Illustrating the AADL Error Modeling Annex (v. 2) Using a Simple Safety-Critical Medical Device*

Brian Larson, John Hatcliff, Kim Fowler
Kansas State University
{brl,hatcliff,kimrfowler}@ksu.edu

Julien Delange
Carnegie Mellon Software Engineering Institute
jdelange@sei.cmu.edu

ABSTRACT

Developing and certifying safety-critical and highly reliable systems almost always includes significant emphasis on hazard analysis and risk assessment. There have been substantial improvements in automation and formalization of other aspects of critical system engineering including model-driven development, analysis of source code and models, and verification techniques. However, hazard analysis and risk assessment are still largely manual and informal activities, tool support is limited (which for both development and auditing, increases time and effort and reduces accuracy and correctness), and artifacts are not integrated with architectural descriptions, system interfaces, high-level behavioral descriptions or code.

The Error Model annex of the Architecture Analysis and Design Language (AADL) provides formal and automated support for a variety of forms of hazard analysis and risk assessment activities. Specifically, it enables engineers to formally specify errors, error propagation, error mitigation – using annotations that are integrated with formal architecture and behavioral descriptions written in AADL. Plug-ins to the Open-Source AADL Tool Environment (OSATE) process these annotations to provide various forms of (semi)-automated support for reliability predication and tasks necessary to support common hazard analysis and risk assessment techniques such as Failure Modes and Effects Analysis (FMEA), Fault Tree Analysis (FTA), and Functional Hazard Analysis (FHA).

In this paper, we illustrate basic aspects of Error Modeling in AADL using a simple safety-critical medical system – an infant incubator called "Isolette". We summarize standard tasks involved in FMEA and FTA, we illustrate the principal steps involved in AADL Error Modeling for the Isolette, and we describe how those steps relate to FMEA and FTA. We give a brief survey of emerging automated analysis tools implemented as plug-ins to the AADL OSATE environment that process error modeling annotations. We believe this introduction to Error Modeling in AADL can expose engineers of high-integrity systems to techniques and tools that can provide a more rigorous, automated, and integrated approach to important risk management activities.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods, Reliability*; D.2.6 [**Programming Environments**]: [Integrated environments]; D.2.11 [**Software Architectures**]: [Languages]

Keywords

hazard analysis, risk assessment, error modeling, error analysis, AADL, formal architecture

1. INTRODUCTION

Innovations in model-driven development, static analyses, type systems, and automated deduction techniques have improved methods and tools for developing high-integrity systems. These improvements center around system design, implementation, and verification and validation. However, developing and certifying safety-critical and highly reliable systems includes additional activities such as hazard analysis techniques (HAT), risk assessment, and reliability predictions that play an important role in safety evaluation and certification [4, 13].[1] For example, "bottom up" hazard analysis techniques assess how individual components of a system may fail and analyze (via a systematic tracing of system data and control paths as well as other more indirect notions of coupling) how those failures might impact the desired functionality and safety of the system. "Top down" techniques may start by identifying (a) the forms of accidents and losses that should be avoided, and (b) system states that might lead to those accidents, and then proceed by tracing down through the system architecture, design, or implementation to identify failures or faults that might lead to those

*Work supported in part by the US National Science Foundation (NSF) (#0932289, #1239543), the NSF US Food and Drug Administration Scholar-in-Residence Program (#1065887, #1238431) the National Institutes of Health / NIBIB Quantum Program, and the US Air Force Office of Scientific Research (AFOSR) (#FA9550-09-1-0138).

HILT'13, November 12–14, 2013, Pittsburgh, PA, USA.

http://dx.doi.org/10.1145/2527269.2527271.

[1] According to Leveson [14, pp. 7–14], and we agree, safety is not synonymous with reliability. Safety is freedom from accidents – events that result in loss of life or some other form of loss important to system stakeholders. Reliability is the probability that a piece of equipment or component will perform its intended function satisfactorily for a prescribed time and under stipulated environment conditions. The techniques and tools that we discuss in this paper can address both safety and reliability.

states. Tooling and automated support for these safety and risk assessment activities has not kept pace with advances in other dimensions of development highlighted above.

1.1 Short-comings in Conventional Risk Assessment Techniques

Informal (non-machine readable) inputs: Advances in software development environments have illustrated the usefulness of attaching various forms of annotations and pragmas directly in source code and other machine-readable development artifacts. These annotations and artifacts into which they are embedded can then be used as inputs to automated analyses that propagate information through artifacts and either produce additional annotations or analysis results directly linked to the artifacts. For example, in modern development environments for Java, C#, and Ada, a developer may introduce simple annotations indicating that certain variables are intended to hold reference values that are *always non-null* (remaining variables are allowed to hold either null *or* non-null values). This information is then propagated throughout the program to (a) deduce when other variables are always non-null and (b) to detect possible run-time exceptions due to dereference of null pointers.

Unfortunately, despite long-standing formal approaches from academia [8] and automation in some commercial tools, inputs to HAT are most commonly captured in separate natural-language-based text documents or spreadsheets. Thus, even though many aspects of HAT include identifying component failure modes (which, *e.g.*, could be captured as simple enumeration annotations attached to a component) and reasoning about propagation of information (*e.g.*, how faults and effects of faults flow through the system), the primary inputs to HAT (*e.g.*, system architectures including both hardware and software component structures, component failure rates, common hazards for a particular device type) are usually not represented formally or precisely enough for now-common paradigms of code-level static flow analysis and extended type-checking to be applied in support of HAT.

Manual construction: One of the main purposes of HAT is to systematically guide engineers through a set of analysis steps leading to critical decision points – points in the analysis that require judgements about the absence or presence of hazards, risk mitigation strategies, and the degree of residual risk. However, many of the steps leading up to these decision points are highly repetitive and intermediate results are calculated in a straightforward manner from other development artifacts (*e.g.*, identifying enclosing components or subsystems in a Failure Modes and Effects Analysis (FMEA)). Although some tools exist provide limited forms of automated support for these steps, such tools are not widely applied and their effectiveness is limited. The practical impact is that the bulk of HAT activities are carried out manually.

Lack of integration with development artifacts and limited traceability: In current practice, activities and results of HAT are recorded in text editors or spreadsheets. Information driving HAT, including specifications of system architecture descriptions, dependencies between system components, and inter-component information flow is spread across many artifacts. This hinders traceability between different artifacts, and the informal nature of the information representation prevents automated support for navigating traceability links. In contrast, an approach that includes formal descriptions of system architectures could enable HAT information including error types, error propagation paths, and component failure probabilities to be captured directly as annotations in a system architecture description.

1.2 AADL and Error Modeling Annex

AADL [6, 1] is a strong candidate for formal architecture specification in high-integrity systems. AADL was created in response to the high cost associated with (far too frequent) failed subsystem integration attempts due to ambiguous or incompletely documented component interfaces. AADL is now used in several industrial development settings. For example, on the System Architecture Virtual Integration (SAVI) effort, aircraft manufacturers together with subcontractors use AADL to define a precise system architecture using an "integrate then build" design approach. In this approach, important interactions are specified, interfaces are designed, and integration is verified before the internals of components are built. Once correct integration is established, contractors provide implementations that are compliant with the architecture [7, 19].

A number of AADL users are interested in the development of hazard analysis, reliability prediction, and risk assessment techniques (all separate but related issues) that can be deeply integrated with formal architecture specifications and system integration activities. This interest has prompted the development of the AADL Error Modeling framework. AADL includes an *annex* mechanism by which additional modeling notations or supporting tools can be added to the standard, and this mechanism is used to define the current version of the Error Modeling Framework – Error Model Version 2 (EMV2). As described in the EMV2 annex document [5], EMV2 enables modeling of different types of faults, fault behavior of individual system components, modeling of fault propagation affecting related components in terms of peer to peer interactions and deployment relationships between software components and their execution platform, modeling of aggregation of fault behavior and propagation in terms of the component hierarchy, as well as specification of fault tolerance strategies expected in the actual system architecture. The objective of EMV2 is to support qualitative and quantitative assessments of system dependability, *i.e.*, reliability, availability, integrity (safety, security), and survivability, as well as compliance of the system to the specified fault tolerance strategies from an annotated architecture model of the embedded software, computer platform, and physical system. Thus, EMV2 provides a foundation for addressing the shortcomings described in Section 1.1.

1.3 This Paper

We have found it useful to advocate for the use of AADL EMV2 by illustrating its application to simple systems and by explicitly identifying how it might support and improve upon aspects of conventional risk assessment techniques. Regarding the shortcomings of conventional risk assessment techniques identified above, AADL EMV2 provides formal machine-readable inputs to the risk assessment process that are directly integrated with formal architecture descriptions and other implementation oriented artifacts. This provides an annotation-based approach to HAT that developers will recognize as similar to annotation-based code-level static analysis and verification techniques. With these formal arch-

itecture-integrated annotations as a foundation, we believe that many of the tedious, repetitive, and error prone, manual steps in HAT can be automated. Moreover, because the vision of AADL includes code generation from formal architecture descriptions, the architecture-integrated approach of EMV2 provides the basis for eventual development of risk assessment tools that offer strong traceability throughout implementation-oriented development artifacts.

The specific contributions of this paper are as follows:

- We illustrate the basic error modeling constructs and methodology of AADL EMV2 using a simple safety-critical system – an infant incubator, referred to as an "Isolette." This example, orginally introduced by Lempia and Miller in the FAA Requirements Engineering Management Handbook [12], is also being used by Blouin to illustrate the AADL Requirements Annex [3]. Thus, the work in this paper contributes to what we hope will eventually be an end-to-end illustration of AADL-based development using the Isolette example.
- We describe how EMV2 modeling and tools formalize and provide automated support for important artifacts and tasks required in conventional risk assessment techniques including FMEA and FTA.
- We provide as open-source artifacts the AADL models for the Isolette including EMV2 annotations.[2]

2. ISOLETTE EXAMPLE

The Federal Aviation Administration's (FAA) Requirements Engineering Management Handbook (REMH) [12] uses an example of an infant incubator called an "Isolette" to illustrate best practices for writing requirements for embedded systems. We use the Isolette as the primary illustration in this paper because it is relatively simple (and thus can be discussed within the space constraints of this paper) while still rich enough to illustrate a number of dimensions in risk assessment.

