ABSTRACT

Cyber-physical systems, used in domains such as avionics or medical devices, perform safety-critical functions where a fault might have catastrophic consequences (mission failure, severe injuries, etc.). Their development is guided by rigorous practice standards to avoid any error. However, as more software-based functions are integrated into a system, interaction complexity has increased significantly over the years. While software appears to ease upgrades and adaptation, interaction complexity, e.g., due to shared hardware resources, has resulted in high error leakage to system integration. Late discovery of errors introduced in requirements and architecture design have resulted in costly rework, making up as much as 70% of the total software system cost.

To overcome these issues, architecture-centric model-based approaches abstract system concerns into analyzable architecture models. These models are then analyzed to spot and detect errors, issues or defects that are usually detected lately in the development process (likely testing or operational phases) and incur a costly rework and re-engineering efforts. This predictive analysis approach is often used for time-related performance criteria, such as schedulability and latency.

Despite their importance, safety and reliability criteria are still investigated by system engineers in a labor-intensive process and are often not revisited later in development. Assumptions made during such early analysis may be violated during design and implementation and may ignore fault contributors that are due to software design and coding errors.

To address this issue, we have added an error behavior annotation to SAE AADL, an international language standard for modeling embedded software system architectures that captures the interaction between software, hardware and the physical system in a single notation. We have added tools to process the enhanced notation in support of safety and reliability practice standards, such as SAE ARP4761. By automating system analysis and generating adequate documentation, we show how we can assist engineers in validating system architecture and make the safety/reliability evaluation process repeatable and less error-prone.

Categories and Subject Descriptors

C.4 [Performance of Systems]: Design Studies, Modeling techniques, Reliability, availability and serviceability; D.2.1 [Software Engineering]: Requirements/Specifications—Languages, Tools, Methodologies; D.2.4 [Software Engineering]: Software/Program Verification—Reliability, Validation; D.2.11 [Software Engineering]: Software Architectures—Languages, Domain-Specific Architectures

General Terms

Design, Verification, Reliability, Languages, Standardization

Keywords

AADL, error, ARP4761, reliability, Fault-Tree Analysis, Fault Hazard Assessment, Markov Chain

1. INTRODUCTION

1.1 Context

Cyber-physical systems perform critical functions under constrained and potentially hostile circumstances. Because an error or a failure can have catastrophic consequences [12], they must be designed carefully and validated/certified according to a rigorous process to prove assurance of correct operation and increase confidence of system design. Verification criteria depend on applications criticality: the most demanding standards require validating and inspecting software code to prove evidence of behavior correctness.

However, more operations are now implemented with software [12], which has many advantages: easier upgrade and customization, affordability for building and ease of adaptation to particular needs. However, this trend increases the number and interaction complexity of software components collocated on the same networked execution platform. Such complexity resulted in a new set of interaction challenges due to software-induced fault root causes that are difficult to test for [5]. For example, one non-critical component may overrun its deadline so that critical function does not have enough computing resource to complete their task.

1.2 Current Problem

Safety and reliability practice standards, such as ARP4761 [21], MILSTD882 [26], and Do-178B/C [19], provide guidance for validation and verification of safety-critical software and systems. As shown in figure 1, this process is done manually and mostly relies on experience with previous systems and the ability of engineers to correctly interpret a textual specification of the system.

In addition, safety-related errors/defects are likely not detected until integration testing/operational phases although most of them
Architecture models expressed in the SAE International Architecture Analysis and Design Language (AADL) standard [23] and its Error Model Annex standard [24]. These annotations enhance the existing architecture model with information that supports the various safety and reliability practices of the ARP4761 standard from the same architecture model and complement other form of software system analysis as well as software system builds from validated models. This is shown in figure 2.

The paper is organized as follows. We first give an overview of AADL and the ARP4761 [21] standards. Then we introduce the Error Modeling extension of AADL for representing safety concerns. This is followed by a description of the tool support for automating certification practices required by ARP4761 [21]. Finally, we illustrate the application of our approach with a case study.

