Software Engineering Institute
Carnegie Mellon University

Architecture Fault Modeling and Analysis with the Error Model Annex, Version 2

Peter Feiler
John Hudak
Julien Delange
David P. Gluch

June 2016

SPECIAL REPORT
CMU/SEI-2016-TR-009

Software Solutions Division

http://www.sei.cmu.edu

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

This report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

DM-0003594

Table of Contents

List of Figures

List of Tables

Executive Summary

Safety-critical software-reliant systems must manage component failures and previously unidentified conditions of anomalous interaction among components as hazards that affect a system's safety, reliability, and security so that the potential effects of residual hazards on the system operation are reduced to an acceptable risk. Standards and recommended practices for safety-critical systems—such as DO-178B/C, SAE ARP4754A, and SAE ARP4761 in the aerospace industry—outline methods such as Functional Hazard Assessment, Failure Mode and Effect Analysis, Fault Tree Analysis, and availability and reliability prediction via reliability block diagrams. Security-related practices are typically addressed through separate guidance.

This report provides guidance on the use of the Error Model Annex, Version 2 (EMV2), notation [SAE 2015], a revision of the SAE AS-5506/1 Error Model standard for architecture fault modeling and analysis [SAE 2006]. EMV2 augments architecture models expressed in the Architecture Analysis & Design Language (AADL) with fault information to characterize anomalous conditions. Automated safety, reliability, and security analyses from the same annotated architecture model ensure consistency across analysis results.

The report introduces EMV2 concepts for architecture fault modeling of systems consisting of components in the context of an operational environment in terms of three levels of abstraction:

- focus on fault sources in a system and their impact on other components or the operational environment through propagation
- focus on a system or component fault model to identify faults and their occurrences within a system (component), their manifestation as failure, the effect of incoming propagations, conditions for outgoing propagation, and the ability of the system (component) to recover or be repaired
- focus on relating the fault model of system components to the abstracted fault model of the system

This layered abstraction allows for scalable compositional analysis.

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

EMV2 allows users to specify which system components are expected to detect, report, and manage anomalous conditions and their propagation and reflect the effects of recovery and repair actions taken by the system on the error behavior state. The implementation of health monitoring and fault management functionality of the system is modeled in the AADL core model. In other words, EMV2 expresses anomalous behavior of systems independent of whether and how it is actually managed by the system.

The report includes a discussion of several example models:

- a Global Positioning System with a focus on the interaction between operational and failure modes
- a dual-redundant flight guidance system to show consistency between an abstracted architecture fault model and its composite model for a fault-tolerant system
- an example of abstractly specifying a fault behavior interface for a network protocol stack
- an example of specifying an abstract fault model for a dual-channel avionics network to ensure correct and consistent use
- a triple-redundant system that has both physical and logical redundancy

Abstract

Safety-critical software-reliant systems must manage component failures and conditions of anomalous interaction among components as hazards that affect a system's safety, reliability, and security so the potential effects of hazards on system operation are reduced to an acceptable risk. Standards and recommended practices for safety-critical systems outline methods for analysis, but security-related practices are typically addressed through separate guidance. This report provides guidance on using the Error Model Annex, Version 2 (EMV2), notation for architecture fault modeling and analysis, which supports automated safety, reliability, and security analyses from the same annotated architecture model to ensure consistency across analysis results. EMV2 augments architecture models expressed in the Architecture Analysis & Design Language with fault information to characterize anomalous conditions. The report introduces concepts for architecture fault modeling of systems in an operational environment at three levels of abstraction. In addition, EMV2 introduces the concept of error types to characterize exceptional conditions and their propagation. Finally, EMV2 allows users to specify which system components are expected to detect, report, and manage anomalous conditions and their propagation and to reflect the effects of recovery and repair actions as error behavior states. The report includes several example models.

1 Introduction

This report provides a guide to architecture fault modeling using the SAE International Architecture Analysis & Design Language (AADL) [SAE 2012] and the Error Model Annex, Version 2 (EMV2), standard [SAE 2015]. The resulting models are the basis for various forms of safety and dependability analysis. A separate report discusses how architecture fault models support the ARP4761 safety analysis practice, illustrated with a wheel braking system [Delange 2014].

1.1 Background

Development efforts for safety-critical software-reliant systems must manage component failures and previously unidentified conditions of anomalous interaction among components as hazards that affect reliability, safety, and security so that the potential effects of residual hazards on the system operation are reduced to an acceptable risk. Reliability focuses on providing continued operation despite failures. Safety focuses on unsafe conditions due to failures, malfunctions, or unexpected interactions between system components and the environment that result in catastrophic consequences for human life, health, property, or the environment. Security focuses on the protection of systems from accidental or malicious access, use, modification, destruction, or disclosure.

Standards and guidance documents for safety-critical systems—such as DO-178B/C, SAE ARP4754, and SAE ARP4761 in the aerospace industry—outline recommended practices such as Functional Hazard Assessment (FHA), Preliminary System Safety Assessment (PSSA), full System Safety Assessment (SSA), and Common Cause Failure Analysis. Other methods include Failure Modes and Effect Analysis (FMEA) and its variant Failure Mode, Effects, and Criticality Analysis; Fault Tree Analysis (FTA); availability and reliability prediction via reliability block diagrams and stochastic Petri net or Markov models; and architectural designs such as using system partitioning as a protection scheme for fault isolation and containment (DO-178B/C and ARINC-653).

These methods are labor intensive and, therefore, are often applied once in the life of a system. For example, when a development team performs an FHA, they record only hazards that result in catastrophic failure. Later when performing an FMEA, experts explore failure modes and one or two levels of effects. Due to the large effort involved, tradeoffs between alternative designs are often not examined using an FMEA. Consistency between FHA, FMEA, FTA, and other analyses is typically maintained by inspection of documents that capture the results of these analyses, another time-consuming and fallible activity. Alternatively, by generating safety analysis representations, such as fault trees, from AADL models annotated with EMV2, we ensure that architectural changes are consistently propagated and reflected in analysis reports when repeated. This approach maintains consistency across analyses beyond safety to include performance, security, and other objectives.

While these practices typically focus on safety and reliability, the underlying concepts of hazards and their propagation can also be used to identify security concerns that potentially result in intrusion and violation of information confidentiality. Partitioned architecture designs have been promoted as a key to Multiple Independent Levels of Security/Safety (MILS) [Rushby 1981], which is reflected in the Common Criteria for Information Technology Security Evaluation [ISO 2005].

As safety-critical systems have become more software reliant, they have also experienced major cost increases. Software systems can make up more than 75% of the system cost, and as much as 70% of that is the cost of rework due to errors introduced in requirements and architecture design but not discovered until system integration [NIST 2002]. Several root causes with system-wide impact have emerged, as evident from the increased fault leakage from requirements and architecture design into system integration and later into the system life cycle [Feiler 2010]. This fault leakage is due to mismatched assumptions in the interfaces among software components, between software components, and between the hardware platform and the physical system, in particular with respect to nonfunctional properties. Root causes of mismatched assumptions among system components include the following:

- base type and abstract data type representations, including inconsistent use of measurement units and value ranges
- real-time processing of time-sensitive data streams and the impact of latency jitter, mismatched streaming rates, and dropped or corrupted data elements on continuous control system behavior and discrete system state interaction
- synchronizing and coordinating redundant processing streams, in particular detection and recovery logic for anomalous system behavior such as Byzantine failure behavior
- interactions between state-based systems with replicated, mirrored, and coordinated state machines representing operational and failure modes (mode confusion)
- impact of state vs. state change communication under faulty transfer conditions, and the use of sampled processing to communicate events, leading to possible loss of events (inconsistent state)
- performance impact of resource management due to mismatched resource demand and capacity and unmanaged shared resource usage
- virtualization of resources, such as partitioned architectures, resulting in potential inconsistency of logical and physical redundancy and affecting reliability and availability
- virtualization of time, resulting in temporal inconsistencies and potential loss of information

1.2 Virtual System Integration and Architecture Fault Modeling

The SAE AADL standard provides a notation for specifying the architecture of software-reliant systems [Feiler 2012, SAE 2012]. It introduces component concepts that are specific to the architectures of software systems:

- packages, data components, subprograms, subprogram groups, and system and abstract components to specify a functional or software design architecture
- processes and threads with port connections, remote service calls, and shared data access to specify the runtime architecture of a software system
- processors, virtual processors, memory, buses, virtual buses, and their interconnections via bus access to specify a hardware platform
- devices, buses, and systems with both logical and physical connections to specify interactions with the physical system

Well-defined execution semantics, communication timing semantics, and standardized extensions allow for qualitative and quantitative analyses of multiple quality attributes of a multitier system

architecture from the same architecture model. Furthermore, they can be analyzed incrementally as the architecture hierarchy is refined and evolved. The aerospace industry's System Architecture Virtual Integration (SAVI) initiative has demonstrated the value of virtual integration and predictive analysis of architecture models for discovering system-level issues early in the development process, thus reducing the cost of major rework [Redman 2010]. These results led to an approach to system quality certification and improvement that combines predictive analysis based on virtual integration with increased formalization of requirements, increased use of static analysis such as model checking, and assurance cases to manage evidence that the system design and implementation have met functional and operational quality requirements [Feiler 2013]. Figure 1 illustrates this approach.