Figure 1 presents a diagram the Isolette's primary system components and environment interactions. The Isolette thermostat takes as input an air temperature value from a temperature sensor and controls a heat sources to produce an air temperature within a target range specified by the clinician through the operator interface. Safety concerns include ensuring that infant is not harmed by air temperature inside the isolette being too hot or too cool. The Isolette uses a subsystem separate from the operational thermostat to sound an alarm if hazardous temperatures are detected.

Figure 2 illustrates the AADL graphical component architecture notation for the top-level system architecture for the Isolette. Triangles represent component ports, and lines represent data flow connections between ports. AADL also includes a textual representation that allows a variety of formal specification and property notations to be associated with different elements in the architecture. Space constraints do not permit a detailed explanation, but important AADL features are rich structuring support (allowing nested components, grouping of ports and connections, abstract components and refinement, etc.), buffered and unbuffered ports, a variety of dispatch modes for threads (event triggered, time triggered), as well as a rich type system and a rigorous notion of what it means for two opposing ports to be "plug-compatible."

[2]The Isolette AADL+EMV2 artifacts are available at `http://santos.cis.ksu.edu/BLESS/examples/isolette.zip`.

3. CONVENTIONAL RISK ASSESSMENT TECHNIQUES

3.1 Failure Modes and Effects Analysis (FMEA)

Failure Modes and Effects Analysis (FMEA), helps a designer to determine if the design must change or improve to reduce potential failures. FMEA examines single-point failures, both their types and propagation to systemic effects. FMEA has the primary purpose of determining operational faults and safety. FMEA can have the secondary purpose of estimating the system reliability from the component reliabilities [4, pp. 235–259]. FMEA is a tabular, bottom-up approach for single-point failures; it is both qualitative and quantitative in nature. It helps a designer or analyst to determine failure effects at various levels: functional or component level, modular or assembly level, subsystem level, and top-level system.

Typically a designer or analyst will begin the FMEA at a specific level of abstraction in an architecture, consider the component boundaries (interfaces) at the level of abstraction, and study how failures propagate and affect other subsystems [4, pp. 235–259]. The goal for FMEA is to assess system functionality and safety in the event that a component fails. FMEA aims to identify both the type of failure within each component and its effect on component behavior. Once identified and the component and system effects understood, FMEA aims to determine the extent of criticality for the system. The criticality helps a developer to address risks in reliability and to set priorities during design. An FMEA, which can occupy hundreds of pages and is often manually prepared by a domain expert, can tell a regulator that the designer(s) attempted a measure of discipline and rigor during development. While quantity does not imply discipline and rigor, it can be an indicator of effort. The point is that performing FMEA correctly is a lot of work; automation and integrated tools such as those offered by the OSATE EMV2 tools are sorely needed.

FMEA attempts to answer these questions:

- How can each component fail?
- What are the effects of each failure?
- What are the consequences of each failure?

The effects are the physical manifestations of a failure. The consequences are the outworking of those manifestations on the system or its operators.

If reliability data are available, then FMEA addresses these questions as well:

- How frequently can a component fail?
- How does a component's failure affect system reliability?

3.1.1 FMEA Inputs

Part 1: Determine the system context, its mission, system design, the level of analysis (component, module, or subsystem), operational constraints (e.g. logical dependencies, data flow), and the boundaries where failures appear or stop. Failure effects propagate over boundaries or are contained by them.

Part 2: Determine specific data for each component:

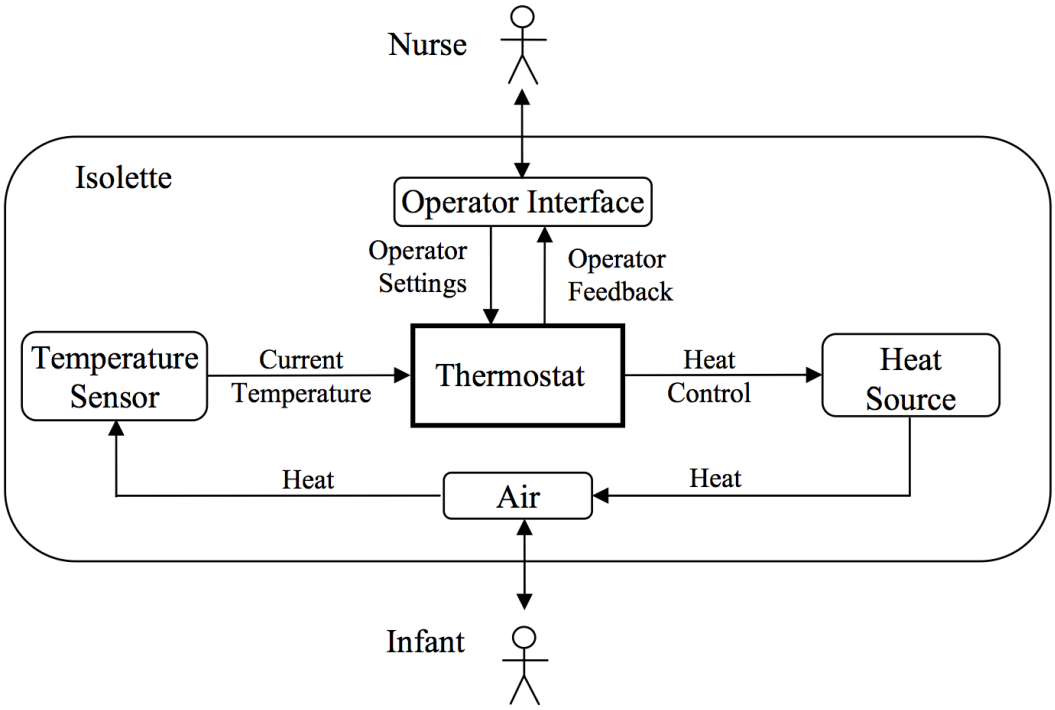

Figure 1: Operational Context for Isolette Thermostat

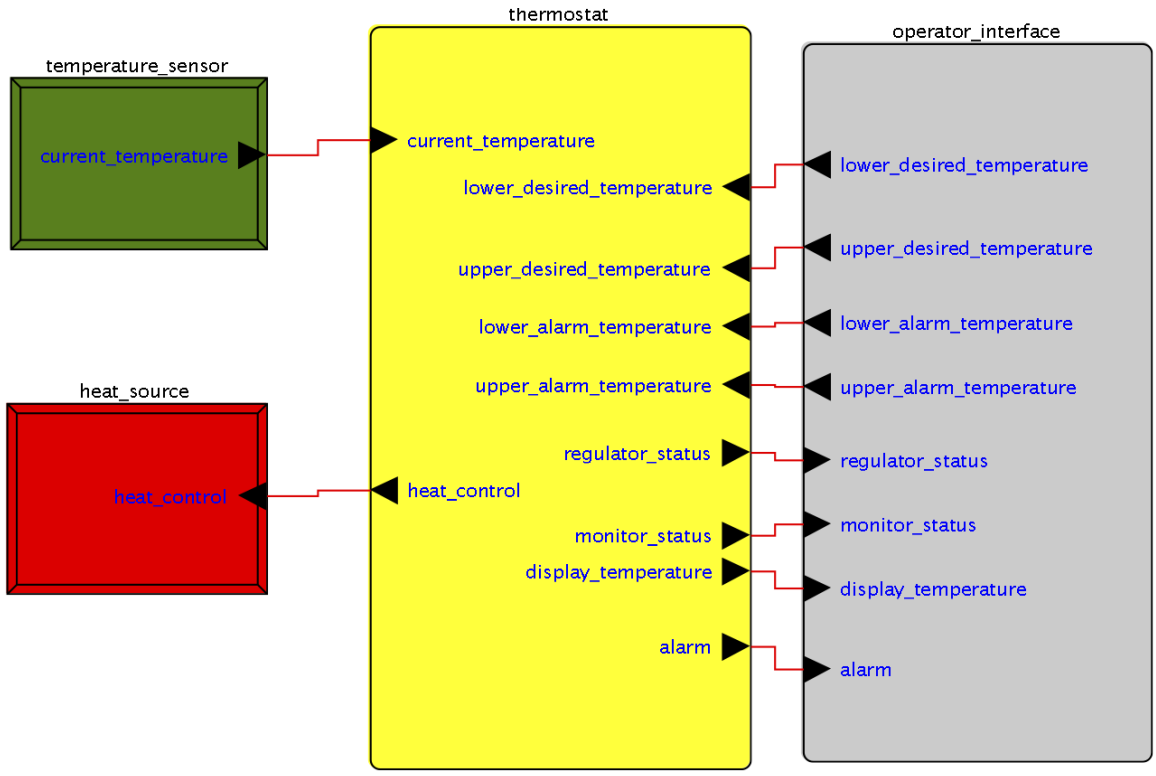

Figure 2: AADL Model of Isolette

- Possible failure types, e.g. two electrical signal pins shorted together
- Possible operational modes, e.g. expected mechanical actions from control operations
- Connection to other components
- Immediate effects of failure
- Systemic effects of failure
- (For reliability calculations: probability of failure or occurrence)

3.1.2 FMEA Outputs

The results of a FMEA are typically listed in tabular form and include, for each component, an enumeration of the modes in which a component can fail, the immediate effect of those failures as visible at the component's boundary, and the potential effects of the failure at the system level. Figure 3 illustrates just one set of potential failure effects for the heating element component of the Isolette.

