Supporting the ARP4761 Safety Assessment Process with AADL

Julien Delange, Peter H. Feiler

Supporting the ARP4761 Safety Assessment Process with AADL

Julien Delange and Peter Feiler
Carnegie Melon Software Engineering Institute, 4500 5th Avenue, 15213 Pittsburgh, PA USA

Abstract

Cyber-physical systems, used in domains such as avionics or medical devices, perform 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 software importance continues to grow, integration and validation activities are becoming overwhelming. 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 models that are analyzed to detect errors, issues or defects that are usually detected lately in the development process. This predictive analysis approach is automated by tools, avoiding any tedious manual efforts. In this paper, we present our model-based approach for capture and analyze system safety and reliability. We have added an error behavior annotation to SAE AADL, a standard for modeling embedded software system architectures. We also implemented analysis methods to automate safety analysis and production of safety assessment reports, as requested by safety recommended practices standards (such as ARP4761).

Introduction

Context

Cyber-physical systems perform critical functions under constrained and potentially hostile circumstances. Because errors 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.

Much functionality is 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 system complexity of software components collocated on the same networked execution platform. This creates new 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.

Current Problem

Recommended practice standards for system safety assessment and certification, such as ARP4761 [13] or DO-178B/C [19], provide guidance for validation and verification of safety-critical software and systems. In current practice the safety assessment process is is labor-intensive, mostly relying on engineer's ability to correctly interpret a textual specification of the system. In addition, most errors are introduced during requirements and architecture design phases [15] but are likely not detected until integration or operational phases. Resulting re-work efforts late in the development process at 100-1000 times the cost of in-phase correction make such systems increasingly not affordable and delay in product delivery [8].

System safety requirements are often not refined into software safety requirements and safety analysis does not extend into identifying software induced hazards and into verifying software design against these safety requirements to ensure that potential software errors are avoided or correctly handled. For example, for safety purposes, a software subsystem must be checked to ensure that it does not send more than its bandwidth allocation on a shared network in order to assure timely delivery of data between other subsystems on the same network. Another example is to ensure that a change in the deployment configuration of software onto hardware

to balance the workload across processors does not violate assumptions about non-collocation of software subsystems on the same processor or the isolation of subsystems with different criticality level in a mixed criticality application system.

A formalized definition of each component allows integration issues to be detected analytically during the design process, rather than system integration test. This would thus address one issue to reduce the re-work efforts and keep software production affordable [8, 15].

Approach

To address these issues, we propose to automate the safety validation process and generate validation materials from formal specifications. To do so, we propose an architecture centric modeling framework that captures the system and software architecture annotated with safety-related requirements and auto produce certification documents [13]. This would increase stakeholders' confidence in system reliability, safety and correctness.

We implement such a framework by adding a specification of faults and hazards, error propagation and mitigation behavior to architecture models expressed in the SAE International Architecture Analysis and Design Language (AADL) standard [16] and its Error Model Annex standard [7]. 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 1.

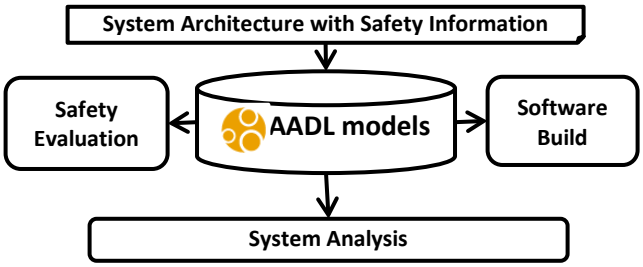

Figure 1 - Architecture-Centric Model Based Method for System Safety Evaluation

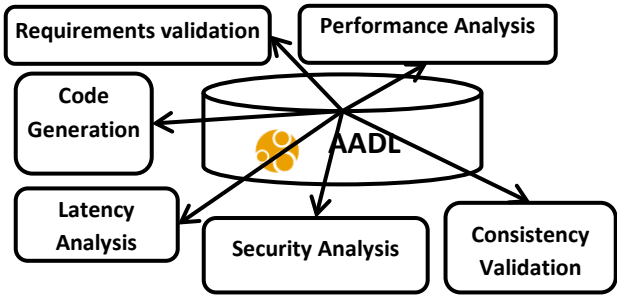

Figure 2 - AADL ecosystem for Software System Design and Implementation

The paper is organized as follows. We first give an overview of AADL and the ARP4761 [13] 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 [13]. Finally, we illustrate the application of our approach with a case study.