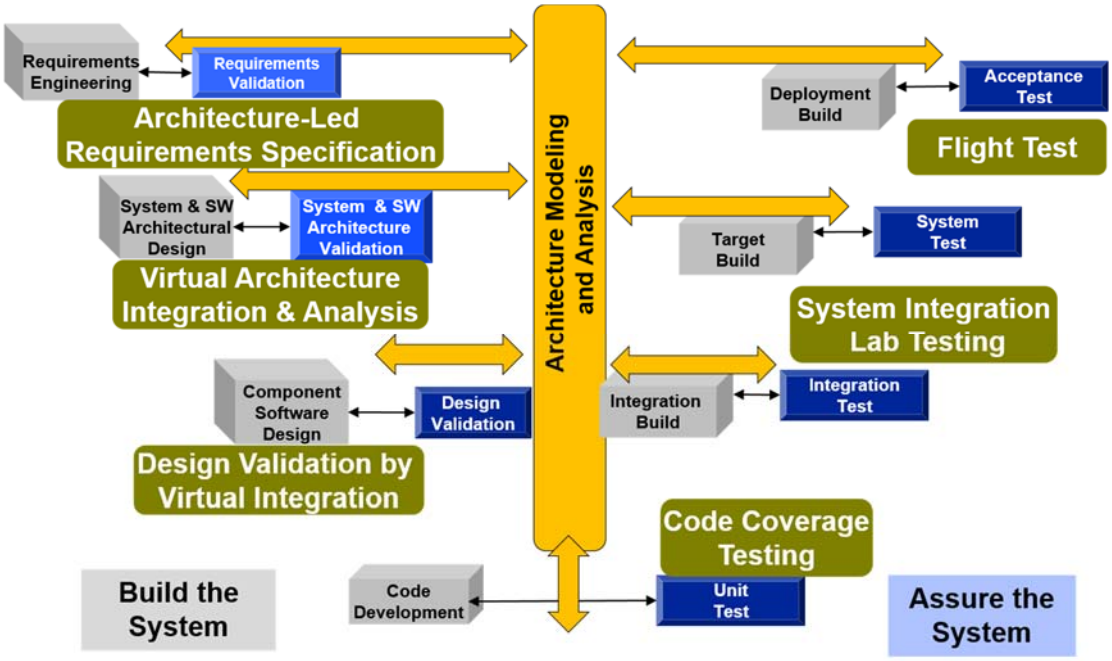

Figure 1: Virtual Integration and Incremental Verification and Validation

One standardized extension, the AADL Error Model Annex standard [SAE 2006], supports architecture fault modeling by enabling annotation of an architecture model with fault occurrence and resulting failure and fault propagation behavior to address dependability concerns in safety-critical systems. A development team can leverage potential fault propagation paths already represented in AADL models to generate representations for safety analysis (see Figure 2). The results allow engineers to use an automated safety analysis process by supporting qualitative and quantitative analysis of system reliability, availability, safety, security, and survivability. The process can also include determining compliance of the system to the specified fault-tolerance strategies. The effectiveness of this approach has been demonstrated on safety assessments of satellite systems [Hecht 2011]. EMV2 has recently been used to demonstrate the automation of the different safety analysis activities that are part of the SAE ARP4761 standard for conducting safety assessments in civil airborne systems and equipment. This case study is the subject of a separate report [Delange 2014].

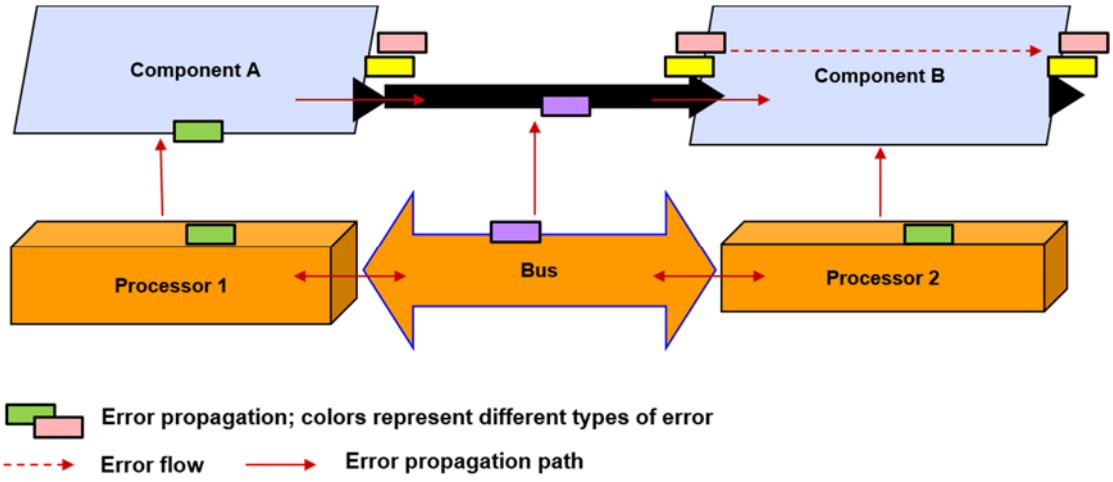

Figure 2: Architecture Fault Model

We have revised the Error Model Annex standard based on users' experience with the original annex in industrial pilot projects to improve the set of concepts and intended semantics for modeling the fault occurrence, propagation, and management behavior of a system. EMV2 is the subject of this report [SAE 2015].

1.3 Language Concepts in EMV2

We use the word *error* in the names of several EMV2 concepts to distinguish them from similar concepts in the AADL core language and other annexes. For example, within EMV2 an *error event* represents the occurrence of an anomalous condition, while *event* in the AADL core language represents a Boolean signal communicated through event ports. In Section 1.4, we relate the EMV2 concepts to terms and concepts commonly used by the safety, reliability, and dependability communities.

EMV2 supports architecture fault modeling at three levels of abstraction, each of which uses specific language concepts defined later in this section:

- focus on fault propagation across the system: propagation of faults and their impact between the system and its operational environment and between subsystems within a system. This level allows for safety analysis in the form of hazard identification and fault impact analysis.
- focus on fault and recovery behavior of components: identification of faults and their occurrences, their manifestation in the component as failure modes, the effect of incoming propagations on failure modes, the propagation of failure modes and incoming propagations as outgoing propagation, and the ability of the component to detect and recover or repair. This level allows for probabilistic reliability and availability analysis.
- focus on compositional abstraction of fault models: relate the fault model of system components to the abstracted fault model of the system. This level allows for scalable compositional fault analysis.

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

1.3.1 Fault Propagation Across the System

The first level of abstraction focuses on fault propagation between subsystems and within the environment. The specification of fault propagation in EMV2 corresponds to the Fault Propagation and Transformation Calculus (FPTC) [Paige 2009]. The following concepts are used to annotate system components:

- *Error propagation* and *containment* are associated with interaction points (ports, data and bus access, remote service calls, deployment binding points) to other components and specify the different types of effect, such as bad value or no service, that a component failure or incoming propagation can have on other components. They can also specify that a component is expected not to propagate certain types of effects. Outgoing and incoming propagation and containment specifications act as contracts between interacting components; they are guarantees and assumptions that must be verified.
- *Error types* characterize the different types of errors being propagated (e.g., a value error or timing error) or different types of error events (e.g., a component being overheated, cracked, or stuck).
- *Error sources* identify components as sources of error propagation; that is, they specify when a component's internal failure results in a propagation.
- *Error paths* and *sinks* specify how components respond to incoming propagations. They describe whether a particular error propagation is passed on to other components in the same form, propagated to other components as a different error type, or contained by the component.
- *Propagation paths* are determined by the logical and physical connectivity in the architecture, the deployment of software on hardware, and user-defined propagation paths not recorded in the AADL core model.
- *Probability properties* are associated with the occurrence of error propagations, sources, paths, and sinks.

Figure 3 illustrates error types associated with outgoing and incoming ports to indicate error propagations, shown as rectangles of different colors. The propagation path between components follows the port connection between Components A and B. Component A is a source of a specific type of error caused by a specific type of failure in Component A (shown as a colored oval). Component A also passes on incoming errors from its *in* port to its *out* port.

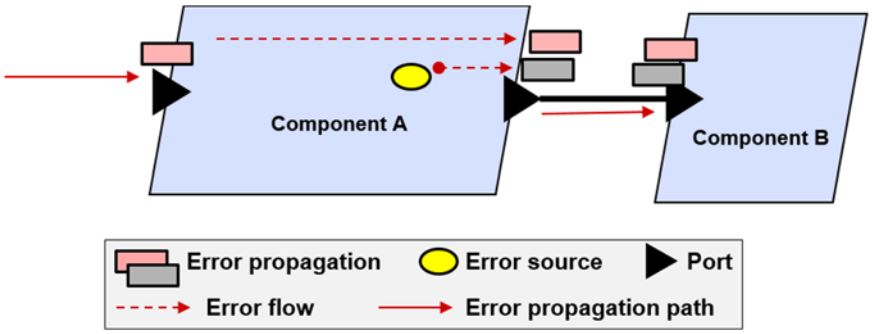

Figure 3: Error Propagation Between Components

1.3.2 Fault and Recovery Behavior of Components

The component fault and recovery behavior specification of EMV2 characterizes the possible fault occurrences and resulting failure modes of the component as a whole. A specification for component error behavior describes the occurrence and handling of error events and incoming propagation in different failure modes, handling of redundant input to tolerate failures in external components, and restrictions that failure modes place on operational modes. The following concepts are used to annotate system components:

- three types of *error behavior events:*
  - *error event:* represents the occurrence (activation) of a fault within a component
  - *recover event:* reflects actions taken by the system to recover from a failure
  - *repair event:* reflects repair or replacement of the failed component by an external agent
- probability properties associated with the occurrence of these events
- *error behavior state* in an *error behavior state machine:* represents the fact that a component is operational or is in a failure state (failure mode)
- *error behavior transition* with trigger conditions: specifies how error, recover, and repair events of the component as well as incoming error propagations from external components change the error behavior state of the component
- *outgoing propagation condition:* specifies that certain error behavior states, incoming error propagations, or combinations result in a particular outgoing error propagation
- *detection condition:* specifies whether a particular component is expected to detect and handle error behavior states and propagations, represented by an event in the AADL core model
- *mode mapping:* specifies how an error behavior state (failure mode) can restrict a set of operational modes in the AADL core model

Figure 4 shows a component with an error event representing failures within the component and two error behavior states: *Operational* and *Failed*. The event causes a transition from *Operational* to *Failed*, and the *Failed* state is observable by others in the form of an error propagation.