3.1.3 FMEA Methodology / Tasks

Step 1: Understand and list potential hazards that lead to failures within the system. List the components to be analyzed.
Step 2: Collect and list failure modes for each component. An example of domain expertise, for the isolette, is that a designer might know that a heater element can experience corrosion in its connectors that increases electrical resistance and lowers heat dissipation.
Step 3: Collect and list effects for each component. The immediate effect is the failure as observed at component or module boundary, e.g. the module stops operating. The systemic effect is the observed effect of failure on overall system behavior, e.g. a failed module forces the system to stop operating. Note that examining more failure effects multiplies the number of lines in the analysis; also moving towards the component level and greater detail greatly expands the effort in developing FMEA.
Step 4: (if calculating reliability) - List probability of failure for each component (*e.g.*, from MIL-HDBK-217). This step assumes independent failures, that is there are no common causes between component failures.
Step 5: (if warranted or desired) - additional columns can be added to enhance understanding of failures and hazards as suggested by Ericson [4, pp. 235–259]:

- Causal factors between failure mode and effects columns to give more comment to type or location of failure or extenuating circumstances
- Method of failure detection after the effects columns, e.g.: inspection, test, none
- controls after the failure detection column, *e.g.*, Quality Assurance (QA), Built-in-test, None
- Hazard after the controls column, *e.g.*, Fire, Premature operation, Damage, None
- Final column for "Recommended Action"

3.2 Fault Tree Analysis (FTA)

Fault Tree Analysis, or FTA, helps a designer to determine if the design must change or improve to reduce potential failures. FTA examines sources, or root causes, of potential faults; it starts with descriptions of high-level, systemic fault types and then traces down to lower-level subsystems and modules to potential explanatory causes. FTA has the primary purpose of educating designers to potential problems for operational faults and safety. FTA has the secondary purpose of performing root cause analysis when a fault occurs [4, pp. 183–221].

FTA is a graphical, top-down approach for examining high-level faults. FTA is both qualitative and quantitative; it uses Boolean algebra, logic, and probability to generate descriptions of fault paths from cause to systemic effect. Like FMEA, FTA helps a designer or analyst to determine failure effects at various levels: functional or component level, modular or assembly level, subsystem level, and top-level system. Unlike FMEA, FTA can handle multiple, simultaneous failures and can support probabilistic risk assessment [4, pp. 183–221].

The goal for FTA is a top-down analysis focused on system design. FTA aims to identify potential root causes of system-level faults, which can provide a basis for reducing safety risks and can document safety considerations. Once FTA identifies potential root causes, the designer can assign criticality of the fault for the system. Like FMEA, assigning criticality within FTA helps a developer to address risks in safety (or reliability if reliability calculations are included) and set priorities during design. An FTA, which can occupy hundreds of pages and is often manually prepared by a domain expert, can tell a regulator that the designer(s) attempted a measure of discipline and rigor during development. As with FMEA, the same caveats regarding quality and quantity apply, and improvements in tooling and automation would provide significant benefits.

FTA attempts to answer these questions:

- What are the root causes of failures?
- What are the combinations and probabilities of causal factors in undesired events?
- What are the mechanisms and fault paths of undesired events?

FTA is similar to FMEA in using criticality and reliability but from a top-down perspective that can handle multiple, simultaneous failures. If criticality data are included, then FTA can address risk to reduce both severity and likelihood of problems. If reliability data are available, can calculate the probability of failure.

3.2.1 FTA Methodology / Tasks

Step 1: Define the system by collecting design artifacts, such as requirements, schematics, source code, and models. Layout the concept of operations, or CONOPs, to further the definition. Finally, understand the system behavior.
Step 2: Define undesired fault event by performing the following:

- Identify the final outcome of the undesired event
- Identify sub-events that lead to final event
- Begin to structure the connections using logic-gates
- Do Step 3 before completing structure of connections

Step 3: Establish rules of analysis by defining the boundaries of the analysis and the concepts that you can use: [4, pp. 194].

- I-N-S, which means, What is immediate (I), necessary (N), and sufficient (S) to cause the event? I-N-S helps the analyst from jumping ahead and focus on event chain.

Com-ponent	Failure	Immediate effect	System effect	failure rate (failures/yr.)	probability of detection
Heater element	fails open	airflow not heated	patient not warmed, alarm will sound once temperature drops below threshold, display will show low temperature	0.0876	0.9999
	fails on	airflow continuously heated, thermosafety switch opens on high temperature turning off heater	patient warmed too much, alarm will sound once temperature raises above threshold, display will show high temperature	1.9E-05	0.999
	fails to heat to specification	airflow not heated completely but warm enough	patient warmed sufficiently, display shows correct temperature	0.00028	0.05
	fails to heat to specification	airflow not heated completely and not warm enough for application	patient not warmed, alarm will sound once temperature drops below threshold, display will show low temperature	0.0876	0.9
	fails intermittent	airflow heated sometimes, not other times	when failure is open then patient not warmed, alarm will sound once temperature drops below threshold, display will show low temperature	0.0876	0.9
	fails intermittent	airflow heated sometimes, not other times	When heater is operational then patient warmed sufficiently, display shows correct temperature	6.6E-05	0.8

Figure 3: Example FMEA Outputs for Isolette (excerpts)

- SS-SC, asks, What is the source of the fault?
  - If the fault is a component failure, then classify as SC (state-of-the-component) fault.
  - If the fault is not component failure, then classify as SS (state-of-the-system) fault.
  - If the fault is SC, then perform event ORs of the P-S-C inputs.
  - If the fault is SS, then develop the event further by using I-N-S logic.
- P-S-C, which means, "What are the primary (P), secondary (S), and command (C) causes of the event?" P-S-C helps the analyst focus on specific causal factors.

Step 4: Build the fault tree, which is a repetitive process. At each level determine the cause, the effect, and the logical combination of logic symbols. The construction rules are almost self-evident but Ericson describes good, disciplined techniques [4, pp. 195–197].
Step 5: Establish cut sets, which are critical path(s) of subevent combinations that cause the undesirable final state event. While Ericson provides in-depth mathematical treatment of cut sets and probabilities, you can often perform a mere inspection to reveal the weak links that indicate the most important cut set(s) that lead to the fault event [4, pp. 199–206].

3.2.2 FTA Outputs

Figure 4 provides excerpts of an FTA report for the Isolette. This example output illustrates potential root causes for a failure to warm the air in the Isolette. Use of the top level OR-gate indicates that the failure may have several causes including an operator error, heater subsystem failure, air flow blockage, or thermosafety switch failure.

4. ERROR MODELING WITH AADL

The capability to model fault behavior, and from that model predict failure rates, was integral to MetaH from which AADL was derived. The original AADL standard SAE AS5506 incorporated some basic error modeling capabilities. The SAE International standard subcommittee AS-2C is expected put to ballot in the summer of 2013 a substantial revision to the error model framework, designated as Error Model Version 2 (EMV2). EMV2 incorporates several improvements such as a flexible error type system and pre-declared error type hierarchies with formal semantics.

As described in the AADL EMV2 draft standard [5, Section E.1], from the bottom-up, the error models of low-level components typically capture the results of failure modes and effects analysis. From the top-down, the error models of the overall system and high-level subsystems typically capture the results of system hazard analysis. One purpose of the formal modeling approach enabled by EMV2 is to ensure that the results of these analyses as captured in an architecture specification are consistent and complete with respect to each other. This enables an integrated approach that insures consistency and completeness between hazard analysis, failure modes and effects analysis (FMEA), and the safety and reliability analyses that relate the two.

In this section, we give an example-driven introduction to the basic EMV2 language constructs. While we aim for a reasonable coverage of constructs, space constraints do not permit a discussion of some of the more advanced features. We hope to provide a broader discussion in an expanded version of this paper to be released in the near future.

4.1 Faults, Errors, and Failures

Unfortunately, there are varying definitions in the literature of common terms such as faults, errors, and failures. To ground our discussions in upcoming sections, we present below definitions used in the EMV2 annex document [5].

> A *fault* is a root (phenomenological) cause of an error that can potentially result in a failure, i.e., an anomalous undesired change in the structure or data within a component that may cause that component to eventually fail to perform according to its nominal specification, i.e., result in malfunction or loss of function. Examples of faults include overheating of hardware circuits, or programmers making coding mistakes when producing source text. Errors, resulting failures, and error propagations are effects of a fault. The activation of a fault is represented by the Error Model concept of *error event*.
>
> An *error* is the difference in state from a correct state. The activation of a fault places a compo-

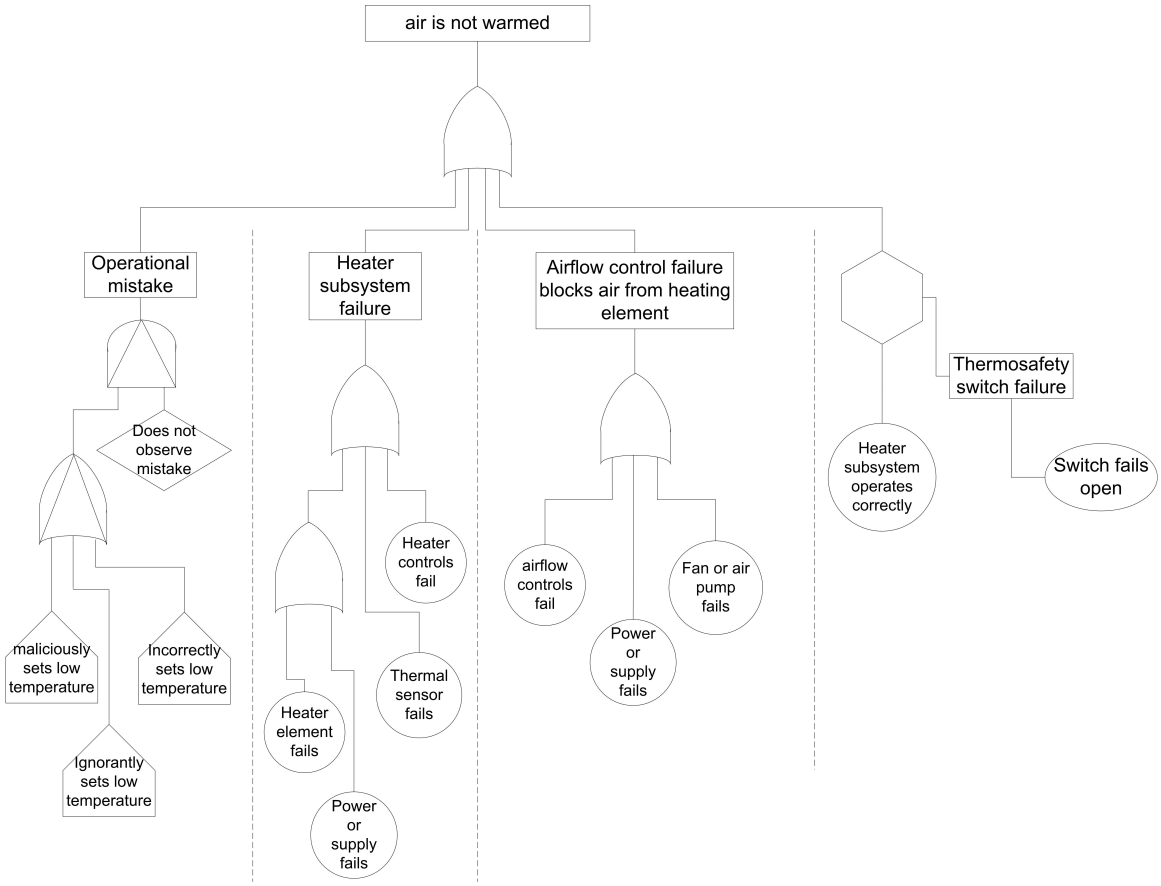

Figure 4: Example FTA Outputs for Isolette (excerpts)

nent into such an error state (possibly after some delay). An erroneous component may persist in that error state for some period of time before it behaves in a way that violates its nominal specification. For example, a burned out transistor (an activated fault) in an adder circuit does not cause a processor to violate its nominal specification until that circuit is used (after a delay) and produces an incorrect output value (erroneous state information). The Error Model concept of *error behavior state machine* is used to define error states, transitions, and conditions that trigger a transition.