2. RELATED WORK

Existing literature [12, 1] show evidence that software errors and bugs might have catastrophic consequences. For that reason, software operating critical functions must be carefully designed but also analyzed to avoid and prevent any error and their propagation. Several methods and approaches address this concern: some [21, 26] focus on hazards as result of failure events and their propagations (and evaluate system safety using Fault-Tree Analysis - FTA - or Failure Mode and Effects Analysis - FMEA), while others [11] focus on safety-related constraints that must be satisfied by a system design.

Our contribution in this paper aims to support safety evaluation (as the one [21, 26]) from an architecture model and ensure system consistency across different notation and analysis. To do so, we reuse an existing industry standard architecture language [23] and extend it to support safety evaluation. Similar extensions exist for expressing constraints on architecture models.

2.1 AADL

The Architecture Analysis and Design Language (AADL) [23] is a modeling language standardized by SAE International. It defines a notation for describing embedded software, its deployment on a hardware platform, and its interaction with a physical system within a single and consistent architecture model. 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 plat-
form 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 architectural criteria into the model (for example, criticality of a subprogram or task).

2. **Annex languages.** Specialized languages [22] can be attached to AADL components to augment the component description and specify additional characteristics and requirements (for example, specifying the component behavior [7] 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 languages annexes. It is appropriate for users to capture system internals details and for tools to process and analyze the system architecture from models.

In order to support safety validation, our contribution to the language and its associated toolset are:

- Augmenting the original Error Model Annex language for AADL with safety semantics and a fault ontology to support the modeling of error behaviors.

- Developing new tools that automatically produce safety validation materials from AADL models with Error Model annotations.

Some existing work partially addresses the automation of safety evaluation using the original Error Model Annex standard (for example by generating failure effect analysis [4] and Fault-Tree Analysis [25, 13]). However, the support addresses a subset of the required documentation. Covering other needs is a new challenge and requires appropriate semantics and associated tools.

The next section presents the ARP4761 [21] safety standard, detailing the necessary constructs to add in AADL models for its support.

### 2.2 ARP4761

ARP4761 is a standard recommended practice for evaluating system safety and reliability. It defines a process, identifies applicable methods/approaches and illustrates their use to a case-study related to the avionics domain. The process consists of a Functional Hazard Assessment (FHA) of the global system (e.g. the Aircraft). It is the input to a Preliminary System Safety Assessment (PSSA) and System Safety Assessment (SSA) of each sub-system.

The PSSA consists of a FHA and Fault-Tree Analysis (FTA) of the sub-system under investigation. FTA provides a convenient view of system fault hierarchy and dependencies, highlighting the root cause of each failure. Establishing it for each sub-system (for example, loss of power) in relation with the overall system failures (for example, loss of the braking system of the aircraft due a failure of power supply) helps in understanding the impact and propagation of each fault in the architecture. Such a method is actually supported either in commercial [16] or open-source [14] tools.

The SSA is a continuation of the PSSA that provides evidence that metrics, values, and data established and obtained in the PSSA could be verified and validated. To do so, several methods may apply. Among them, the use of formal specification is recommended. It could be used to analyze, simulate and verify system characteristics. This type of analysis has been successfully performed [13] using Open-Source tools such as PRISM [10].

### 3. THE AADL ERROR-MODEL ANNEX

The Error Model Annex is a standardized extension to the core language for adding component safety-related information to AADL models [20]. This annex allows users to specify stochastic error behavior state machines whose transitions are triggered by error events and whose effects are identified as outgoing and incoming error propagations. A first version of the Error-Model Annex was published in 2006, our contribution is a revision that extends its notation and semantics to support safety standards and best practices. The resulting revised Error Model Annex, which is going into ballot, has the following new capabilities:

- support of a mechanism to characterize an error event, state, or propagation with a user definable error type, e.g., the propagation of an out of range value error,

- a predefined set of error types as a common ontology of types of error being propagated.