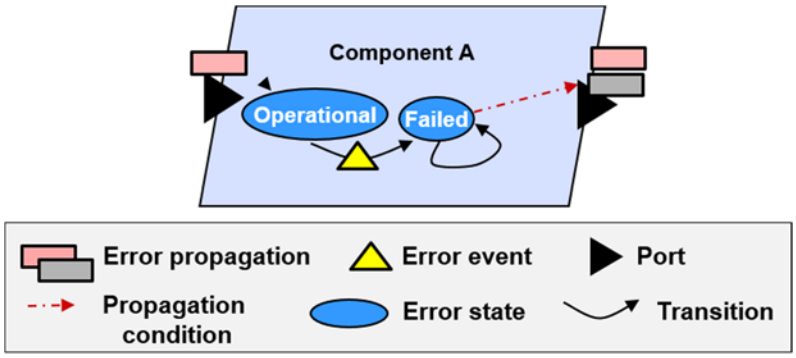

Figure 4: Component Error Behavior

1.3.3 Compositional Abstraction of Fault Models

The compositional abstraction of fault models in EMV2 provides a *composite error behavior state specification* of a system in terms of the error behavior states of its subsystems. This specification defines the conditions under which a component is in a particular error behavior state expressed in terms of the error behavior states of its parts. This specification reflects redundancy in the parts to achieve fault tolerance and corresponds to the logic in fault trees. It must be consistent with the abstract error behavior specifications of the interacting parts.

Figure 5 shows a composite flight guidance system (FGS) with its implementation by several subcomponents, each with its own specification for component error behavior. The figure also shows the FGS with component error behavior that represents an abstraction of its implementation. A composite state declaration specifies that FGS is in the *Failed* state if a component of either flight guidance–autopilot pair fails or if the actuator fails.

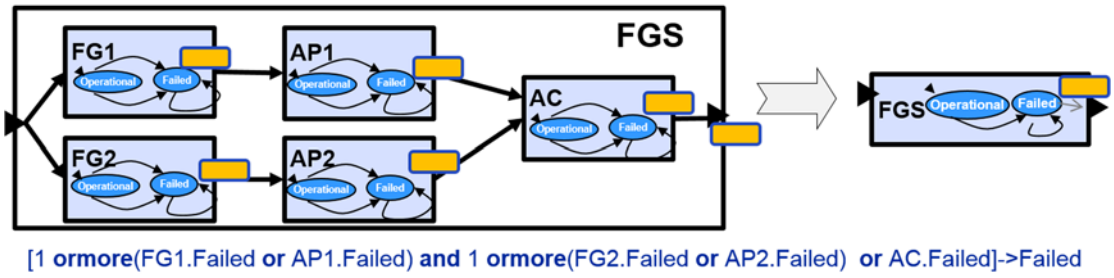

Figure 5: Composite Error Behavior and Its Abstraction
Note: FG = flight guidance component, AP = autopilot component, AC = actuator.

In this report, we discuss an approach to architecture fault modeling that supports safety analysis practices in an architecture-centric manner and does so for both the system architecture and the (embedded) software system architecture. Architecture-centric design and development has value because requirements, including safety requirements, can be specified in an architecture model that describes both the system in its operational context and its composition in terms of subsystems. This approach allows an architecture specification or design to be validated at each level of architectural abstraction for completeness, consistency, and correctness and to be verified against its requirements specification or higher level design specification. The refinement of an architecture specification at each level propagates derived requirements down the architecture hierarchy.

1.4 Terms and Concepts

In EMV2 we always use the keyword **error** to characterize concepts, for example, error event, error behavior state, and error propagation. In this section, we relate EMV2 language concepts to terms defined in *Systems and Software Engineering—Vocabulary*, a common vocabulary for all systems and software engineering work established by the International Organization for Standardization, the International Electrotechnical Commission, and the Institute of Electrical and Electronics Engineers [ISO 2010]. This will help the reader understand how to appropriately use the EMV2 language constructs.

Error is defined as

> ***1.*** *a human action that produces an incorrect result, such as software containing a fault.* ***2.*** *an incorrect step, process, or data definition.* ***3.*** *an incorrect result.* ***4.*** *the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition*
>
> *cf. failure, defect*
>
> *EXAMPLE omission or misinterpretation of user requirements in a software specification, incorrect translation, or omission of a requirement in the design specification. [ISO 2010]*

The definition of error encompasses mistakes by humans (the effect of a failure by the human), defects in a process that can lead to defects in a design or operational system, the effect of incorrect system behavior, and a characterization of anomalous behavior as an indication of a failure.

In EMV2 we consistently use the term *error* as a keyword to avoid confusion with similar constructs in the AADL core language or other annexes, such as event vs. error behavior event or state vs. error behavior state. See also defects, failures, and effects.

Defect is defined as "a generic term that can refer to either a fault (cause) or a failure (effect)" [ISO 2010]. See also fault and failure.

Fault is defined as

> ***1.*** *a manifestation of an error in software.* ***2.*** *an incorrect step, process, or data definition in a computer program.* ***3.*** *a defect in a hardware device or component. Syn: bug*
>
> *NOTE: A fault, if encountered, may cause a failure. [ISO 2010]*

The definition of fault includes one of the definitions for error. It also is defined in terms of defect, which in turn includes a definition in terms of fault. It also is the effect of human or process errors.

In EMV2 we represent fault types as error types. An EMV2 property lets us distinguish between design faults and operational faults. In a fault propagation specification, the presence of a fault in a component is expressed as an error source with the appropriate error type. In a component fault and recovery behavior specification, it is expressed as an error event with an error type, where an instance of that error event represents the activation of the fault (see failure).

Failure is defined as

> ***1.*** *termination of the ability of a product to perform a required function or its inability to perform within previously specified limits;* ***2.*** *an event in which a system or system component does not perform a required function within specified limits. [ISO 2010]*

The definition of failure encompasses the event of a fault activation and the manifestation of the fault activation in the component. The manifestation can be a malfunction, such as loss of service or anomalous behavior. See also failure mode.

In EMV2 we represent failures as error sources in fault propagation specifications and as error events in component fault and recovery behavior specifications. The manifestation of the failure is represented by an error behavior state. A transition specification describes how error behavior events change the error behavior state. See also failure.

Failure mode is defined as "the physical or functional manifestation of a failure" [ISO 2010]. According to the definition, a failure mode is associated with a physical or logical component.

In EMV2 we represent failure modes as error behavior states. Error behavior states are specified as part of error behavior state machines. Error behavior states can have error types to characterize different types of failure modes. Different types of faults are mapped to the respective type of failure mode through a transition specification. See also failure.

Failure mode and effect analysis (FMEA) is defined as

> *[Technique] an analytical procedure in which each potential failure mode in every component of a product is analyzed to determine its effect on the reliability of that component and, by itself or in combination with other possible failure modes, on the reliability of the product or system and on the required function of the component; or the examination of a product (at the system and/or lower levels) for all ways that a failure may occur. For each potential failure, an estimate is made of its effect on the total system and of its impact. In addition, a review is undertaken of the action planned to minimize the probability of failure and to minimize its effects. [ISO 2010]*

Although the *Vocabulary* refers to "effects," it does not define the term [ISO 2010]. In the context of the FMEA definition, an effect refers to the failure mode. The effect of a failure is its propagation to other components and their responses to this propagation. Failure propagation is also referred to as fault propagation. The outgoing propagation can be due to a component failure or due to an incoming propagation. From the receiver's perspective, there is no difference, but from a diagnostic perspective, it is useful to identify the source.

In EMV2 we represent the propagation of failure modes and incoming effects as outgoing error propagations. Components can be the source of an outgoing propagation, they can be the sink of an incoming propagation, or they can pass on incoming propagations as outgoing propagations of the same or a different type. Conditions for outgoing propagations provide traceability to the cause, such as a failure (an error behavior state) triggered by an error event. An incoming propagation may affect the failure mode, or it may be passed on while the component is not in a failure mode. The probability of occurrence is associated with error behavior events, error sources, error behavior states, and error propagations to support stochastic analysis. The logic conditions also reflect fault tolerance strategies such as redundancy.

Hazard is defined as

> ***1.*** *an intrinsic property or condition that has the potential to cause harm or damage.* ***2.*** *a source of potential harm or a situation with a potential for harm in terms of human injury, damage to health, property, or the environment, or some combination of these. [ISO 2010]*

The definition of hazard has its root in safety engineering. It refers to both the effect and the source of a failure. Failures that result in loss of service are hazards that affect reliability. From a safety perspective, even minor failures must be considered hazards because combinations of them can have catastrophic effects [Leveson 2012]. The definition of hazard focuses on effects of failures in terms of injury and damage; that is, failures represent safety hazards. We include security hazards under the concept of hazard.

In EMV2 we represent hazards by a multivalued property that can be associated with the error source, error behavior state, and error propagation to support both definitions of hazard.