A *failure* is a deviation in behavior from a nominal specification, i.e., a component can no longer function as intended in terms malfunction and loss of function as a consequence of an error. This may be due to an activated fault within the component or an error propagation from another component. The deviation can be characterized by type, persistence, and degree of severity. The degree to which a failure affects nominal behavior is referred to as severity of the failure. The Error Model concept of *error type* associated with error events, error states, and error propagations, as well as properties are used to characterize a failure.

4.2 Error Types

Hazard analysis techniques such as FMEA, Fault Hazard Analysis, and Subsystem Hazard Analysis [4] often include reasoning about (a) the different ways in which a component may fail and (b) how faults in one component can impact other components in a system.

EMV2 provides the ability to declare *error types* that represents a categorization or taxonomy of faults and errors relevant for a system. The error model language for the first version of AADL (AS5506) (which we refer to as EMV1) modeled only one kind of error. Users did all sorts of creative things with names to represent kinds of errors or faults. Nevertheless, Aerospace Corporation used EMV1 to model errors in satellites and ground stations with *thousands* of AADL components from which Markov models were extracted and solved. The enhancements included in EMV2 were inspired by the Fault Propagation and Transformation Calculus (FPTC)[18], developed by Wallace at York University. EMV2 improves upon FPTC by adding an error type system with elegant ways of expressing groups and combinations of errors.[3]

In EMV2, an error type can represent a category of fault arising in a certain component, the category of error being propagated, or the category of error represented by the error behavior state of a system or component. Error types can be organized into different type hierarchies, e.g., types relating to value errors and types relating to timing errors. These type hierarchies give rise to the conventional subtyping/inclusion polymorphism found in object-oriented languages with inheritance.

[3]The first author has been made significant contributions to the error type system of EMV2, including simplifying the original proposal for EMV2's type system, refactoring the EMV2 grammar to reduce the number of productions by two-thirds, and proposing a cleaner semantics.

Figure 5 gives a conceptual view of common error types that are *pre-defined* in EMV2. The left side of the figure illustrates errors related to the function of system services. For example, in the context of network enabled applications, the failure of the system to initialize its network authentication service might be classified as a *Service Omission* error. The right side of the figure illustrates errors that might be associated with data values or the communication of those values. Regarding the timing errors on the far right side of the figure, the failure of a network or bus to deliver a data value from a provider to a client within the bounds of its quality of service contract might be classified as a *Late Delivery Error*. In the value errors, a component that emits on its interface a value that lies outside of its specified range might be classified as an `OutOfRange` error. As an example of the subtyping that arises from such hierarchies, an `OutOfRange` error can be either an `AboveRange` error or an `BelowRange` error.

Error hierarchies such as those graphically represented in Figure 5, are actually defined textually in an AADL annex clause. Figure 6 provides an excerpt of the textual representation corresponding to the error types presented on the right-hand side of Figure 5.

```
TimingError: type;
EarlyDelivery: type extends TimingError;
LateDelivery: type extends TimingError;
ValueError: type;
UndetectableValueError: type extends
  ValueError;
BenignValueError: type extends ValueError;
OutOfRange: type extends BenignValueError;
OutOfBounds: type extends BenignValueError;
BelowRange: type extends OutOfRange;
AboveRange: type extends OutOfRange;
```

Figure 6: Standard, Predeclared Timing and Value Error Type Declarations

EMV2 also provides the ability to declare error types customized to a particular application. These may be completely new hierarchies, or they may extend or rename predefined error types. As presented in Section 3.1.3 **Step 2**, initial steps in a FMEA will identify failure modes for each component. For the Isolette example, we might use an EMV2 custom error definition like the one in Figure 7 to capture basic failure modes and other errors related to analysis of the Isolette. Hierarchy is used to introduce categories for `Alarm` and `Status` errors. These are subsequently refined to errors for specific Isolette components. The declaration of `ThreadFault` illustrates the ability to rename an EMV2 predeclared error type to obtain an Isolette-relevant name.

A heat-control error may harm the infant by becoming too hot or cold. Therefore, `HeatControlError` is the most important error type. The hazard for this error is mitigated by sounding an alarm if the isolette becomes dangerously warm or cool.

4.3 Attaching Error Sources to Architectural Models

In conventional approaches to HAT, association of different errors to system components and behaviors is only done *informally* in textual documentation – making the informa-

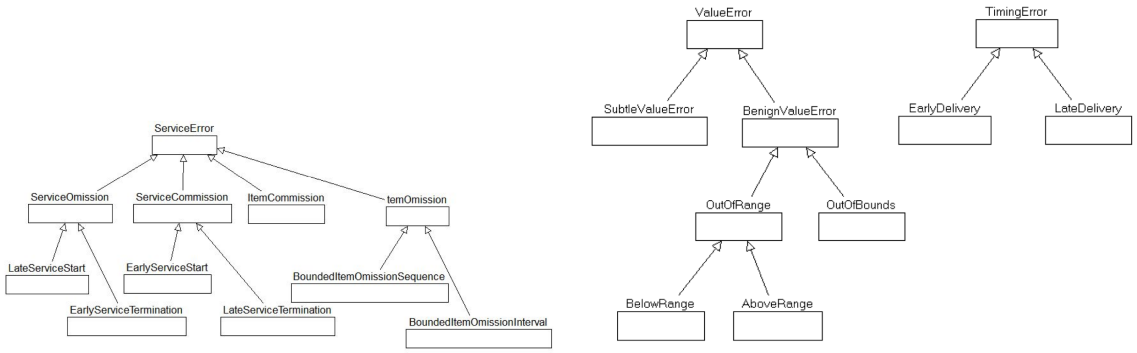

Figure 5: AADL Predeclared Error Types (excerpts)

```
annex EMV2
{**
error types
  HeatControlError: type;
  AlarmError : type;
  FalseAlarm : type extends AlarmError;
  MissedAlarm : type extends AlarmError;
  StatusError : type;
  RegulatorStatusError : type extends
    StatusError;
  RegulatorModeError : type extends
    StatusError;
  MonitorStatusError : type extends
    StatusError;
  MonitorModeError : type extends
    StatusError;
  ThreadFault renames type
    ErrorLibrary::EarlyServiceTermination;
  InternalError : type;
  DetectedFault : type;
  UndetectedFault : type;
end types;
**};
```

Figure 7: Custom Isolette Error Types

```
device temperature_sensor_ts features
  current_temperature : out data port
    Iso_Variables::current_temperature;
annex EMV2
{**
error propagations
    use types ErrorLibrary;
  current_temperature: out propagation
    {OutOfRange,UndetectableValueError};
flows  --define source of errors
  f: error source current_temperature
    {OutOfRange,UndetectableValueError};
properties
  EMV2::Occurrence =>  --out-of-range likelihood
    Iso_Properties::TemperatureSensorOutOfRange
      applies to current_temperature.OutOfRange;
  EMV2::Occurrence =>  --undetectable likelihood
     Iso_Properties::SensorUndetectableValueError
       applies to
       current_temperature.UndetectableValueError;
end propagations;
**};
end temperature_sensor_ts;
```

Figure 8: Temperature Sensor Error Model

tion hard to leverage in automated analysis or automated traceability queries. For example, as illustrated in Figure 3 of Section 3.1.2, in a conventional FMEA, the analysis would involve creating a table in a text document (perhaps following some template) that associates a component with the particular faults that occur within it. By formalizing both architecture and error types in AADL and EMV2, one can *formally* associate an error with a component by directly annotating the architecture model. With this information in place, not only can (all or portions of) conventional textual reports like those in Figure 3 be auto-generated, but the information can also be leveraged for automated analysis.

As an example, Figure 8 captures error properties of the temperature sensor component of the Isolette (see Figure 2). We aim to capture the fact that faults within the temperature sensor may cause it to produce both (a) erroneous temperatures that can be detected because the values are outside of the specified range of the sensor, or (b) temperatures that are within range (and thus cannot be detected by a range check), but are nevertheless incorrect. To formalize the results of Section 3.1.3 **Step 3** in which one identifies the immediate effects of a component failure as observed at component or module boundary, we begin by declaring that data emitted from port `current_temperature` may have error types `OutOfRange` and `UndetectableValueError` (*i.e.*, in range, but incorrect). The flow declaration indicates that this component is the *source* of the error, *i.e.*, the error originates internally to the component and does not flow into the component from the context, and that the error may impact other components in the context via the `current_temperature` output port. To formalize the reliability information called out in Section 3.1.3 **Step 4**, the occurrence properties provide information concerning the probability of occurrence for each of the errors, which can be leveraged by a variety of probabilistic analyses. For hardware components, such probabilities might be derived from MIL-HDBK-217F *Reliability Prediction of Electronic Equipment* or similar sources.