![Figure 3: AADL Ecosystem for Software System Design and Implementation](image-url)

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 3. AADL has already been successfully used to validate several quality attributes such as Security [9, 5], Performance or Latency [6]. Supporting analysis functions have been designed in the Open Source AADL Tool Environment (OSATE) [3], an Eclipse-based framework.
• separation of error propagation specification and component error behavior specification,

• explicit specification of error types propagated or not propagated out (guarantee) and expected or not expected as incoming propagation (assumption),

• explicit specification of a system error state in terms of the error states of its components,

• descriptive and stochastic properties on error model elements, e.g., to capture hazard descriptions or probability of fault occurrence.

The Error Model Annex language supports architecture fault modeling in several ways:

• **Focus on types of errors**: An error type system that allows the user to characterize fault occurrences, error state and error propagation in a consistent manner. A set of standardized types to characterize error propagations represents a common fault ontology.

• **Focus on error propagation between components**: 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 passes on incoming error propagations, possibly transforming the error type into a different one. This level of architecture fault model specification allows for hazard identification, fault impact analysis, and stochastic fault analysis.

• **Focus on error behavior of a component**: 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.

• **Focus on the composite error behavior of a component**: 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.

The following paragraphs introduce the main concepts of the error-model annex: faults types, error events, propagations points and the error state machine for specifying the component error behavior.

### 3.1 Error Types and Fault Ontology

The error type mechanism identifies and classifies errors into type hierarchies. Types within the same type hierarchy cannot occur at the same time. As shown in figure 4 and listing 1, timing error has two sub-types: late delivery and early delivery. LateDelivery and EarlyDelivery are mutually exclusive.

![Figure 4: Error-Type Hierarchy](image)

**Listing 1: Error-Type Set Example**

```plaintext
Myset: type set {TimingError, InvalidValue, TimingError + InvalidValue};
```

An error type set is used to specify sets of possible error types. Listing 1 shows an error type set of all subtypes of the super-type TimingError, a value error of type InvalidValue, and a product type indicating combinations of both.

These error type set are used to specify possible error event types and error propagation types. Since the error types are part of a type system, type checking of these type sets ensures that any error type being propagated out of a component can be handled by other components this component interacts with.

For example, one component may specify MySet as outgoing error propagations, while another component indicates that it expects only TimingError as incoming error propagation.

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 [17, 18]. The ontology consists of the following hierarchies of error types:

- Omission and commission errors in the service provided by a component or of individual service items (loss of a message or command, unintended incoming data, etc.),
- Timing errors and value errors on individual items being communicated (value transmitted too late/early, outdated data, etc.)
- Rate and sequence errors for streams of service items (e.g., streams of sensor readings, inconsistent value within a data stream, etc.),
- Replication errors in the form of asymmetric value, timing, and omission errors in redundant systems (e.g., redundant systems have inconsistent states or values), and
- Concurrency errors when accessing shared logical or physical resources (e.g., inconsistencies of shared data between several concurrent tasks)

Errors that occur inside a component, e.g., a software component in a fault containment unit such as a protected address space or partition, manifest themselves to other components as error propagation of one of the above error types.

The error types of the fault ontology are defined in an error type library. Modelers can extend this set of error types, and define aliases, e.g. NoPower as alias for ServiceOmission. Modelers can also define their own error type hierarchies, for example, error types to characterize error events to characterize errors in software components, such as stack overflow, array out of bound, or divide by zero.
3.2 Component Error Events and Propagation

The annex introduces the concept of error event that represents an internal error occurring within a component when a fault is activated. This event may propagate to the other components along error propagation paths. These are the connections between the components and the deployment bindings between software and hardware components. Note that users can also specify recover events to model the ability of a component to return to a working condition due to fault management and repair events to model the result of a repair activity that involves replacing system parts.

Error propagations are specified for incoming and outgoing component features, such as ports and access features, as well as for bindings. Outgoing propagations specify the error types that are expected to be propagated out and error types that are expected to be contained by the component. Incoming propagation specifications indicate the types of errors that a component is willing to accept and those that it expects not to be propagated.