Security is defined as "the protection of system items from accidental or malicious access, use, modification, destruction, or disclosure" [ISO 2010]. The definition of security includes accidental malicious indication of anomalous behavior either from outside a system or by unauthorized crossing of a system's internal boundaries—typically in a manner that takes advantage of faults. The term *system item* covers information as well as physical components.

Threat is defined as

> ***1.*** *a state of the system or system environment which can lead to adverse effect in one or more given risk dimensions.* ***2.*** *a condition or situation unfavorable to the project, a negative set of circumstances, a negative set of events, a risk that will have a negative impact on a project objective if it occurs, or a possibility for negative changes. [ISO 2010]*

The definition of hazard focuses on the effects of failures in terms of injury and damage; that is, failures represent safety hazards. In EMV2, we include security hazards under the concept of hazard, and threats are one class of security hazards.

Fault tolerance is defined as

> ***1.*** *the ability of a system or component to continue normal operation despite the presence of hardware or software faults.* ***2.*** *the number of faults a system or component can withstand before normal operation is impaired.* ***3.*** *pertaining to the study of errors, faults, and failures, and of methods for enabling systems to continue normal operation in the presence of faults.*

The definition of fault tolerance focuses on the presence of hardware and software faults. Fault tolerance includes fault detection, fault isolation, and fault recovery. Fault recovery elements include fault containment, fault masking, fault repair, and fault correction. Fault avoidance focuses on not introducing faults or eliminating them before the system goes into operation.

Error tolerance is defined as "the ability of a system or component to continue normal operation despite the presence of erroneous inputs" [ISO 2010]. The definition of error tolerance focuses on propagated errors.

In EMV2 tolerance of both faults and errors is supported by transition and outgoing propagation conditions that refer to error behavior states and incoming propagations. Expectations on fault tolerance through redundancy is expressed through appropriate logic in the transition and outgoing propagation conditions. Expected detection of faults by the system is specified by error detection declarations as well as recover and repair events. These specifications provide traceability into the fault tolerance architecture of the system.

For mechanical components, the error behavior specified by EMV2 can reflect both fault occurrences in the physical components due to operational use and latent design faults that are triggered during operation. For physical systems, long design cycles are assumed to reduce design errors to a minimum over time. For software, however, all fault occurrences result from design and coding errors, and we cannot make zero-defect assumptions for software. EMV2 allows users to characterize error events as representing design errors and operational errors to help them distinguish and respond to each type appropriately.

1.5 Reader's Guide

Section 2 introduces the concepts of error model libraries and error model subclauses. An error model library contains reusable elements, such as error types and type sets, error behavior state machines, type mapping sets, and type transformation sets. An error model subclause contains component-specific error model annotations, such as error propagations and error flows, and component-specific error behavior in terms of error behavior events, error behavior states and transitions, and composite error behavior. The error model subclause models the error behavior of a component in terms of the error behavior of its subcomponents.

Section 3 introduces the concepts of error types and type sets, their organization into error libraries, their extension and adaptation to specific application domains through aliases, and the concepts of type products and type hierarchies to represent interaction between error types. This section also introduces the reader to an ontology of error propagation types that is a good starting point for fault modeling. The section concludes with a discussion of type mappings and the ability to specify type equivalence of independently developed error type libraries.

Section 4 focuses on fault propagation across the system and its impact. It introduces the concepts of error propagation points and outgoing and incoming error propagation specifications. It also introduces the concepts of error sources, error sinks, and error paths, which allow users to abstractly specify how a system component deals with errors. The error propagation path determines what components are affected by an outgoing propagation from a component. Finally, the concept of error containment, combined with error propagation, specifies the fault model assumptions and contracts that a component makes about its interactions with other components.

Section 5 covers specification of component fault behavior. It introduces the concepts of error events, states, and transitions. Reusable error behavior state machines ease the job of defining such behavior. Recover and repair events, as well as outgoing propagation conditions and error detection conditions, support the specification of fault tolerance strategies.

Section 6 focuses on compositional abstraction of error behavior. It explains how to specify the abstracted behavior of a system in terms of the error behavior states of its components.

Section 7 discusses the use of properties with error model elements as well as their definition by users. It also provides a summary of properties that have been predeclared as part of the EMV2 standard.

Section 8 presents advanced topics in EMV2. They include the concept of inheritance of error model specifications between component types and implementations as well as through the component extension hierarchy. This section also covers error model specifications for feature groups, user-defined propagation points and paths, error type mappings and type equivalence, type transformations and their use in specification of connection error behavior, and the mapping between operational modes and failure modes, which are also known as error behavior states.

Section 9 discusses several example models. They include a Global Positioning System (GPS) system with a focus on the interaction between operational and failure modes, a dual-redundant flight guidance system to show the consistency between an abstracted architecture fault model and its composite model for a fault-tolerant system, an abstract specification of a fault behavior interface for a network protocol stack, an abstract specification of a fault model for a dual-channel avionics network to ensure correct and consistent use, and a triple-redundant system that has both physical and logical redundancy.

Section 10 provides the EMV2 syntax rules.

2 Error Model Libraries and Subclause Annotations

EMV2 supports architecture fault modeling and analysis through the annotation of architecture models expressed in AADL with

- reusable error model libraries
- component-specific error behavior annotations through error model subclauses

This section explains how to use libraries and subclauses.

2.1 Error Model Library

2.1.1 Role of an Error Model Library

The error model library provides reusable collections of error type and type set definitions, as well as mapping and transformation rules between error types. For example, users can map an incoming error propagation type, such as an out-of-range value, into an outgoing error propagation type, such as a missing value. In addition, an error model library can contain reusable specifications for error behavior state machines that consist of error events, states, and transitions. EMV2 includes a library of predefined error types.

2.1.2 Using the Error Model Library

An error model library is defined in an AADL package. An AADL package can contain only one error model library—a restriction of the AADL core language. However, the package can contain other declarations. Within the package, use an AADL annex library clause, which identifies the annex as *EMV2*. Refer to the library using the package name.

An error model library consists of zero or one error type library (**error types**) declaration, zero or more error behavior state-machine (**error behavior**) declarations, zero or more **type mappings** declarations, and zero or more **type transformations** declarations, in that order. The full syntax rule for error model libraries can be found in Section 10.1.

An example error model library specification is illustrated in Figure 6. It shows the definition of several error types, a type set, and an error behavior state machine with one error event (Fail), two states (Operational and Failed), and one transition (from Operational to Failed when the Fail error event occurs).

```
package MyErrorLib
public
annex EMV2 {**
error types
     PowerFailure: type;
     TransientPowerLoss: type;
     PowerErrors: type set { PowerFailure, TransientPowerLoss };
     SensorFailure: type;
     NoService: type;
     BadData: type;
     NoData: type;
     MissingCmd: type;
 end types;
```

```
error behavior TwoState
events
  Fail: error event;
states
  Operational: initial state;
  Failed: state;
transitions
  Failure: Operational -[Fail]-> Failed;
end behavior;
 **};
end MyErrorLib;
```

Figure 6: Error Model Library

2.2 Error Model Subclause

2.2.1 Role of an Error Model Subclause

An error model subclause provides component-specific fault model specifications by annotating component type, component implementation, and feature group type declarations. These specifications support the three levels of abstraction discussed in Section 1.3. They do so by utilizing error type libraries and error behavior state machines that are defined in error model libraries.

Supporting the error propagation specification, the error model subclause lets us specify incoming and outgoing error propagations and containments, as well as error sources, paths, and sinks.

Supporting the error behavior specification for components, the error model subclause lets us identify the error behavior state machine applicable to the component. It supports the introduction of component-specific error events, error behavior transition conditions in terms of error events, incoming propagations to the component, conditions under which propagations to other components occur, and whether the component will detect a failure represented by an error behavior state or incoming propagation. It also lets us specify how error behavior states affect the operational modes of the component.

Supporting the behavior specification for composite components, the error model subclause lets us specify how each abstract error behavior state of a component relates to the error behavior states of its parts.

Finally, the error model subclause allows us to associate property values with any element defined within the error model subclause. This can be done using properties defined in the EMV2 Annex standard or properties defined by users.

2.2.2 Using the Error Model Subclause

A component-specific error behavior specification is defined by an AADL annex subclause identifying the annex as *EMV2*. The subclause is declared in a component type, component implementation, or feature group type. An error model subclause consists of

- a **use types** declaration (see Section 3.2)
- a **use type equivalence** declaration (see Section 8.4)
- a **use mappings** declaration (see Section 8.4)
- a **use behavior** declaration (see Section 5.2)

- an **error propagations** section (see Section 3)
- a **component error behavior** section (see Section 5)
- a **composite error behavior** section (see Section 6)
- a **connection error** section (see Section 8.4)
- a user-defined **propagation paths** section (see Section 8.2)
- a **properties** section (see Section 7)

The subclause elements must be declared in the above order, but all of these elements are optional.

The **use types** declaration lists the error type libraries whose content becomes accessible to the subclause without having to qualify references with the error type library name.

The **use mappings** clause specifies the type mappings to be used in error paths when no target type is specified.

The **use behavior** declaration identifies the error behavior state machine to be used for the component. The subclause has separate sections for error propagations, component error behavior, and composite error behavior. The identified error behavior state machine applies to all of these sections.

An example subclause is shown in Figure 7. The full syntax rule for the error model subclause is shown in Section 10.5.

```
package Devices
public
device PowerSupply
features
 PowerOutlet: provides bus access;
annex EMV2 {**
 use types MyErrorLib;
 use behavior MyErrorLib::TwoState;
error propagations
  PowerOutlet: out propagation {PowerFailure};
end propagations;
**};
end PowerSupply;
end Devices;
```

Figure 7: Example of an Error Model Subclause

3 Error Types and Common Type Ontology

EMV2 provides the concept of *error type* to characterize the types of errors to be propagated, the type of activated fault represented by an error event, and the type of failure mode represented by an error behavior state.