4.4 Error Propagation, Termination, and Transformation

Common HAT, including FMEA, require as input the possible ways that one component could interact or interfere with another. The example FEMA output in Figure 3 of Section 3.1.2 illustrates that reports following the recommended

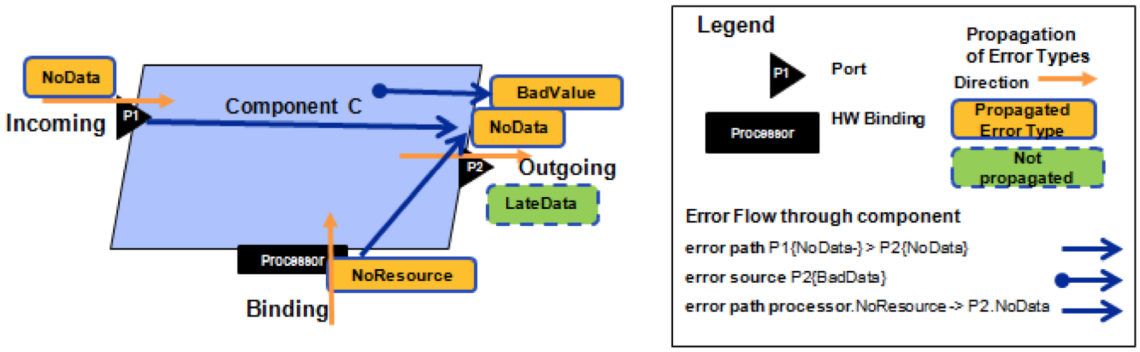

Figure 9: AADL EMV2 Error Propagation (from the AADL Error Model Annex)

```
system implementation isolette.impl
subcomponents
  thermostat : system thermostat_th.impl;
  temperature_sensor : device
    temperature_sensor_ts.impl;
  heat_source : device heat_source_hs.impl;
  operator_interface : system
    operator_interface_oi.impl;
connections
ct : port
  temperature_sensor.current_temperature
    -> thermostat.current_temperature;
  . . .
end isolette.impl;
```

Figure 10: AADL Isolette Component Implementation

format for recording FMEA outputs often have large gaps in justification/reasoning about effects and impact of errors. In particular, the FEMA output lists immediate effects of a failure at the current component boundary and then lists systemic effects. Intuitively, to derive the systemic effect, one would need to reason about how errors propagate through the components in the system, *i.e.*, one would need to reason about error paths. Interactions between components giving rise to propagation paths could be either be intended and be explicit in the architecture (*e.g.*, port connections reflecting read/write relationships between components) or be unintended (*e.g.*, electrical arcing, or electromagnetic interference or heat damage due inadequate shielding). AADL EMV2 provides formal specification of both direct/explicit and indirect/implicit interactions. Thus, it provides a rigorous methodology and formal justification for how one can move from reasoning about the immediate effects of a failure to systemic effects. Here, we limit the discussion to how EMV2 captures explicit interactions.

Figure 9 (taken from from the AADL Error Model Annex specification) illustrates that EMV2 enables one to declare how errors flow along interaction points of components (*e.g.*, features such as ports as well as deployment bindings). For example, the `BadValue` error can originate within the component and propagate out the `P2` port. In contrast, the `NoData` originates in the component's context, flows into the `P1` port and is propagated through the component and out the `P2` port. One can also declare that the component should not produce a `LateValue` error; similar declarations capture the fact that an incoming error is recognized within the component and mitigated in some way. In general, for each component one can specify an error flow indicating whether a component is the source or sink of an error propagation (indicated by `source` and `sink` keywords), or whether it passes

```
  current_temperature : in data port
    Iso_Variables::current_temperature;
  . . .
annex EMV2 {**
  use types ErrorLibrary, isolette;
  error propagations
  current_temperature: in propagation
    {OutOfRange,UndetectableValueError};
  alarm: out propagation {AlarmError};
  monitor_status: out propagation
    {MonitorStatusError};
  regulator_failure: in propagation
    {ItemOmission,ItemComission};
  end propagations;
**};
end thermostat_th;
```

Figure 11: Thermostat declarations illustrating EMV2 error propagations

on an incoming propagation as an outgoing propagation of the same or different error type (indicated by `path` keyword).

Components sometimes transform errors. A component detecting an out-of-range error on an incoming data port can transform it into an omission error (by discarding the data) rather than sending bad data to the next component. In this way, transmission, detection, suppression, and transformation can be modeled. For medical devices, precise characterization of error types and their flows allows risk analysis to focus on those error types that (may) cause patient harm.

Consider the example of explicit interaction (captured in our AADL formal architecture specification of Figure 10) over the connection `ct` that communicates temperature data between the temperature sensor and the thermostat. When reasoning about the fault/failure properties of the thermostat, we wish to indicate how errors coming from the thermostat's system context may propagate or be mitigated by the thermostat. In the EMV2 annotations in Figure 11, we indicate that the port `current_temperature` of the thermostat is prepared to receive propagations of error types `OutOfRange` and `UndetectableValueError`. If other error types such as `TimingError` could be received, the inconsistency will be identified by the EMV2 plugin to OSATE. This illustrates the similarity between error model analysis and error-type-checking in AADL models and conventional static analysis and type-checking on source code. Figure 11 also illustrates that *error type sets* allow multiple error types to be treated together. Error type sets are enclosed by curly brackets. The error types expected on in port `current_temperature` is the error type set `{OutOfRange,UndetectableValueError}`. Similarly, out port `monitor_status` emits the only error type in its set, `Moni-`

torStatusError. Note that the error type OutOfRange includes both AboveRange and BelowRange by inheritance.

Another structuring mechanism, *error products*, enables one to characterize an error in terms of a conjunction of error types. For example: a message may be late; a message may have incorrect value; a message may be both late and incorrect as captured by the respective error types below.
{LateDelivery, ValueError, LateDelivery*ValueError}

An error product (the star) defines a new error type to be the confluence of multiple error types.[4]

To illustrate situations in which errors are transformed, consider the interaction between the temperature sensor and heat source reflected in Figure 12. The declared error path indicates that an incoming UndetectableValueError on the current_temperature port becomes a HeatControlError on the heat_control output port because the bad temperature value was used to control the heat source.

```
flows mrmsve:
  error path current_temperature
    {UndetectableValueError} ->
      heat_control(HeatControlError);
```

Figure 12: Error Flow Path

4.5 Error State Machines

The manner in which a component generates or propagates errors often depends on the error state of components. For example, a completely failed component transmits no errors. EMV2 allows the definition of error state machines, and their association with components. In the Isolette example, a simple error state machine models this notion. When such a state machine is specified, the error propagation behavior of a component (omitted due to space constraints) may be conditioned on specific states in the state machine (see Figure 16).

It is important to understand that an error state machine is not an abstraction of a system's functional implementation; it does not specify, *e.g.*, the transition semantics of error handling routines. One would not generate implementation code from an error state machine. Rather, an error state machine is an analysis artifact that reflects the analyst's understanding of how a component generates and transforms error types in a manner that depends on the operational state of the component. Error state machines model the existence of errors that may not (yet) be detected.

4.5.1 Events

Error events trigger transitions of error state machines. Figure 13 shows the declaration of error event fail.

```
    events fail: error event;
```

Figure 13: Error Event

4.5.2 States

Error states represent the current error behavior of its component. Exactly one error state must be initial.

[4]Error products are not used in the Isolette example, and are included here for completeness.

Figure 14 shows the declaration of two error states, working and failed, of which working is the initial state.

```
    states
      working: initial state;
      failed : state;
```

Figure 14: Error States

4.5.3 Transitions

Error transitions define changes of error state, caused by an error event. Figure 15 says a fail event causes transition from working state to failed state.

```
    transitions
      working -[fail]-> failed;
```

Figure 15: Error State Machine Transition

4.5.4 Three Error State Machines

The EMV2 annex library in isolette.aadl defines three state machines used by components. The simplest error state machine models components that stop when they fail. When component's EMV2 annex subclause includes "use behavior isolette::FailStop;", it means to use the state machine defined in Figure 16 to model its errors.

```
--error state machine for components that
--have out-of range values when failed
  error behavior FailStop
    use types isolette;
    events fail: error event;
    states
      working: initial state;
      failed : state;
    transitions
      working -[fail]-> failed;
  end behavior;
```

Figure 16: Fail-Stop Error State Machine

The next error state machine is used to model components, that have become unreliable, but have not failed outright. The temperature sensor was modeled to sometimes fail producing an out-of-range value, but other-times fails producing incorrect readings, but not so bad as to be out of range. It is these subtle value errors that occur and propagate in an unrecognized fashion through the system that often cause the most problems.

Much of error analysis is trying to predict the likelihood and effect of undetected errors. How can one model the effect of undetected errors to reliably predict their occurrence and effects in deployed systems? Use EMV2 to explicitly model known unknowns!

The following error state machine models both hard failures (upon hardfail, transition from working to failed), and

```
--error state machine for components that
--may put out undetectable value errors
  error behavior FailSubtle
    use types isolette;
    events
      hardfail: error event;
      subtlefail: error event;
    states
      working: initial state;
      failed : state;
      flakey: state;
    transitions
      working -[hardfail]-> failed;
      working -[subtlefail]-> flakey;
  end behavior;
```

Figure 17: Fail-Subtle Error State Machine

subtle failures (upon softfail, transition from working to flakey).

For components having subcomponents, error models may define their error state in terms of the error states of subcomponents. Figure 18 shows an error machine used when defining composite component behavior.

```
error behavior CompositeFailure
  use types isolette;
  states
    Operational: initial state;
    ReportedFailure: state {DetectedFault};
    MissedFailure: state {MissedAlarm};
    FalseAlarm: state {FalseAlarm};
end behavior;
```

Figure 18: Composite Error State Machine

4.6 Error Flows

AADL includes the notion of *flow* specification that specifies, *e.g.*, that values on a particular input port flow into (are used to calculate the output value) an output. In a similar fashion, error flows specify relationships between input and output ports. However, instead of describing how values are propagated between ports, error flows describe how errors propagate within components. Thus, error flows capture *intra-component* flows. Error propagation between components (*i.e.*, *inter-component* flow) follows architectural connections (*e.g.*, if output port O on component A is connected to input port I on component B, an error may propagate between A and B along this connection). [5]

Error flows may be:

source origin of an error (fault or hazard)

sink detection and/or suppression of incoming error

path transmission of error through component

4.6.1 *Error Sources*

Error sources model hazard occurrence, deviation in some way from intended behavior.

[5]EMV2 has a way to express error flows between components that don't share an explicit connection.