In addition the modeler can specify error flows for components. An error flow is an error source (a component internal error event results in a propagation), an error sink (the received error propagation is contained or masked by the component), or an error path (the component passes the error through as an outgoing propagation or it may transform it into a different propagated error type).

The error events, propagations, and flows can have properties that indicate a hazard characterization and a probability of occurrence. This provides a basis for early safety analysis similar to the Fault Propagation and Transformation Calculus (FPTC) [17].

3.3 Component Error Behavior

The error behavior of an individual component is characterized by an error behavior state machine. Such state machines can be defined as reusable items in an error model library.

A component state defines a particular state of the component regarding its error behavior. A basic state machine would contain two states: Operational (active when the component is operating without any error) and Failed (active once an error is triggered). A transition defines the condition under which a state change occurs. It is composed of a source state (the initial state of the component), a destination state (the final state after the transition is triggered) and a condition (error events that need to be triggered/activated to activate the transition). Using or previous example, two transitions could be added:

1. One from Operational to Failure triggered when the Failure error event is activated.
2. One from Failure to Operational triggered when the Recover event is activated.

Listing 2 illustrates the textual declaration of the state machine and figure 5 shows its corresponding graphical representation.

```
error behavior Simple
states
  Operational: initial state;
  Failed: state;
transitions
  t1: Operational -> Failed;
  t2: Failed -> Operational;
end behavior;
```

Component error behavior is specified by:

- identifying an error behavior state machine in an error model library,
- specifying component-specific transitions in terms of incoming error propagations,
- specifying conditions under which an outgoing error propagation occurs,
- specifying conditions under which an error state or error propagation is detected by the actual system.

This is done in an annex subclause declared inside a component type or component implementation. These declarations specify possible mappings of error types from an error event or incoming propagation to a resulting error type of a transition destination state, the error type of an outgoing propagation, or the error code used by the actual system to report a detected error condition.

3.4 Composite Error Behavior

The error behavior of a system component can also be specified in terms of the error behavior of its parts. For example, if a coffee machine is in the Failed mode when one of its sub-part (the boiler or the filtering system) is Failing. The Error Model Annex language supports this through composite error behavior specifications.

```
composite error behavior
states
  [sensor1. Failed and sensor2. Failed] -> Failed;
  [sensor1. Operational or sensor2. Operational] -> Operational;
end composite;
```

For example, if a system contains two redundant sensors, the main system will be in the failure error state if both sensors are failing. Otherwise, it will still be in the operational state. Listing 3 shows how to specify such a state machine using the textual description of the language.

The composite error behavior specification must be consistent with the component error behavior specification for the same component. 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.
3.5 Predefined Error Properties

The Error Model Annex introduces properties to capture error-specific characteristics. It utilizes the property mechanism of the core AADL language for that purpose. This means that modelers can define additional Error Model specific properties beyond those predefined in the standard document.

In particular, the following properties are of interest in this paper:

- **Hazards**: contains several fields describing the fault characteristics: failure and effect descriptions, severity, likelihood, operational phase, environment, risk, comments, etc. This property allows multiple hazard characterizations to be associated with an error source, outgoing propagation, error state, and can be different for specific error types. This property is processed to produce the FHA report.

- **OccurrenceDistribution**: specifies the distribution method used to compute the error event distribution (fixed, exponential, etc.) and its associated parameters (occurrence rate, probability, etc.). This property is used to produce the FTA and export the model into formal notation (such as Markov Chain).