Related Work

Existing literature [12, 1] shows evidence that software errors and bugs might have catastrophic consequences. Several methods and approaches address this concern: some [13] 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 (such as ARP4761 [13]) from an architecture model and ensure system consistency across different notation and analysis. To do so, we reuse an existing industry standard architecture language [16] and extend it to support safety evaluation. Similar extensions exist for expressing constraints on architecture models.

Architecture Analysis Design Language

The Architecture Analysis and Design Language (AADL) [16] 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. For each one 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. 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 can be attached to AADL [16] components to augment the component description and specify additional characteristics and requirements (for example, specifying the component behavior). 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.

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 2. 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.

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

1. **Augmenting the original Error Model Annex** language for AADL with safety semantics and a fault ontology to support the modeling of error behaviors.
2. **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 [10, 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 [13] safety standard, detailing the necessary constructs to add in AADL models for its support.

SAE ARP4761

SAE 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.

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.

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. This annex allows users to specify 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 [10], this effort is a revision that extends its notation and semantics. The enhanced 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.

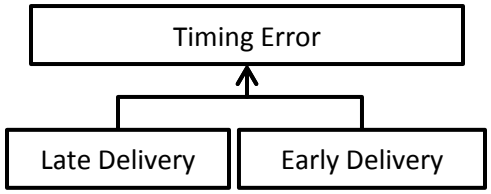

Figure 3 - Error-Type Hierarchy

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

Listing 1 - Error-Type Set Example

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 3 and Listing 1, timing error has two sub-types: late delivery and early delivery, which are mutually exclusive.

An `error type set` is used to specify sets of possible error types. Listing 1 shows an error type set that consists of all subtypes of `TimingError` and a value error of type `InvalidValue`. 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 (messages, commands, etc.),
- Timing errors and value errors on individual items being communicated,
- Rate and sequence errors for streams of service items (e.g., streams of sensor readings),
- Replication errors in the form of asymmetric value, timing, and omission errors in redundant systems, and

- Concurrency errors when accessing shared logical or physical resources.

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.

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].

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.

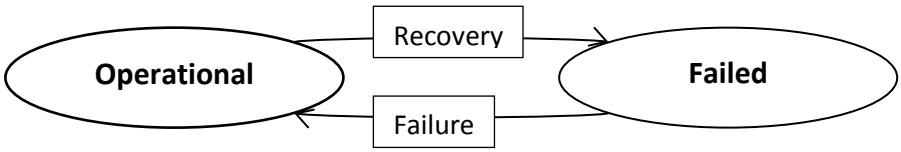

Figure 4 - Error Behavior State Machine

Listing 2 illustrates the textual declaration of the state machine and Figure 4 shows its corresponding graphical representation. 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.

```
error behavior Simple
events
 failure : error event ;
 recov : error event ;
states
 Operational : initial state ;
 Failed : state ;
transitions
 t1 : Operational -[ failure ]-> Failed;
 t2 : Failed -[recov]-> Operational;
end behavior ;
```

Listing 2 - Error Behavior State Machine

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

Listing 3 - Example of a Composite Error Behavior

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, 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.

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.

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.).

Supporting the Safety Evaluation Process

We automate the safety analysis process by processing the architecture fault model (i.e., the core AADL models enhanced with Error Model clauses), generating appropriate analysis models, such as fault propagation graphs, Markov models and fault trees, and producing documentation required by ARP4761 (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 errors 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).fault.

These safety analysis functions are included in the Open Source AADL Toolkit Environment (OSATE) [3], our Eclipse-based AADL modeling framework. It is freely available under an Open-Source license (the Eclipse Public License). OSATE utilizes interfaces with existing safety analysis tools (as OpenFTA [14] for the FTA) and can be adapted to established in-house tool suites.