Error types can be grouped into *type sets*. A reference to the type set represents that collection of types. Type sets can act as constraints regarding types. For example, a type set associated with an incoming error propagation declaration indicates the set of acceptable incoming error types.

Error types and type sets are defined in reusable *error type libraries* that can be imported into an error model subclause. Users can import several type sets into the same error model subclause and use mapping rules to explain the relationship between them.

Error types can be combined into *type products* to indicate that a failure or propagation is characterized by both types at the same time. For example, a sent message may be late and contain an out-of-range value.

Finally, error types can also be placed into a *type hierarchy*. Error types from the same hierarchy cannot be combined in a type product as they are not expected to occur in combination. For example, a message cannot be late and early at the same time.

We proceed by describing the use of these concepts and introducing a common set of reusable error types as an error propagation ontology.

3.1 Error Types and Type Sets

3.1.1 Role of Error Types and Type Sets

An error type is a categorical label that is used to characterize the type of error in error propagation, error containment, error flow, error event, and error behavior state declarations. This label is also used to characterize conditions for state transition, outgoing error propagation, and detection declarations. Users do this by specifying a set of error types as part of their declarations.

For an error propagation or containment declaration, the type set indicates the collection of error types that potentially are or are not propagated through an error propagation point, such as a port. An instance of an error propagation is represented by a *type instance*, which is an instance of one of the elements listed in the type set. An error propagation instance of an element in a type set can be thought of as a type token that is propagated through the system.

For error events, an error type set specifies a collection of possible fault occurrences. For example, instead of defining 15 different error events, one for each kind of error that can occur in a component, we can specify a single error event and indicate the various error types as a type set annotation to the error event declaration (see Section 5.1.4). An occurrence of an error event is represented by an instance of one of the type set elements.

Similarly, instead of introducing 15 error behavior states to match the 15 types of error events, we can specify a single *Failure* or *Malfunction* state that is annotated with a type set of error types

that characterize the different kinds of failures or malfunctions that can occur for a component. In other words, the state *Failure* can be viewed as having substates, as many as one for each element in the type set represented by *Failure*. Each substate can hold an instance of one or more type set elements that represent the substate.

3.1.2 Using Error Types and Type Sets

Error types and type sets are defined in error type libraries (see Section 3.2). The syntax rules for error type libraries, error types, and type sets are shown in Section 10.2.

An *error type* declaration defines a new error type. The error type declaration consists of the defining name for the error type followed by a colon and the keyword **type**. Every error type definition within an error type library must have a unique name.

A type set can be specified in two ways: by specifying the elements of a type set explicitly in different contexts or by declaring a named type set and then referring to the type set by name.

1. A *type set specification* is a comma-separated list of one or more error type and type set names enclosed by curly brackets. The elements of a type set are expected to be unique error types.
2. A *named type set* declaration defines the name, followed by a colon and the keywords **type set**, and a type set specification listing the elements of the set. Users then refer to the named type set in different error propagation declarations instead of specifying multiple error types repeatedly.

An element of a type set specification can refer to a named type set, which then includes all the elements of the referenced type set in this type set. This effectively allows users to add elements to an existing type set or to specify unions of type sets.

Figure 8 shows the declaration of three error types related to valves and two type sets. The type set *ValveError* represents a collection of error types related to a valve. Error types can be added to a type set, resulting in a new named type set, as illustrated by *MoreValveError* in Figure 8. Multiple error type sets can be combined in a defining error type set declaration or in a type set specification. The resulting type set is a union of the elements of the referenced type sets as well as additional error types listed as elements. This allows users to define aggregations of type sets. For example, in Figure 8 *SystemErrors* is an aggregation of the type sets *HydraulicErrors* and *ElectricalErrors*.

```
package MyValveErrorLib
public
annex EMV2 {**
error types
error types
     StuckOpen: type;
     StuckClosed: type;
     ValveLeak: type;
     ValveError: type set {StuckOpen, StuckClosed};
     MoreValveError: type set {ValveError, ValveLeak};
end types;
**};
end MyValveErrorLib;
```

Figure 8: User-Defined Error Types

Type set specifications are used to annotate error propagations, error flows, error behavior states, and error events with acceptable sets or error types. Figure 9 illustrates the use of a type set specification in error propagation declarations. This error model subclause imports the types from *MyValveErrorLib* with the **use types** keywords. The out propagation *ValveFlow* specifies a type set composed of the error types *StuckOpen* and *StuckClosed*. The out propagation *ValveFlow2* shows a type set specification that references the named type set *ValveError*. The out propagation *ValveFlow3* shows a type set specification that is the union of the named type set *ValveError* and the error type *ValveLeak*.

```
annex EMV2 {**
use types MyValveErrorLib;
error propagations
   ValveFlow: out propagation {StuckOpen, StuckClosed};
   ValveFlow2: out propagation {ValveError};
   ValveFlow3: out propagation {ValveError, ValveLeak};
end propagations;
**};
```

Figure 9: Use of Type Sets in Error Propagations

A type constraint is used in transition, outgoing error propagation condition, and detection condition declarations to indicate the condition that the referenced error propagation point must satisfy. A type constraint consists of a type set specification or the specification `{NoError}`, where **NoError** is a keyword. If the type constraint is a type set specification, the error propagation point must have an error propagation instance of one of the types listed in the type set specification. If the type constraint specifies `{NoError}`, no error propagation instance must be present at the error propagation point. This allows us to specify conditions for a transition if one incoming error propagation point has a propagated type instance and another error propagation point is error free, for example, `InFlow1{ValveLeak} and InFlow2{NoError}`.

A *type instance* declaration is used to specify a resulting error type: the target error type of an error path, the resulting error type of the target state in a transition declaration, or the resulting propagated error type in an outgoing propagation declaration. Users do this by enclosing the error type in curly brackets.

3.1.3 Observations

Users can introduce different sets of error types for different types of components. This allows them to use error type names that are meaningful in the context of the component. Users can then place these sets of error types in separate error type libraries—for example, if the set will be found in different packages—or they can place a different type set in the same error type library.

Users may introduce one set of error types to be used with error events to reflect a particular activated fault, such as *Overheated*; a separate set of error types to be used with error behavior states to reflect the resulting failure mode, such as *DegradedOperation*; and a third set of error types to be used with error propagations to represent the resulting effect on other components, such as *SlowService*.

3.2 Reusable Error Type Libraries and Aliases

Error types and error type sets can be organized into error type libraries for use in error model subclauses. Names of error types and type sets can be adapted with context-specific names by using alias declarations. In this section, we describe the definition (**error types**) and use (**use types**) of error type libraries and illustrate the use of aliases (**renames**).

3.2.1 Role of Error Type Libraries and Aliases

Error type libraries provide reusable sets of error types and type set definitions. Error model libraries can be defined as extensions of existing error type libraries. They allow users to add error types to an existing error type library and make the extended collection of error types and type sets available under a new error type library name.

The error types and type sets of multiple error type libraries can be made accessible to an error model subclause using the **use types** keywords. They allow users to refer to error types and named type sets without qualifying them with the error type library name.

Users can define aliases for existing error type and type set definitions using **renames**. These aliases are equivalent to the original error type or type set. An alias may provide a more meaningful name in a particular context while at the same time reflecting the fact that it represents the same type as the original. For example, a user may define an error type *NoService*, which in the context of a power supply may be referred to by the alias *NoPower*.

3.2.2 Using Error Type Libraries and Aliases

An error type library is declared within the **error types** section of an error model library, which is terminated by the keywords **end types**. It consists of a list of error type, type set, error type alias, and type set alias declarations in any order, followed by an optional **properties** section as shown in Figure 10. All error type definitions, type set definitions, and alias definitions must have unique names within the error type library. The syntax rules for error type libraries are shown in Section 10.2.

```
package MyExtendedValveErrorLib
public
annex EMV2 {**
error types extends MyValveErrorLib with
     SlowOpen: type;
     SlowClose: type;
     DrippingValve renames type ValveLeak;
     AllValveError: type set {MoreValveError, SlowOpen, SlowClose};
properties
     EMV2::Description => “Lubrication issue” applies to SlowOpen, SlowClose;
end types;
**};
end MyExtendedValveErrorLib;
```

Figure 10: Error Type Library

The **properties** section of the error type library allows us to associate property values with elements of the error type library. Figure 10 shows an example of a *Description* property applied to the error types *SlowOpen* and *SlowClose*. For more on property associations in EMV2, see Section 7.

An error model library can contain only one error type library. An error type library is referred to by the name used for the error model library, that is, the name of the package containing the error model library (see Section 2.1.2).

An error type library can be declared as an extension of another error type library by specifying the library to be extended using the **extends** clause (see Figure 10). This extension will include all error type, type sets, and aliases of the original error type library in the name space of the newly defined error type library. Both the locally declared error types, type sets, and aliases as well as those of the library being extended are considered part of the error type library. The local definitions must not conflict with those of the library being extended.

Users can introduce an *alias* for an existing error type or type set that may be more meaningful in different contexts using the keywords **renames type** or **renames type set**. The example in Figure 10 shows the declaration of *DrippingValve* as an alias for *ValveLeak*.