Figure 19 comes from the `temperature_sensor` device stating that `OutOfRange` or `UndetectableValueError` may occur, and will be emitted by the `current_temperature` port.

```
f: error source current_temperature
  {OutOfRange,UndetectableValueError};
```

Figure 19: Error Source

4.6.2 *Error Sinks*

Error sinks explicitly state that incoming errors are not further propagated.

Any `OutOfRange` error arriving at port `current_temperature` is detected and suppressed.

```
mmmoor: error sink current_temperature
  {OutOfRange};
```

Figure 20: Error Sink

4.6.3 *Error Paths*

An error path describes how errors flow through components, possibly being transformed into a different type of error.

Either an `ItemOmission` or `ItemComission` error arriving at either port `interface_failure` or `internal_failure` will be transmitted by port `monitor_mode` as a `MonitorModeError`.

A `UndetectableValueError` arriving at port `current_temperature` is also transmitted by port `monitor_mode` as a `MonitorModeError`.

```
mmmiff: error path interface_failure
  {ItemOmission,ItemComission}
      -> monitor_mode(MonitorModeError);
mmminf: error path internal_failure
  {ItemOmission,ItemComission}
      -> monitor_mode(MonitorModeError);
mmmct: error path current_temperature
  {UndetectableValueError}
      -> monitor_mode(MonitorModeError);
```

Figure 21: Error Paths

4.7 Error Detection

Sometimes error models need to cause (or at least influence) behavior, such as when errors are detected by hardware. The Isolette model has a `detect_monitor_failure` device component. Figure 22 shows putting out boolean `internal_failure` signal when in the `failed` state.

4.8 Error Properties

EMV2 annex subclauses may have their own properties. EMV2 properties use core AADL property grammar. The

```
component error behavior
  detections
    failed -[ ]-> internal_failure!;
  --in "failed" state send event out
  --port internal_failure
end component;
```

Figure 22: Error Detection

standard EMV2 property set in EMV2.aadl defines many useful properties for error models.

Figure 23 shows the `detect_monitor_failure` device component type with its EMV2 annex subclause which

- uses types from the standard `ErrorLibrary.aadl`, namely `ItemOmission`
- uses the `FailStop` error state machine shown in Figure 16
- propagates `ItemOmission` from its `internal_failure` port for false-negatives (failing to indicate a problem when it exists)
- does not emit `ItemComission` (false-positives)
- sources the `ItemOmission` errors emitted by its `internal_failure` port, and
- signals `internal_failure` when in failed state

The properties hold the quantitative values of error models.

4.8.1 Occurrence Distribution

The likelihood of errors is defined using `EMV2::OccurrenceDistribution` properties.

In Figure 23 `fail` is the error event in `FailStop` that triggers transition from `working` to `failed` indicated by port `internal_failure`; and `dmf` is source for `ItemOmission` errors. The occurrence distributions for `fail` and `dmf` are conveniently collected into the `Iso_Properties` property set shown in Figure 24.

Figure 25 shows the beginning of standard, predeclared property set EMV2.aadl. For `Fixed` probability distributions, only the probability value need be specified. The other values of a `DistributionSpecification` are needed by more complex probability distributions.

4.8.2 Occurrence Probability

Error occurrence probability is specified with the `ProbabilityValue` in `EMV2::DistributionSpecification`. Figure 24 shows the probability values for monitor failure rate, and detection of monitor failure.

4.8.3 Hazards

As indicated above, error sources are hazards. Figure 23 specifies `EMV2::Hazard` property for error source `dmf.ItemOmission`.

Figure 26 shows the `Hazard` property defined in EMV2.aadl.

4.8.4 Severity and Likelihood

```
device detect_monitor_failure
features
  internal_failure : out data port
    Base_Types::Boolean
    {BLESS::Assertion =>
      "<<INTERNAL_FAILURE()>>";};
annex EMV2
{**
  use types ErrorLibrary;
  use behavior isolette::FailStop;
error propagations
  internal_failure: out propagation
  {ItemOmission};
  internal_failure: not out propagation
    {ItemComission};
flows
  dmf: error source internal_failure
    {ItemOmission};
end propagations;
component error behavior
  detections
    failed -[ ]-> internal_failure!;
end component;
properties
--failure rate for temp monitor
  EMV2::OccurrenceDistribution =>
    Iso_Properties::MonitorFailureRate
      applies to fail;
--rate of detection failure
  EMV2::OccurrenceDistribution =>
    Iso_Properties::DetectionMonitorFailureRate
      applies to dmf;
--definition of hazard causing failure
  EMV2::Hazard =>
    [ crossreference => "REMH A.5.2.4";
    failure => "monitor failure w/o report";
    phase => "all";
    description => "monitor failure missed";
    comment => "not detecting monitor failures
    loses mitigation of heat control errors";
    ] applies to dmf.ItemOmission;
  ARP4761::Severity => Hazardous
    applies to dmf.ItemOmission;
  ARP4761::Likelihood => ExtremelyImprobable
    applies to dmf.ItemOmission;
**};
end detect_monitor_failure;
```

Figure 23: Detect Monitor Failure Function

```
--rate at which temp monitor fails
  MonitorFailureRate : constant
    EMV2::DistributionSpecification =>
    [ProbabilityValue => 1.6E-7;
     Distribution => Fixed;];
--error rate of detecting monitor failure
  DetectionMonitorFailureRate : constant
    EMV2::DistributionSpecification =>
    [ProbabilityValue => 1.7E-10;
     Distribution => Fixed;];
```

Figure 24: Monitor Failure Rate Properties

EMV2 provides predeclared property sets for both MIL-STD-882 *System Safety Program Requirements/Standard Practice for System Safety* and ARP 4761 *Guidelines and Methods for Conducting Safety Assessment Process on Civil Airborne Systems and Equipment* to declare severity and likeli-

```
property set EMV2
  is
OccurrenceDistribution :
  EMV2::DistributionSpecification
    applies to (all);
DistributionSpecification : type record (
  ProbabilityValue : aadlreal;
  OccurrenceRate : aadlreal;
  MeanValue : aadlreal;
  StandardDeviation : aadlreal;
  ShapeParameter : aadlreal;
  ScaleParameter : aadlreal;
  SuccessCount : aadlreal;
  SampleCount : aadlreal;
  Probability : aadlreal;
  Distribution : EMV2::DistributionFunction;);
DistributionFunction : type enumeration
  (Fixed, Poisson, Exponential,
   Normal, Gauss, Weibull, Binominal);
```

Figure 25: EMV2 Occurrence Distribution Property

```
Hazard: record
  (crossreference: aadlstring;
    failure : aadlstring;
    phase : aadlstring;
    environment : aadlstring;
    description : aadlstring;
    verificationmethod : aadlstring;
    risk : aadlstring;
    comment : aadlstring;) applies to (all);
```

Figure 26: Hazard Property

hood for hazards. Figure 27 shows the property set for ARP 4761 in ARP4791.aadl.

```
property set ARP4761
  is
Severity : inherit enumeration
  (Catastrophic, Hazardous, Major, Minor,
   NoEffect) applies to (all);
Likelihood : inherit enumeration
  (Probable, Remote, ExtremelyRemote,
   ExtremelyImprobable) applies to (all);
end ARP4761;
```

Figure 27: ARP 4761 Severity and Likelihood

In Figure 23, the ARP 4761 severity for failing to report monitor failure was rated `Hazardous`; its likelihood was rated `ExtremelyImprobable`.

Certainly, the labels used for ARP 4761 likelihood need to be consistent with probability values. The detection monitor failure rate probability value (Figure 24) is `1.7E-10` which corresponds to `ExtremelyImprobable`: $p < 10^{-9}$.

Whereas likelihood has mathematical specificity, severity is inherently subjective and discontinuous. ARP 4761 defines severity levels Catastrophic, Hazardous, Major, Minor, NoEffect.

The notion of Risk Priority Number (RPN) as currently practiced in Failure Modes, Effects, and *Criticality* Analysis gives failure severity a number, and then multiplies by failure likelihood (then divides by "likelihood of detection") to determine risk priority. This notion has a major, even unacceptable, limitation. It models a continuous function that can bury, or hide, more severe categories by less severe categories. For instance, a Catastrophic error that results in death (criticality = 5) might be quite infrequent and highly detectable so that its RPN is a lower value than a more frequent, less detectable Hazardous or Major error that results in injury or pain (criticality = 3 or 4); this situation, which is not unusual, would obscure the Catastrophic error that results in death. Severity, or criticality, for medical devices is not continuous, it has step changes between categories. Each error within Catastrophic or Hazardous categories should be examined with its own separate FMEA.

Consequently, RPN and quantifying severity commits category error; some things cannot be measured by a single number: Instead, risk analysis must treat each class of severity distinctly, with greatest interest in errors that can kill. Proper classification of hazards is thus supremely important and should be performed by the most senior and experienced system engineers in conjunction with domain experts.

5. EMV2 ANALYSIS TOOLS

The EMV2 plug-in to OSATE comes with several analysis tools. They analyze and process EMV2 descriptions attached to an AADL architecture for evaluating system safety or reliability. They produce either document or evidence of software faults or defects.

The following section gives an overview of the toolset and its capabilities but does not provide guidance for using the tools. Users that are looking for help and support may refer to OSATE help [17] or the SEI technical report about the specific Error-Model Toolset [2],

5.1 Instance vs Declarative Models

Although people write text expressing declarative models, all EMV2 analysis tools process instance models. Understanding the difference is crucial to using these EMV2 tools. Instance models are generated from declarative models by OSATE. For very simple systems, declarative and instance models can be practically identical when there is only one design choice. The distinction between declarative and instance models can be hard to grasp, but once understood seems obvious.

Declarative models can include many design options. In the textual, declarative model, components may have several levels of subtyping; particular component types may have more than one implementation; components may use AADL's prototypes to describe a class of designs using parametric polymorphism; component implementations may have arrays of subcomponents; component features may be arrays of ports.

Instance models are built from declarative models and represent a particular design, out of possibly many. To generate an instance model from a declarative model, a component implementation is selected. The instantiation function then resolves all the polymorphisms into concrete types, and creates a data structure for each component in the system. If component arrays are used, a data structure for each element is created.