The next sections illustrate the support of the FHA, FTA generation. The other methods are supported in a similar fashion and have been demonstrated on the wheel braking systems example that accompanies the SAE ARP4761 standard document [Delange].

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.

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.

Example

We apply our approach to a case study that contains several components with error model annotations. The following sections describe its translation into AADL and the use of our tools to support ARP4761 safety process.

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 send 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 5. 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 about30 seconds

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

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

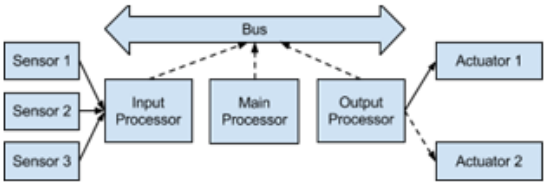

Figure 5 - Overview of the Embedded Case Study

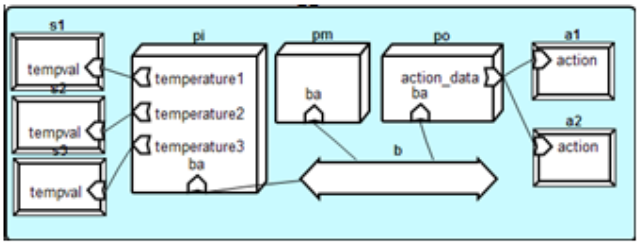

Figure 6 - AADL model of the Embedded Case Study

AADL Model

This architecture is then translated into an AADL model (as shown in Figure 6) 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 (PWN or serial bus) – mapped into `a required bus access` feature. The main system aggregates all these components altogether to represent the hierarchical architecture.

Architecture Fault Model

We add error information 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 instances.

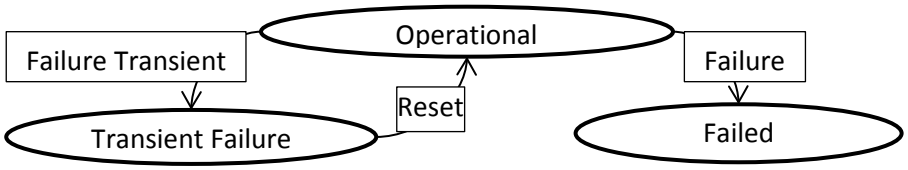

Figure 7 - Error Behavior State Machine for the Processor Components

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 7:

- 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, we add the `OccurenceDistribution` 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 er 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 Listing 4. Finally, we associate the `Hazard`, property with the `error source` declarations.

```
composite error behavior
states
[a1.Failed or a2.Failed ]-> Failed ;
[(s1.Failed and s2.Failed)or(s1.Failed and s3.Failed)or(s3.Failed and s2.Failed)]-> Failed ;
[po.Failed or pm.Failed or pi.Failed ]-> Failed ;
end composite ;
```

Listing 4 - Composite Error Behavior of the Overall System

Supporting the ARP4761 Safety Assessment Process

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 8.

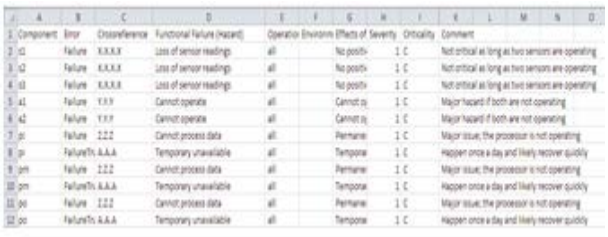

Figure 8 - Generated Functional Hazard Analysis

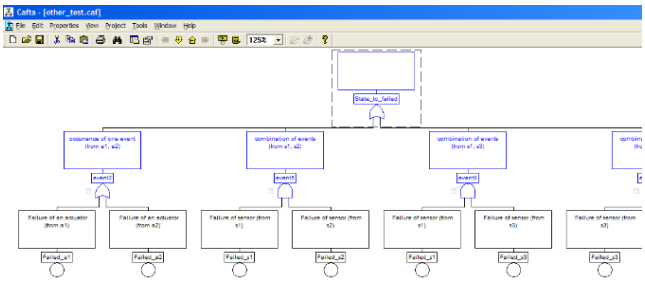

Figure 9 - Fault-Tree Analysis of the Embedded Case-Study

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 9, showing all error events that contribute to switch the top-level system into the `Failed` error state.