An error type library can also be declared to use error types from other libraries with the **use types** clause. In this case, the error types, type sets, and aliases can be referenced in local error type, type set, and alias declarations, but they are not included in the name space. For example, users can define aliases for an error type of the predeclared error type library, but only the aliases are available to subclauses whose **use types** refer to this error type library. Figure 11 shows a set of alias declarations for error types from *MyValveErrorLib* plus a type set for the newly named error types. When referring to *WaterValveErrorLib* in a **use types** clause, only the local names are available.

```
package WaterValveErrorLib public
annex EMV2 {**
error types
    use types MyValveErrorLib;
     DrippingValve renames type ValveLeak;
     RustedShut renames type StuckShut;
     RustedOpen renames type StuckOpen;
     WaterValveError: type set {DrippingValve, RustedShut, RustedOpen};
end types;
**};
end WaterValveErrorLib;
```

Figure 11: Error Type Library of Aliases Only

In an error model subclause, users can use the **use types** declaration to specify that the error type, type set, and alias definitions in a list of error type libraries become accessible. Two different error type libraries may be listed that have an error type, type set, or alias definition with the same name. In this case, the conflicting names must be qualified when they are referenced. Qualify a reference to an error type, type set, or alias by preceding the name with the error type library name followed by a double colon ("**::**").

3.2.3 Observations

Typically, users will define their own error types when characterizing the different kinds of faults that can exist in a component. However, users may reuse the predeclared error type library in EMV2. In this case, users might want to use alias declarations for the original names to associate context-specific labels to these types.

When error types, type sets, and aliases are made accessible within an error model subclause using **use types**, their names do not conflict with the names of items defined in the subclause, such as events and transitions.

3.3 Type Products and Type Hierarchy

The EMV2 type system allows users to define type instances as error *type products* to represent the fact that a single error occurrence is characterized by two or more error types. Users can also place error types into a *type hierarchy* by declaring an error type to be a subtype of another error type. In this section, we describe how to use these two concepts.

3.3.1 Roles of Type Products and Type Hierarchies

The role of a *type product* is to characterize an instance of an error propagation by a combination of error types. For example, an outgoing message on a port can be characterized as an out-of-range value and late. Similarly, we may specify that a transition occurs only under the condition that a message is both out of range and late.

The role of a *type hierarchy* is to indicate that certain error types cannot occur simultaneously as part of the same instance; that is, they cannot be elements of the same type product. For example, a message cannot be both late and early. In a type hierarchy, a super type acts as a type set consisting of its subtypes. In other words, when a type with subtypes is referenced as a type set element, all subtypes are considered to be part of the type set.

3.3.2 Using Type Products and Type Hierarchies

A type product is specified by listing error types separated by an asterisk (*). The elements of a type product must be from different type hierarchies. The syntax rules for error types and type products are shown in Section 10.2.

Type products can be used as elements of type sets and type constraints. Figure 12 illustrates the use of a type product in the type set of an outgoing error propagation. The type set specifies that the message can be late, that its value can be out of range, and that the message can be both late and out of range.

```
Message: out propagation {OutOfRange, LateDelivery, OutOfRange*LateDelivery};
```

Figure 12: Use of Type Product

Users can place error types into a *type hierarchy* when they define the error type by declaring it as an extension of another error type using the keyword extends. The textual syntax is shown in Figure 13. Here, *EarlyDelivery* and *LateDelivery* are two subtypes of *TimingError*, indicating that they cannot occur at the same time for a particular type instance.

```
TimingError: type;
EarlyDelivery: type extends TimingError;
LateDelivery: type extends TimingError;
```

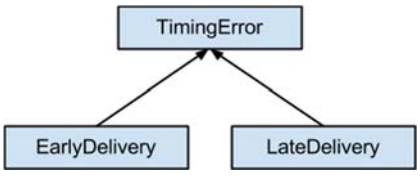

Figure 13: Error Type as a Subtype in a Type Hierarchy

An error type that is a subtype can itself have subtypes. An error type cannot be a subtype of more than one error type. This constraint is enforced syntactically.

When an error type with subtypes is listed as an element of a type set or type constraint, all of its subtypes are considered part of the type set.

When an error type with subtypes is listed as an element of a type product, any of its subtypes effectively are combined with the other elements of the type product. If two or more type product elements are error types with subtypes, the result is effectively all possible combinations of the subtypes.

In defining a type set, users can list two different subtypes from the same type hierarchy as elements, but one cannot be the subtype of the other. For example, if types A, B, and C are subtypes of type Z, and type C has subtypes D and F, users can list subtypes of type Z (e.g., A and F), but they cannot list both of those subtypes and a type that either of them extends (i.e., C cannot be included in the new set since F extends C).

3.3.3 Observations

Users may initially define error types as independent types and place them in a type set. They may later determine which error types cannot be associated with an error event or error propagation at the same time and place them into the same type hierarchy.

Each type hierarchy has a single root type. Multiple type hierarchies can be defined in the same error type library. A type hierarchy defined in one error type library can be extended by defining types as subtypes even when those definitions are declared in another error type library.

3.4 An Ontology of Common Error Propagation Types

In this section, we introduce a set of error types that has been predefined in EMV2. It is available as an error type library named *ErrorLibrary*. It represents an ontology of common error types to characterize error propagations. This ontology draws on previous work [Bondavalli 1990, Powell 1992, Walter 2003].

3.4.1 Role of the Ontology of Error Types

Components can fail in a number of different ways and affect other components. While the number of error types characterizing an error event or an error source may be large and specific to the component, the effects on other components can be characterized by a smaller number of error types. The error type ontology focuses on how error types are propagated. Users can define libraries of error types to characterize the different domain-specific ways that components can fail and associate those with error events.

For example, the effect of a sensor failure is that it sends an incorrect reading (value error), it misses a reading (item omission), or it does not provide any readings (service omission). These three types of effects may be caused by a number of different factors, such as overheating, radiation, low power, or a material defect. These are domain- and component-specific, user-defined error types.

For software components that operate in fault containers (such as a process with a runtime-enforced address space protection or a partition with both space and time enforcement), the error propagation is limited to propagation paths along explicit interaction channels (such as port connections, shared data access, and subprogram service calls) and execution platform bindings. Fault containers allow us to map a large number of software component faults into a limited number of propagation types and propagation paths. For example, a divide by zero in an arithmetic expression or a deadline miss by a periodic thread might manifest itself as an omission of output that the recipient of this output can observe. Within a fault container, a software error such as buffer overflow or incorrect use of pointers can potentially corrupt other code or data.

Error types are defined by viewing components as providers of a service that consists of a sequence of service items. The categories of error types include errors related to service, value, timing, and redundancy and concurrency. Furthermore, within each category the error types may characterize the service as a whole, the sequence of service items, or an individual service item. The Error Model Annex standard [SAE 2015] includes a formal specification of each error type.

3.4.2 Service-Related Errors

Service errors (*ServiceError*) represent errors related to the number of delivered service items. We distinguish between omission errors, which represent service items not delivered, and commission errors, which represent delivery of service items that were not expected.

The error types for individual service items as subtypes of *ServiceError* are

- *ItemOmission:* the omission of a single service item, such as a lost message
- *ItemCommission:* provision of an item when not expected, such as a spurious message

The error types for a sequence of service items are

- *SequenceOmission:* a number of missing service items, such as missed sensor readings
  - *BoundedOmissionInterval:* a minimum number of service items between item omissions, such as missed sensor readings
  - *TransientServiceOmission:* a limited sequence of item omissions, such as a temporary power outage
  - *EarlyServiceTermination:* omission of all service items partway into the service provision, such as a power failure
  - *LateServiceStart:* initial service items not provided, such as difficulty in starting a generator to provide power
- *SequenceCommission:* a limited sequence of item commissions with the following subtypes:
  - *TransientServiceCommission:* a limited sequence of extra service items, such as extra alarm messages
  - *LateServiceTermination:* additional service items after the expected termination of service, such as warning messages about an overheated engine after the engine stops
  - *EarlyServiceStart:* extra service items provided before the expected service start, such as engine sensor readings before engine start

The error types for the service as a whole are

- *ServiceOmission:* failure to provide a service when expected, such as no power due to a blown transformer
- *ServiceCommission:* provision of service when not expected, such as inadvertent charge on an inactive power line

These errors have been placed into a type hierarchy, as shown in Figure 14.

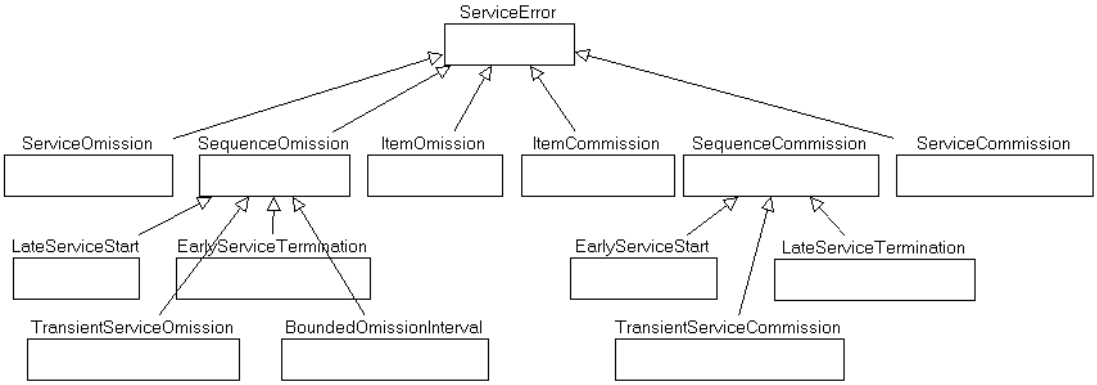

Figure 14: Hierarchy of Service-Related Error Types

3.4.3 Value-Related Errors

Value-related errors deal with the value domain of a service. We distinguish between value errors of individual service items (*ItemValueError*), value errors that relate to a sequence of service items (*SequenceValueError*), and value errors related to the service as a whole (*ServiceValueError*). They form the type set *ValueRelatedError*.