4. **SUPPORTING THE SAFETY EVALUATION PROCESS**

Our safety analysis tools process the architecture fault model, i.e., the core AADL models enhanced with Error Model clauses. They produce materials and documentation to support safety and reliability evaluation process of the ARP4761 standard (for the Preliminary System Safety Assessment - PSSA - and the System Safety Assessment - SSA):

- **Fault Hazard Assessment (FHA)**: a spreadsheet document list and document all potential errors that may occur in the architecture

- **Fault-Tree Analysis (FTA)**: a hierarchical (tree) view of error propagations dependencies in the system (showing that are the conditions for a fault to occur)

- **Formal Methods with Markov Analysis (MA)**: a mapping to a specific notation that is amenable to validate and verify system safety properties (for example, that failure probability of a component). For this purpose, we export the AADL notation into a Markov Chain model.

- **Failure Mode and Effects Analysis (FMEA)**: a document that show all error paths within the architecture (how an error within component may impact the others). All these documents support described in the ARP4761.

    Also, all these functionalities are built-in in the Open Source AADL Tool Kit Environment (OSATE) [3], 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 [14] for the FTA and PRISM [10] for Formal Analysis methods). Next sections present the FHA, FTA and Markov Chains functions.

4.1 Functional Hazard Assessment (FHA)

The Functional Hazard Assessment (FHA) document 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 is a document such as 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 can then be achieved 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, as shown in figure 9.

4.2 Fault Tree Analysis (FTA)

The ARP4761 standard 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 [14] or commercial/proprietary programs such as CAFTA [16].

4.3 Stochastic Analysis

Stochastic analysis and simulation of a system is performed on Markov Chain models or stochastic Petri nets to predict reliability and availability of a system or system function. Such models are normally created manually based on an architecture design document. In our case, a tool translates the AADL model and its error-model specification into a Markov Chain model. This automated export ensures that the stochastic analysis is performed on a system model that is consistent with the architecture specification, in our case expressed as an AADL model. Then, engineers have to write verification formula and method to check system correctness.

Our tool uses error events, error propagations and error behavior state machines to produce the Markov Chain model. The OccurrenceDistribution property provides the occurrence probability value for an error event or error propagation. In such a translation, each AADL component and its associated error behavior state machine maps into a Markov Chain module. These modules are interconnected according to the architecture description: if two AADL components have a port connection, the corresponding Markov Chains modules will be also inter-connected. At this time our tool generates a Markov Chain representation that can be processed by PRISM [10], an Open-Source Model Checker that has been successfully used for simulating critical systems [13].

5. **EXAMPLE**

We apply the outlined approach to a case study that is a typical industrial example. This is an adaptation of the embedded control example referenced in the documentation of the PRISM Model Checker [10]. It contains several realistic reliability and safety specification that can be used to demonstrate our approach. The following sections describe its translation into AADL and the use of our tools to support ARP4761 safety process.
5.1 Overview and System Description

The system is composed of three sensors, three processors connected by a bus and two actuators. The three sensors are operated by the same processor (input processor), which retrieves data and sends it to the main processor through the bus. The main processor does some computation on the data and sends the results to the output processor (through the same bus) that operates the actuator devices. The high-level architecture is shown in figure 6.

The system specification lists the following potential errors:

- Sensors have a permanent failure every month
- Actuators have a permanent failure every two months
- Processors have two kind of failures:
  - A permanent failure every year
  - A transient failure every day, which is recovered in about 30 seconds

Finally, the overall system is considered in the Failed state if one of the following conditions occurs:

- One processor is in the Failed mode (permanent fault)
- Both actuators are in the Failed mode
- Two sensors are in the Failed mode

5.2 AADL Model

This architecture is then translated into an AADL model using the following components:

- Processors are mapped into AADL processor components.
- AADL processor components are connected through a shared bus represented by an AADL bus component using AADL bus access.
- Sensors are mapped into AADL device components. They send data to the processors using a dedicated bus (such as a PWM or a serial bus) – mapped into a required bus access feature.
- Actuators are mapped into AADL device components and receive data from the output processor using the same bus (PWM or serial bus) – mapped into a required bus access feature.

The main system aggregates all these components altogether to represent the hierarchical architecture. The graphical representation is shown in figure 7.

