety Analysis Automation

Current safety analysis practice is labor-intensive. For example, FMEA is often limited to single faults, and explores effects only one/two levels deep. Analysis reports are examined by a few reviewers and not repeated until years later. Previous experiences [10] using AADL and the original Error Model annotations on safety analysis of satellite systems explored as many as 26,000 failure modes, and effects as deep as 25 levels.

Automation of safety analysis allows for:

- early identification of potential problems, including single points of failure, the effects of multiple failures, and unexpected failures.
- exploration of architecture alternatives from a safety perspective with limited additional effort.
- frequent re-analysis as the architectural design evolves,
- extension of safety analysis into the software system architecture.

The result is increased confidence that the safety analysis results correctly reflect the actual system implementation.
VI. CONCLUSION AND PERSPECTIVES

Safety-Critical Systems operate functions where a single fault might have catastrophic consequences. For that reason, they must be analyzed to make sure that potential hazards are a threat. However, because systems are growing and involve development teams operating in various engineering domains, finding the root cause of an error is difficult. Architectures must be evaluated at the earliest to check that planned implementation will enforce its requirement. To do so, we need to integrate safety information into the architecture description.

In this paper, we have presented an architecture fault modeling approach that supports the automation of safety analysis. This approach annotates the SAE Architecture Analysis and Design Language (AADL) with error behavior specifications expressed in EMV2, a revision of the Error Model Annex standard that is currently in ballot. This revision has improved expressive power and semantics and includes a fault propagation ontology. We have illustrated the capability of specifying error behavior at three levels of abstraction on a dual redundant flight guidance system. We have also described different safety analyses in support of the SAE ARP4761 recommended system safety assessment practice and summarized the benefit of automating these analyses.

We are also investigating how this safety-specific extension interplay with other AADL sub-languages (such as the behavior annex [14]). As AADL provides mechanisms to augment the architecture description (now including safety and behavior), making analysis tools aware of these extensions would then improve analysis techniques.

ACKNOWLEDGMENT

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This material has been approved for public release and unlimited distribution.


DM-0001265

REFERENCES