Each of the three types is the root of a separate type hierarchy. The hierarchy allows us to use error types in combination. For example, we can specify that a *BoundedValueChange* error is *OutOfRange*.

ItemValueError consists of

- *DetectableValueError:* a value error that is detectable from the value itself. An example of a detectable value error is an out-of-range value.
- *UndetectableValueError:* a value error that cannot be recognized based on available information. An example of an undetectable value error is a rounding error of a value within range.

DetectableValueError has the following subtypes:

- *OutOfRange:* a value that is outside a specified range, with two subtypes *BelowRange* and *AboveRange*
- *OutOfBounds:* a value error of a multidimensional state variable. For example, in a control system, the control state variable may be within range in each dimension but outside the controllable space.

SequenceValueError consists of

- *BoundedValueChange:* a difference between two consecutive values greater than a specified limit. For example, in a control system, set-point values may be expected to change by less than a specified value.
- *StuckValue:* a value that remains the same for a number of consecutive service items
- *OutOfOrder:* values in the sequence that are not in the correct order

ServiceValueError consists of

- *OutOfCalibration:* a value error in which all values are off by some value. For example, in a control system, an incorrect calibration value may cause all controller output values to be incorrect.

The type hierarchies for value-related errors are shown graphically in Figure 15. The top-level error types are grouped into the type set *ValueRelatedError* (not shown). Note that both sequence and service value errors imply item value errors.

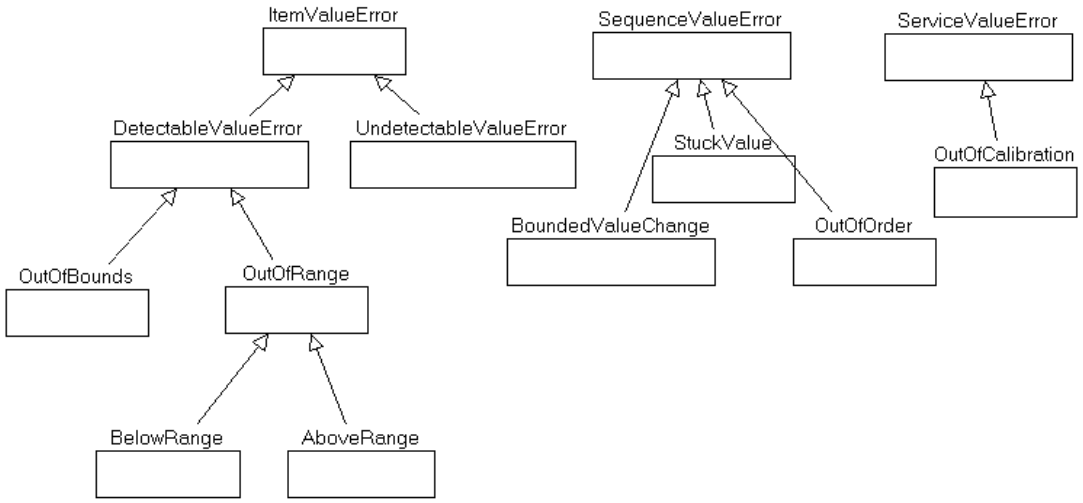

Figure 15: Hierarchy of Value-Related Error Types

Figure 16 shows a predeclared set of aliases for value errors.

```
-- Common aliases for value related errors
ValueError renames type ItemValueError; -- legacy
SequenceError renames type SequenceValueError; -- legacy

IncorrectValue renames type ItemValueError;
ValueCorruption renames type ItemValueError;
BadValue renames type ItemValueError;

SubtleValueError renames type UndetectableValueError;
BenignValueError renames type DetectableValueError;
BenignValueCorruption renames type DetectableValueError;
SubtleValueCorruption renames type UndetectableValueError;
```

Figure 16: Aliases for Value-Related Error Types

3.4.4 Timing-Related Errors

Timing-related errors deal with the time domain of a service. We distinguish between timing errors of individual service items (*ItemTimingError*), timing errors related to a sequence of service items (*SequenceTimingError* or its alias, *RateError*), and timing errors related to the service as a whole (*ServiceTimingError*). They form the type set *TimingRelatedError*.

Each error type is the root of a separate type hierarchy, allowing us to combine them because they occur independently. For example, we can specify that a service that started late may execute at the wrong rate. Item-timing errors characterize the departure or arrival time of individual items. Sequence-timing errors focus on the timing interval, or the rate, between items. Service-timing errors reflect the fact that the service as a whole may be time-shifted, but the rate and times of individual items are acceptable.

ItemTimingError consists of

- *EarlyDelivery:* delivery of a service item before an expected time range, such as a sensor reading arriving before the previous reading has been sampled for processing
- *LateDelivery:* delivery of a service item after an expected time range, such as a sensor reading arriving after the beginning of the next frame

SequenceTimingError, with the alias *RateError*, consists of

- *HighRate:* The inter-arrival time of all service items is less than the expected inter-arrival time. For example, a sender sends periodic messages every 25 ms, while the receiver processes the messages as they arrive and takes an average of 26 ms to complete processing.
- *LowRate:* The inter-arrival time of all service items is greater than the expected inter-arrival time.
- *RateJitter:* Service items are delivered at a rate that varies from the expected rate by more than an acceptable tolerance.

ServiceTimingError, with the alias *ServiceTimeShift*, represents errors where a service delivers all service items time shifted by a time constant. It consists of the two subtypes *DelayedService* and *EarlyService*.

The type hierarchies for timing-related errors are shown graphically in Figure 17. The top-level error types are grouped into the type set *TimingRelatedError* (not shown).

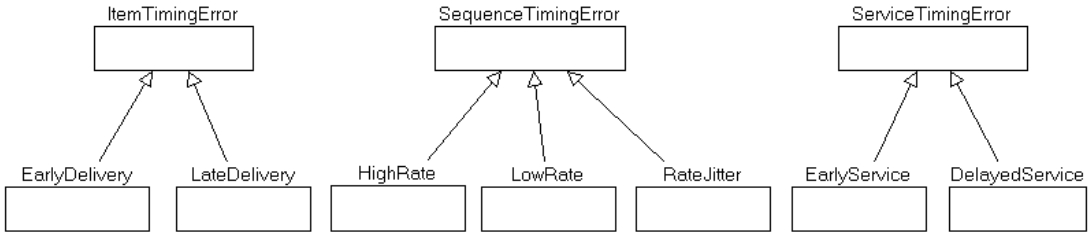

Figure 17: Hierarchy for Timing-Related Error Types

Figure 18 shows a predeclared set of aliases for timing errors.

```
TimingError renames type ItemTimingError; -- legacy
RateError renames type SequenceTimingError; -- legacy
EarlyData renames type HighRate;
LateData renames type LowRate;
ServiceTimeShift renames type ServiceTimingError;
```

Figure 18: Aliases for Timing-Related Error Types

3.4.5 Replication-Related Errors

Replication-related errors (*ReplicationError*) deal with replicates of a service item. Replicate service items may be delivered to one recipient, such as a fault-tolerance voter mechanism, or to multiple recipients, such as separate processing channels. Replicate service items may result from an inconsistent fan-out from a single source, or they may result from an independent error occurring to individual replicates, such as multiple sensors reading the same physical entity or an error in one of the replicate processing channels.

ReplicationError consists of

- *AsymmetricReplicatesError:* At least one of the replicates is different from the others.
- *SymmetricReplicatesError:* All replicates have the same error. For example, the error was introduced before the service item was replicated.

We distinguish between the following asymmetric replicates errors:

- *AsymmetricValue*, with the alias *InconsistentValue:* The value of at least one replicated service item differs from the other replicates.
  - *AsymmetricExactValue:* The values of replicated service items are expected to be exactly the same.
  - *AsymmetricApproximateValue:* The values of replicated service items cannot differ by more than a threshold.
- *AsymmetricOmission*, with the alias *InconsistentOmission:* At least one replicated service encounters omission.
  - *AsymmetricItemOmission:* At least one of the replicates is missing (encounters an *ItemOmission*).
  - *AsymmetricServiceOmission:* At least one of the replicates is missing (encounters a *ServiceOmission*).
- *AsymmetricTiming*, with the alias *InconsistentTiming:* At least one of the replicated service items is delivered outside the expected time interval.

The type hierarchy for replication errors is shown graphically in Figure 19. Note that for *SymmetricReplicationError*, the subtypes *SymmetricValue*, *SymmetricTiming*, and *SymmetricOmission* are not shown.