Figure 28 shows a graphical view of the architecture instance of the Isolette model defined in Figure 29. It shows the system components (`temperature_sensor`, `operator_inter-`

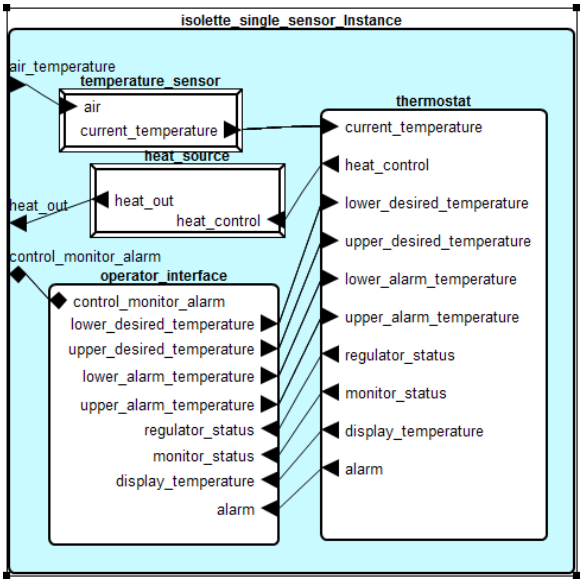

Figure 28: Instance Model of the Isolette Model

face, heat_source, thermostat) and their interconnection. Such representation is provided by the Instance Model Viewer (IMV) within OSATE [17].

5.2 Graphical View of Error Impacts

The Instance Model Viewer (IMV) of OSATE provides graphical support for representing AADL instance models with their associated error annotations. When selecting a component, the tool highlights its impacts when failing. Depending on the connection level between components, the color brightness changes (a dark color means that the component is more likely to be impacted by the selected component). An example of this function is shown in figure 30: components in dark red (operator_interface and heat_source) are impacted by any fault occurring within the temperature_sensor. While it does not distinguish the different error types within the architecture, this function provides an overview of the impact of a failure of a component.

5.3 Consistency Checks

The OSATE toolset provides a function to check the consistency of the error declaration against other architecture artifacts. For example, below are some of the rules that are enforced by the consistency check:

- an error sink cannot be used as a propagation condition for propagating other errors through the component features.
- switching from one error state to another can be triggered only when receiving an error on an error sink.
- in an error transition, all error events and incoming error propagations must be referenced
- two component error behavior transitions cannot have the same condition

All consistency checks are listed in [2] related to the Error-Model Annex and its OSATE support [17].

```
system implementation isolette.single_sensor
subcomponents
  thermostat : system
    thermostat_single_sensor.impl;
  temperature_sensor : device
    Devices::temperature_sensor.impl;
  heat_source : device
    Devices::heat_source.impl;
  operator_interface : system
    operator_interface.impl;
connections
  ct : port
    temperature_sensor.current_temperature
    -> thermostat.current_temperature;
  hc : port thermostat.heat_control
    -> heat_source.heat_control;
  ldt : port
    operator_interface.lower_desired_temperature
     -> thermostat.lower_desired_temperature;
  udt : port
    operator_interface.upper_desired_temperature
    -> thermostat.upper_desired_temperature;
  lat : port
    operator_interface.lower_alarm_temperature
    -> thermostat.lower_alarm_temperature;
  uat : port
    operator_interface.upper_alarm_temperature
    -> thermostat.upper_alarm_temperature;
  rs : port thermostat.regulator_status
    -> operator_interface.regulator_status;
  ms : port thermostat.monitor_status
    -> operator_interface.monitor_status;
  dt : port thermostat.display_temperature
    -> operator_interface.display_temperature;
  al : port thermostat.alarm
    -> operator_interface.alarm;
annex EMV2
  {**
  use types ErrorLibrary, Isolette;
  use behavior CompositeFailure;
  composite error behavior
  states
  [temperature_sensor.failed
    or thermostat.ReportedFailure
    or heat_source.failed]->ReportedFailure;
  [temperature_sensor.flakey
   or thermostat.MissedFailure]->MissedFailure;
  end composite;
  **};
end isolette.single_sensor;
```

Figure 29: Declarative Text Model of Single-Sensor Isolette

This function also looks at all connections and checks for incoming/outgoing specification consistency. If any error types from the out propagation are not included in the error types expected by the incoming propagation, the tool reports an error. This function highlight potential defects and helps one understand if the architecture handles all potential faults. This is of particular interest when integrating components from different models and development teams.

For example, considering that a component is an error source for error types UndetectableValueError and ItemOmission errors, changing the receiver specification from

```
display_temperature: in propagation
   {UndetectableValueError,ItemOmission};
```

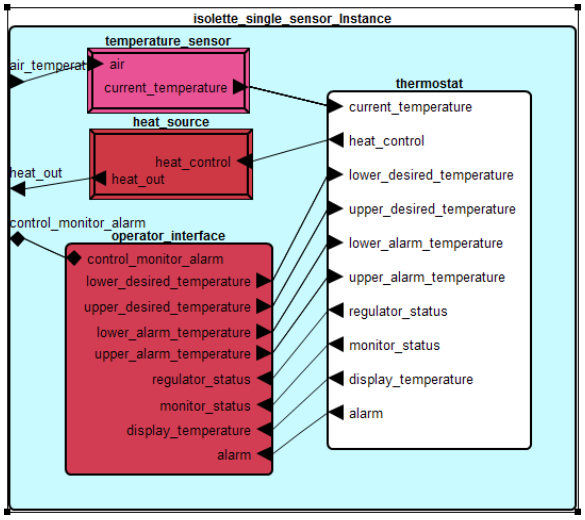

Figure 30: Highlight errors in the Instance Model

to

```
display_temperature: in propagation
   {UndetectableValueError};
```

will trigger an consistency error because the `ItemOmission` type is no longer handled by the receiver.

The toolset will then report an error such as: *"Outgoing propagation displayed_temp {UndetectableValueError, ItemOmission} has error types not handled by incoming propagation display_temperature {UndetectableValueError}".*

5.4 Fault Hazard Assessment

A conventional Functional Hazard Assessment (FHA) is an exhaustive enumeration of all failure conditions in the architecture. It lists all error contributors and reports their associated information (likelihood, description, severity, etc.). Such a document identifies all potential sources of errors and is required by safety validation standard such as ARP4761 [16].

OSATE supports the production of FHA reports. To do so, it analyzes the architecture, extracts all potential error contributors (`error source`, `error event`) and their associated documentation (EMV2 properties such as `EMV2::Hazards`, `EMV2::Severity` and `EMV2::Likelihood`) and generated a spreadsheet that aggregates all this information. A complete description of this function is described in [2]. Figure 31 shows an excerpt of the Isolette Fault and Hazard Assessment report. For each component, it lists the error sources and their associated information (error, cross-reference to other documents, effect, etc.).

5.5 Fault-Tree Analysis

A conventional Fault-Tree Analysis (FTA) is a graphical representation of the contributor to a failure state. This is a top-down approach, showing the resulting error state at the top and listing all its contributors within the architecture. It makes use of the tree notation to show the dependency between error events. Such analysis is especially valuable when analyzing a system and inspecting all components that may contribute to a system failure. Such a technique is required by safety evaluation process such as ARP4761 [16].

OSATE supports the automatic generation of FTAs, constructed by analyzing the architecture and error propagation paths. Figure 32 presents an example of an FTA for the Isolette. The top-level element represents the error state under investigation while the other nodes represent contributing error states/events. OSATE supports the generation of FTA for both open-source (such as OpenFTA [15]) and commercial tools (such as CAFTA).

5.6 Fault Impact

OSATE provides a Fault Impact Analysis that automatically traces the error paths from error sources to affected components. Contrary to the Fault-Tree Analysis, this is a bottom-up approach that lists all impacted components for each error source. While the resulting report document can be quite long, it may be useful when considering faults of a particular component and documenting their impact on the overall architecture. Such a document is similar to FMEA (Failure Mode and Effects Analysis) or FMES (Failure Modes and Effects Summary) required by safety-validation standards, such as ARP4761 [16].

Table 1 shows an extract of the Fault Impact document generated from the Isolette AADL model, depicting the fault impact of `OutOfRange` errors from the `temperature` sensor. The first (non-header) indicates that when the temperature sensor has failed such that its current temperature value is out of range, that error will be received by the `manage_monitor_mode` thread in the `monitor_temperature` process, where it will be detected and masked. The document then contains the list of all error sources and their propagations across the overall architecture.

5.7 Reliability Block Diagram

A Reliability Block Diagram (RBD) provides the reliability value for a using the system components and their relationships. Each component is treated as an isolated block that has a designated reliability or failure rate value. The computation of the reliability measure depends on the na-

	A	B	C	D	E	F	G
1	Component	Error	Crossreference	Functional Failure (Hazard)	Environment	Effects of Hazard	Comment
2	temperature_sensor	"hardfail"	"REMH A.3.2"	"total failure"	"infant intensive care"	"temperature sensor breaks"	"easily detected and alarm issued"
3	temperature_sensor	"subtlefail"	"REMH A.3.2"	"bad value"	"infant intensive care"	"temperature sensor out of calibration"	"undetectable"
4	heat_source	"fail"	"REMH A.3.2"	"heat source breaks"	"infant intensive care"	"mechanical disconnection of heat source"	"always fails open (off)"

Figure 31: Example of the Fault Hazard Assessment for the Isolette Model

Table 1: Single-Sensor Out-of-Range Fault Impact Analysis

Component	Initial Failure Mode	1st Level Effect	Failure Mode
temperature_sensor	failed	current_temperature {OutOfRange} -> monitor_temperature/ manage_monitor_mode	monitor_temperature/ manage_monitor_mode.current_temperature {OutOfRange} [Masked]
temperature_sensor	failed	current_temperature {OutOfRange} -> regulate_temperature/ manage_regulator_interface	regulate_temperature/ manage_regulator_interface.current_temperature {OutOfRange} [Masked]
temperature_sensor	failed	current_temperature {OutOfRange} -> regulate_temperature/ manage_heat_source	regulate_temperature/ manage_heat_source.current_temperature {OutOfRange} [Masked]
temperature_sensor	failed	current_temperature {OutOfRange} -> regulate_temperature/ manage_regulator_mode	regulate_temperature/ manage_regulator_mode.current_temperature {OutOfRange} [Masked]

ture of component interactions (connection in series, parallel, etc.). A complete description of the algorithm is presented in [2]). Such a notation is used by several safety evaluation standards, such as ARP4761, which refer to this representation as a Dependency Diagram (DD).