5.3 Architecture Fault Model

We add error information into the architecture model using the error-model annex to the sensors, actuators, and processors. For the sensors and actuators, we define a basic state machine with two states (Operational and Failed) and one error event (Failure). When the Failure error event is triggered, the component switches from Operational to Failed. Once in the Failed state, it cannot recover as this error is permanent. We associate the state machine with the AADL device type for the sensors and the actuators. We also associate the property OccurrenceDistribution with the Failure event in order to match error occurrence specifications (considering that the time granularity is the second unit): 1.97e-7 for the actuator (one fault every two months) and 3.85e-7 for the sensor (one fault every month). This component error behavior applies to every sensor instance and actuator instance of these two AADL device types.

For the processors, we specify a state machine with three states (Operational, Failed and TransientFailure) and three error events (Failure, FailureTransient and ResetEvent). The corresponding state machine is shown in figure 8:

- The FailureTransient error event represents the occurrence of a transient event with the component switching to the TransientFailure state.
- The recovery of a transient failure is represented by the ResetEvent error event that switches the component from the TransientFailure to the Operational state.
- When the Failure error event is triggered, the component switches to the Failed state. As this is a permanent error, it never recovers from it.

As for the sensor and actuator components, we associate the OccurrenceDistribution property with the following elements:

- Failure error event with a value of 3.17e-8 (one permanent fault per year)
- ResetEvent error event with a value of 0.03 (recover of a transient fault within 30 seconds)
- FailureTransient error event with a value of 1.15e-5 (one transient fault per day)

The error behavior of the top-level system is specified using a state machine with two states: Operational and Failed. The error state is specified from the state of sub-components using a composite error behavior specification that specifies the active state of the system in terms of the states of its subcomponents, as shown in figure 4.

Finally, we associate the Hazard, Likelihood and Severity properties with the error source declarations.

<table>
<thead>
<tr>
<th>composite error behavior states</th>
</tr>
</thead>
<tbody>
<tr>
<td>[s1.Failed or s2.Failed] -&gt; Failed;</td>
</tr>
<tr>
<td>[s1.Failed and s2.Failed] -&gt; Failed;</td>
</tr>
<tr>
<td>[s1.Failed and s3.Failed] -&gt; Failed;</td>
</tr>
<tr>
<td>[s3.Failed and s2.Failed] -&gt; Failed;</td>
</tr>
<tr>
<td>[po.Failed or pm.Failed or pl.Failed] -&gt; Failed;</td>
</tr>
<tr>
<td>end composite;</td>
</tr>
</tbody>
</table>

Listing 4: Composite Error Behavior of the Overall System

Figure 9: Generated Functional Hazard Analysis

5.4 Supporting the ARP4761 Safety Validation Process

5.4.1 Functional Hazard Assessment

Our toolset processes the model to generate the FHA report which enumerates error events and propagation within the architecture that represent hazards. Only hazards with high severity level, reflecting high potential for accident, are included in the report. Being part of the PSSA of the ARP4761, it clearly identifies each component that contributes to a system failure. Using the previously defined AADL model of the embedded control systems, this document lists all error sources occurring from sensors, processors or actuators. Each row corresponds to an error type that can be an error source with its associated component and textual information provided by the Hazards property, including severity and likelihood of a hazard. An extract of the FHA report for this case-study is shown in figure 9.

5.4.2 Fault Tree Analysis

Another document material used during the PSSA of the ARP4761 is the FTA. To support this analysis, our tools generate the FTA from the AADL model. It represents the decomposition of an error event into a tree with all sub-events. To do so, we use the composite error behavior of the system, as defined in listing 4.

The failure conditions that trigger a switch of the main system to the failed error state are translated into the Fault-Tree. An extract of the generated Fault-Tree is shown in figure 10, showing all error events that contribute to switch the top-level system into the Failed error state.

5.4.3 Reliability Assessment

Another method recommended by the ARP4761 standard and used during the SSA is the use of Markov Chain based methods to evaluate the fault occurrence. To support this analysis, we export the architecture fault model into a Markov Chain model.