Conclusion

Cyber-Physical must be carefully designed and validated: as an error potentially means mission failure or loss of life, hazards must be evaluated and eliminated. For that reason, 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, which may lead to inconsistencies. In addition, this practice is labor intensive and error-prone.

To overcome these issues, we have presented an approach to support the safety and reliability evaluation process using architecture (AADL) models. This architecture-centric method leverages the same model for different analyses, resulting in increased consistency between analysis results. Tools automate the generation of documentation materials for supporting the different aspects of the validation process: Fault and Hazard Assessment, Fault-Tree Analysis, etc. We have shown how the Error Model Annex of AADL provides several

mechanisms to describe errors/faults, their propagations. They 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.

Acknowledgements

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.

DM-0000789

References

1. M. Ben-Ari. The bug that destroyed a rocket. *ACM SIGCSEBulletin*, 33(2):58, 2001.
2. Carnegie Mellon Software Engineering Institute. AADL official wiki - embedded control example. https://wiki.sei.cmu.edu/aadl/index.php/Embedded_Control_Example.
3. Carnegie Mellon Software Engineering Institute. OSATE - http://www.aadl.info - Technical report, 2006.
4. B. Ern, V. Y. Nguyen, and T. Noll. Characterization of failure effects on aadl models. In *Proceedings of The 32ndInternational Conference on Computer Safety, Reliability and Security (SAFECOMP 2013)*, 2013.
5. P. Feiler. Challenges in validating safety-critical embedded systems. In *AEROTECH 2009*. SAE, Nov 2009.
6. P. Feiler and J. Hansson. Flow latency analysis with the Architecture Analysis and Design Language (AADL) - TNCMU/SEI-2007-tn-010. Technical report, Carnegie Mellon SEI, December 2007.
7. SAE International. *AADL Error Model Annex, (Standards Document AS5506/1, 2006. in revision as DocumentAS5506/3 2013*, 2013.
8. C. Hagen and J. Sorenson. Delivering military software affordably. *Defense AT&L*.
9. J. Hansson and A. Greenhouse. Modeling and validating security and confidentiality in system architectures. Technical report, Carnegie Mellon Software Engineering Institute, 2008.
10. A.-E. Rugina, P. H. Feiler, K. Kanoun, and M. Kaaniche.Software dependability modeling using an industry standard architecture description language. In *Proceedings of 4thEuropean Congress ERTS, Toulouse*, 2008
11. N. G. Leveson. A new approach to hazard analysis for complex systems. In *International Conference of the System Safety Society*, 2003.
12. N. G. Leveson and C. S. Turner. An investigation of thetherac-25 accidents. *Computer*, 26(7):18–41, 1993.
13. SAE International. *ARP4761 - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, 1996.
14. OpenFTA - http://www.openfta.com
15. National Institute of Standards and Technology (NIST). The Economic Impacts of Inadequate Infrastructure for Software Testing - http://www.nist.gov/director/prog-ofc/report02-3.pdf .Technical report, 2002.
16. SAE International. *AS5506 - Architecture Analysis and Design Language (AADL)*, 2012.
17. R. F. Paige, L. M. Rose, X. Ge, D. S. Kolovos, and P. J.Brooke. Models in software engineering. Chapter FPTC: Automated Safety Analysis for Domain-Specific Languages. Springer-Verlag, 2009.
18. D. Powell. Failure mode assumptions and assumption coverage. In *Fault-Tolerant Computing, 1992. FTCS-22.Digest of Papers., Twenty-Second International Symposium on*, pages 386–395, 1992.
19. RTCA. *Software considerations in airborne systems and equipment certification*.