OSATE supports the generation of a reliability report by computing the reliability value for an error state. The tool does not generate the graphical notation of the RDB but provides the reliability value using failure probability of AADL components. A complete description of this feature is detailed in [2].

5.8 Markov Chain Analysis

A Markov model (a.k.a. chain) represents a system behavior with its states and transitions. It also assigns a probability to each transition. Then, dedicated tools can process this notation, simulate the system behavior or analyze it to evaluate the probability for being in a particular state. Such analysis is required by safety evaluation standards such as ARP4761 [16] and is especially useful to validate a component reliability (probability that a component fails is less than a fixed given value).

OSATE transforms AADL error models into Markov models so that safety engineers use the architecture notation to evaluate its safety. For now, produced Markov models can be used with PRISM [10], an open-source tool for analyzing Markov Chain model. A complete description of that function is included in [2].

6. CONCLUSION AND FUTURE WORK

We believe that risk assessments and safety analyses too often end up being inconsistent and tedious to perform, and that opportunities are missed to reuse and leverage information across different techniques. Moreover, overall accuracy and the ability to to establish rigorous traceability to is hindered by the fact that techniques are not directly integrated artifacts directly tied to implementations.

Modeling in AADL and EMV2 provides engineers of high-integrity systems with techniques and tools that can enable a more rigorous, automated, and integrated approach to important risk management activities. The formalization by EMV2 of error models enables better support for automation of hazard analysis techniques, and ensures that all such analyses apply to the same, single-source of truth. The integration with formal architectural descriptions written in AADL enables strong traceability to artifacts directly tied to implementations (especially in situations where implementation source code for system interfaces is automatically generated from a system's AADL-based architectural specification). When we have presented this approach to software developers, many have indicated that similarities and analogies to code-level annotation-based tools, static analysis, and type checking make the AADL EMV2-based approach much more attractive than conventional approaches.

In this paper, we have illustrated basic aspects of EMV2 on a simple medical device and drawn connections to standard techniques such as FMEA and FTA. Two major directions for future work include (a) applying EMV2 to systems of greater scale and complexity, and (b) illustrating how AADL EMV2 supports other safety/risk-related analyses. Regarding (a), engineers at SEI are applying EMV2 to more complex avionics systems, and illustrating the use of EMV2 in large-scale avionics integration is an emphasis in ongoing work on the SAVI project. In our own research group and in collaborations with FDA engineers, we are applying EMV2 to AADL models of a realistic infusion pump [11]. We are also laying the groundwork for application of AADL and EMV2 to specifying the ASTM 2761 standard

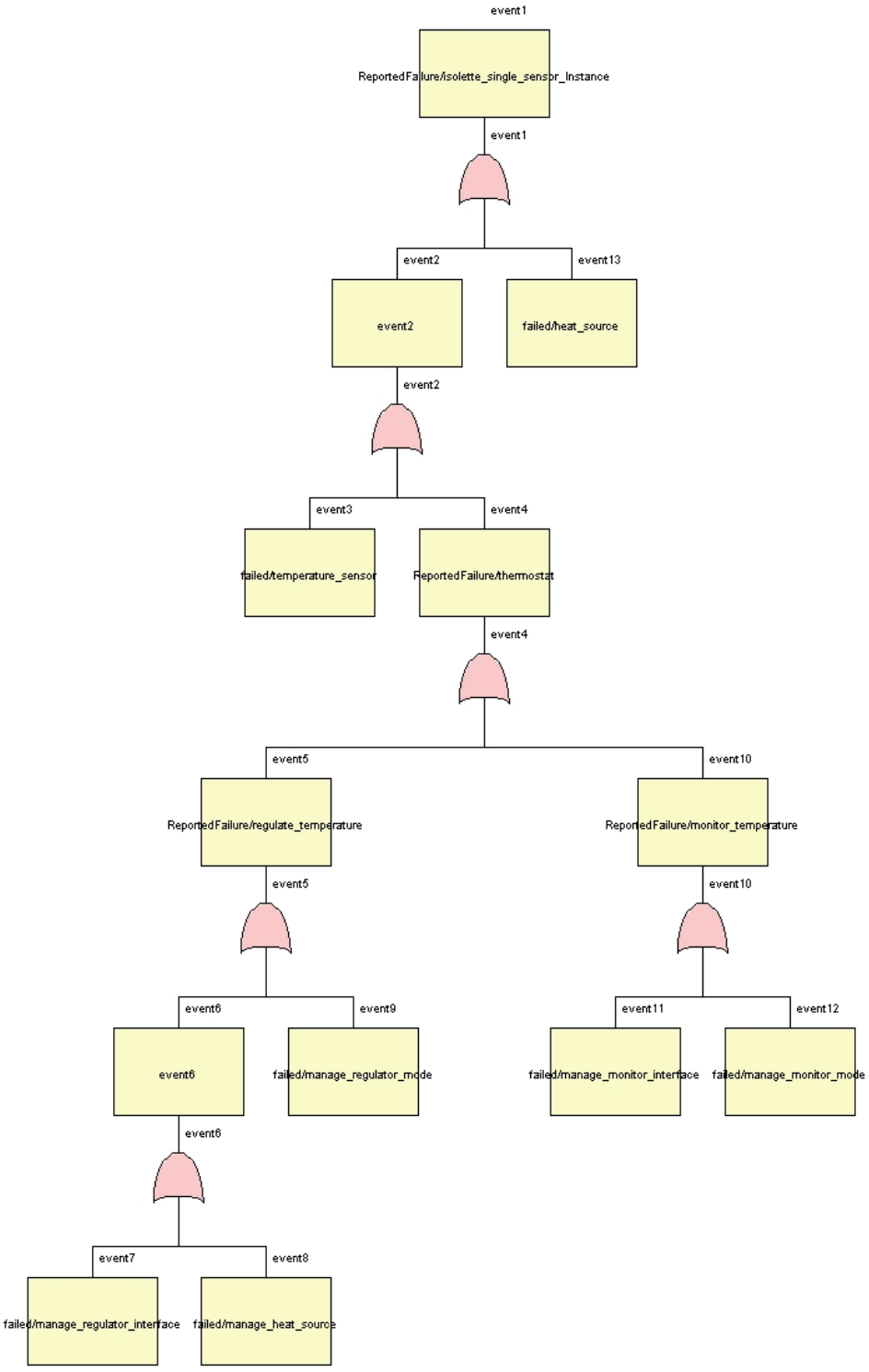

Figure 32: Example of the Fault-Tree for the Isolette Model

Integrated Clinical Environment (ICE) architecture to support development, safety certification, and FDA regulatory review of medical application platforms [9]. These later activities are also supporting efforts on the UL/AAMI Joint Committee on Medical Device Interoperability to develop safety standards for systems of interoperable medical devices.

Regarding (b), since AADL and EMV2 are being used on avionics projects such as SAVI, it seems worthwhile to expand the discussion on how EMV2 supports various safety-related analyses to include a broader of overview of how the methodology and techniques in ARP 4761 could be supported.

Acknowledgements

This work is supported in part by the US National Science Foundation (NSF) (#0932289, #1239543), the NSF US Food and Drug Administration Scholar-in-Residence Program (#1065887, #1238431) the National Institutes of Health / NIBIB Quantum Program, and the US Air Force Office of Scientific Research (AFOSR) (#FA9550-09-1-0138). The authors wish to thank Peter Feiler from SEI and engineers from the US Food and Drug Administration for feedback on this work. Peter Feiler is the author of the EMV2 annex, and his work provides the foundation for the example-based presentation of this paper.

7. REFERENCES

[1] Architecture Analysis & Design Language. www.aadl.info, 2012.

[2] J. Delange, P. Feiler, D. Gluch, and J. Hudak. AADL fault modeling and analysis within an ARP4761 safety assessment. Technical report, Carnegie Mellon Software Engineering Institute, 2013.

[3] E. S. Dominique Blouin, Skander Turki. AADL requirements annex (draft, progress update). |https://wiki.sei.cmu.edu/aadl/images/a/af/Requirements_annex_aadl_standards_meeting_16-19_04_2012.pdf|.

[4] C. A. Ericson. *Hazard Analysis Techniques for System Safety*. Wiley-Interscience, 2005.

[5] P. Feiler. *Architecture Analysis and Design Language (AADL) Annex Volume 3: Annex E: Error Model V2 Annex*. Number SAE AS5506/3 (Draft) in SAE Aerospace Standard. SAE International, 2013.

[6] P. Feiler and D. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language*. Addison-Wesley, 2012.

[7] P. H. Feiler, J. Hansson, D. de Niz, and L. Wrage. System architecture virtual integration: An industrial case study. Technical Report CMU/SEI-2009-TR-017, CMU, 2009.

[8] P. Fenelon and J. A. Mcdermid. An integrated toolset for software safety analysis. *Journal of Systems and Software*, 21:279–290, 1993.

[9] J. Hatcliff, A. King, I. Lee, A. Fernandez, A. McDonald, E. Vasserman, and S. Weininger. Rationale and architecture principles for medical application platforms. In *Proceedings of the 2012 International Conference on Cyberphysical Systems*, 2012.

[10] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[11] B. R. Larson, J. Hatcliff, and P. Chalin. Open source patient-controlled analgesic pump requirements documentation. In *Proceedings of the International Workshop on Software Engineering in Healthcare*, San Francisco, May 2013.

[12] D. Lempia and S. Miller. DOT/FAA/AR-08/32. Requirements Engineering Management Handbook, 2009.

[13] N. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[14] N. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press, 2012.

[15] O-Sys. OpenFTA - http://www.openfta.com, 2013.

[16] SAE International. *ARP4761 - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, 1996.

[17] SEI/CMU. Open Source AADL Tool Environment (OSATE) - `https://wiki.sei.cmu.edu/aadl`, 2013.

[18] M. Wallace. Modular architectural representation and analysis of fault propagation and transformation. In *Proc. FESCA 2005, ENTCS 141(3), Elsevier*, pages 53–71, 2005.

[19] System Architecture Virtual Integration (SAVI) Initiative. `https://wiki.sei.cmu.edu/aadl/index.php/Projects_and_Initiatives#AVSI_SAVI`wiki.sei.cmu.edu/aadl/index.php/Projects_and_Initiatives, 2012.