As described in section 4.3, our tool generates a Markov Chain for each component using its associated component error behavior and the error propagation between components along connections and bindings. Mapping rules translates AADL component into modules in an appropriate notation for stochastic analysis and interconnect them according to the core AADL notation (connection and bindings of the components) and its error-model description (error state, events, propagation, etc.). This automated translation ensures that stochastic analysis relies on the same specification than the architecture (AADL) model and no error has been introduced when translating system specification from one notation to another. Then, user can use the stochastic model for simulation and analysis purposes (for example, probability of an error).

From our AADL model, our tool generates a Markov Chain (using the notation supported by PRISM [10]) with eight modules (corresponding to a component with a component error behavior specification). The resulting model is imported into PRISM [10] in order to simulate and verify system specification. In the PRISM tool[10], users specify results of interest, e.g., the probability that the system remains in an operational state despite fault occurrences reflecting system availability.

Figure 10: Fault-Tree Analysis for the Embedded Case Study

Figure 11: Analysis of Failure Occurrence of One Processor with PRISM

In order to illustrate the use of this stochastic analysis, we import the model into PRISM and check the probability of a transient failure of one processor over 10 days (formulas and examples are publicly available on [2]). A specific verification formula that evaluates transient failure occurrence of processor components. The tool provide a graphical representation of the result, figure 11 shows the associated result. As shown, after one day, the probability of failure of one processor is near 100%, which is consistent with the initial system specifications.
6. CONCLUSIONS AND PERSPECTIVES

Cyber-Physical Systems operate critical functions that must be carefully designed and validated. An error potentially means mission failure or loss of life, hazards must be evaluated and eliminated or handled, depending on their criticality and associated consequences. To perform this process, development of such systems requires following rigorous practice standard to evaluate system safety and show evidence of absence of critical failure. This effort requires the production of various reports that are loosely coupled with system specification and other implementation artifacts. This may lead to inconsistency and invalid assumptions in the different analyses and the architectural design and implementation. This practice is labor intensive and is often performed only once early in the system engineering life cycle by system and safety engineers.

To overcome these issues, we have presented an approach to support the safety and reliability evaluation process using architecture (AADL) models. This architecture-centric model-based approach leverages the same architecture model for different analyses, resulting in increased consistency between analysis results. The approach automates the generation of documentation materials for supporting the different aspects of the validation process: Fault and Hazard Assessment, Fault-Tree Analysis, simulation through Markov Chain models, etc. We support such a process in our AADL tool environment and generate much of the validation material required by safety practice standards such as ARP4761. This support has been illustrated in a case study.

We have shown how the Error Model Annex of AADL provides several mechanisms to describe errors/faults, their propagations and the system error state based on error event or incoming error propagations. These mechanisms provide a convenient flexibility for systems designers, allowing them to associate error behavior specifications with elements of the model. Also, using the same model for validating different architecture criteria would increase stakeholders’ confidence in the correctness of the architecture by showing evidence of requirements enforcement.

In order to ensure consistent analysis results, the tool assumes that the component error behavior specifications are consistent with error propagation specifications between the components and with the error behavior specification of the composite system. Specification of such a set of model consistency constraints is currently being completed as part of the revised Error Model Annex standard.

Another potential improvement is to simplify the interface between system designers and third-party tools to verify/validate the system. For now, verifying system safety and reliability properties with Markov Chain requires knowledge of a tool-specific notation (PRISM) and the mapping between the AADL model and the generated Markov Chain model in order to specify a formula that captures the essence of the analysis. Such a formula could be generated automatically from the high level specification of the reliability or availability requirement associated with the AADL model.

Acknowledgment

Copyright 2013 Carnegie Mellon University

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.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EXCEPT AS EXPRESSLY SET FORTH IN THE OFFICIAL OF THE UNIVERSITY. THIS UNIVERSITY MAKES NO WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL.

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


DM-0000730

7. REFERENCES