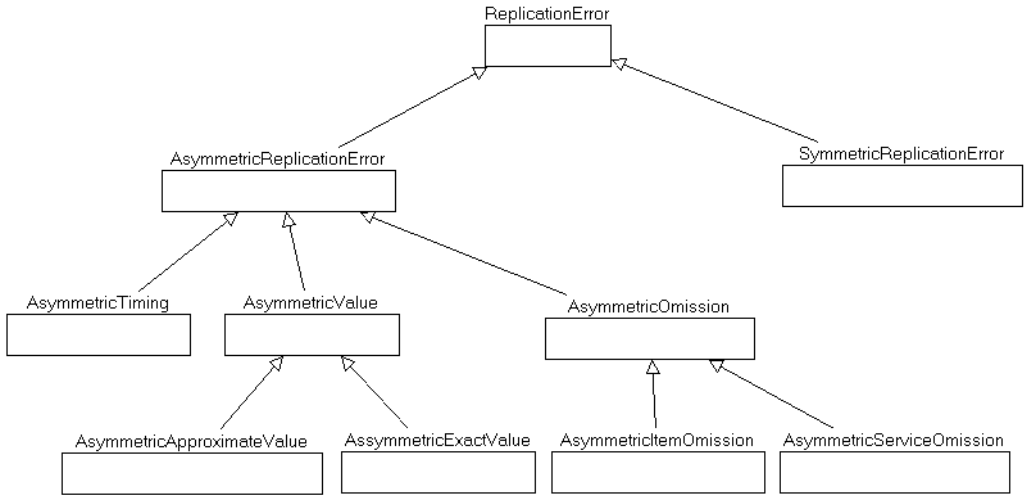

Figure 19: Hierarchy for Replication Error Types

A predeclared set of aliases for replication errors is shown in Figure 20.

```
InconsistentValue renames type AsymmetricValue;
InconsistentTiming renames type AsymmetricTiming;
InconsistentOmission renames type AsymmetricOmission;
InconsistentItemOmission renames type AsymmetricItemOmission;
InconsistentServiceOmission renames type AsymmetricServiceOmission;
AsymmetricTransmissive renames type AsymmetricValue;
```

Figure 20: Aliases for Replication Error Types

3.4.6 Concurrency-Related Errors

Concurrency-related errors (*ConcurrencyError*) address issues when concurrently executing tasks access shared resources. We distinguish between race conditions (*RaceCondition*) in the form of *ReadWriteRace* and *WriteWriteRace* and mutual exclusion errors (*MutExError*) in the form of *Deadlock* and *Starvation*. Figure 21 shows the concurrency error type hierarchy graphically.

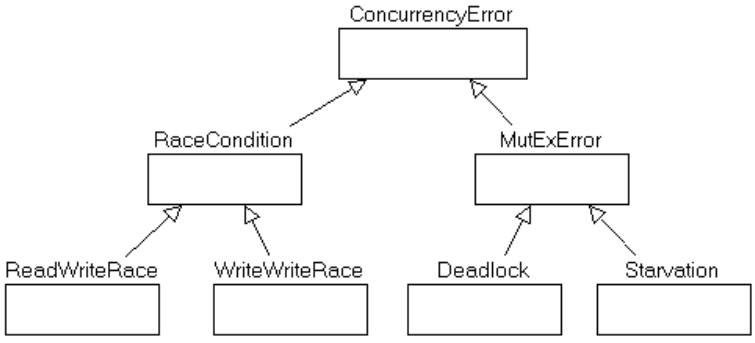

Figure 21: Hierarchy of Concurrency Error Types

3.4.7 Authorization- and Authentication-Related Errors

We have introduced two security-specific error type hierarchies so that EMV2 can be used to characterize security-related hazards (vulnerabilities). These error types complement the error types previously described.

Authorization-related errors (*AuthorizationError*) are related to access control. Authorization errors consist of privilege enforcement errors and privilege administration errors. Examples of authorization errors include ambient authority errors, privilege escalation errors, confused deputy errors, privilege separation errors, privilege bracketing errors, compartmentalization errors, least privilege errors, privilege granting errors, and privilege revocation errors.

Authentication-related errors (*AuthenticationError*) are related to authentication of services (roles, agents), information, and resources.

3.4.8 Using the Ontology as an Error Type Library Named *ErrorLibrary*

EMV2 provides a common set of error types as a standard error type library called *ErrorLibrary* (see Figure 22, which shows its inclusion in the Open Source AADL Tool Environment [OSATE] tool set). This error type library must be named in the **use types** declaration of a subclause so that the types will be accessible without qualification.

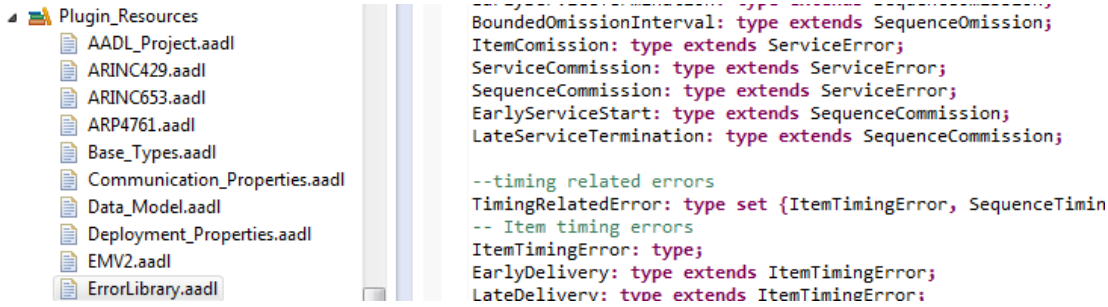

Figure 22: Predeclared Error Types in ErrorLibrary

This ontology is a good starting point for identifying the first-level effect as a propagated error type. Users can define component-specific aliases for the predefined error types and extend the type hierarchy of existing libraries by defining an error type library extension (see Section 3.2).

Users are not required to use the predefined ontology of error types. In particular, when defining error types to characterize various component-specific faults, users may define the error types independently. Users will then use state-transition declarations to specify the resulting error behavior states and outgoing propagation condition declarations to specify the resulting effect on other components in terms of error propagations.

4 Error Sources and Their Impact

This section introduces the concepts for error propagation specifications: *error propagation path*, *error propagation*, and *error flow*. Error propagation paths are determined by the connections between the components and by binding of components to other components within the AADL core model. Examples of bindings include binding of software to hardware and binding of elements of a functional architecture to a system architecture. In addition, users can introduce propagation paths that are not explicitly represented in the AADL core model. Error propagation between components occurs through *error propagation points*, such as the ports through which components interact, sources and targets of bindings, and user-defined propagation points. Propagation paths indicate which components receive propagated errors. Error flows represent abstractions declaring a component as a source or sink of error propagations or declaring how a component passes incoming error propagations on as outgoing propagations.

These concepts allow us to identify error sources as hazards and to understand their impact on other system components and on the operational environment of the system. Error propagation and error flow specifications can be used early in the development process and support early safety analysis activities such as FHA and FMEA. In addition, the concepts of error source, error path, and error sink align with the concepts found in the FPTC [Paige 2009].

4.1 Error Propagation Paths

4.1.1 Role of Error Propagation Paths

Error propagation paths indicate which components will receive outgoing error propagations. Error propagation paths are already defined in the AADL core model of a system. Connection declarations specify how interaction points of different components are interconnected; all port and access connections represent potential propagation paths. Similarly, an AADL core model may include binding specifications between components; they represent propagation paths between binding points of components. Finally, propagation paths may exist between components that are not explicitly represented in the AADL core model. Modelers can specify such paths in the **propagation paths** section of the error model subclause (see Section 8.2).

When a component failure results in an instance of an outgoing error propagation, its type instance is propagated along all error propagation paths whose source is the outgoing propagation point. The recipient component then responds to the incoming type instance according to its error flow specifications or according to its component error behavior specification, consistent with the error flow specification.

4.1.2 Using Error Propagation Paths

Error propagation paths between interaction points of software components are summarized in Table 1.

Table 1: Propagation Paths Between Software Components

Propagation occurs from	Propagation is received by
Thread, device, thread group, process, system, shared data component, or abstract component via outgoing port connection	Thread, device, thread group, system, shared data component, or abstract component via incoming port connection
Thread, device, thread group, process, system, or abstract component via directional data access connection or bidirectional connections with write access	Shared data component via access connection
Shared data component via access connection	Thread, device, thread group, process, system, or abstract component via directional data access connection or bidirectional connections with read access
Subprogram caller via access connection or call binding	Called subprogram and vice versa to reflect return from a call
Subcomponent within a process	Every other subcomponent within a process when there is no component interface and address space enforcement within that process

Error propagation paths between interaction points of hardware components are summarized in Table 2.

Table 2: Propagation Paths Between Hardware Components

Propagation occurs from	Propagation is received by
Device, memory, processor, system, or abstract component via directional bus access connection or bidirectional connections with write access	Bus or abstract component via bus access connection
Bus or abstract component via bus access connection	Device, memory, processor, system, or abstract component via directional bus access connection or bidirectional connections with write access

Table 3 summarizes error propagation paths based on bindings between hardware components as shared resources and the software components bound to them. It also summarizes bindings between a functional architecture and a system architecture.

Table 3: Propagation Paths Based on Bindings

Propagation occurs from	Propagation is received by
Processor and virtual processor based on *Actual_Processor_Binding*	Every thread, thread group, process, and virtual processor bound to the processor—and vice versa
Memory based on *Actual_Memory_Binding*	Every data component, thread, thread group, process, and port bound to the memory component—and vice versa
Bus, virtual bus, processor, device, and system based on *Actual_Connection_Binding*	Every connection and virtual bus bound to the component providing the transfer between the sender and receiver of a connection—and vice versa
System component based on *Actual_Function_Binding*	Every functional component bound to the system component—and vice versa

Connections themselves can be error sources, or they can propagate to or be affected by propagations from components that the connection is bound to. In other words, the error propagation to

the connection destination is affected by the error propagation from the connection source as well as the connection error source and propagations from connection binding. Section 8.5 discusses how the resulting error type on the connection destination is determined.

4.1.3 Observations

Software errors may lead to error propagation between subprograms and threads within the same process, or the same partition in a partitioned architecture, if there is no address space enforcement that ensures a subprogram or thread accesses other components only via its interaction points. In such a case, it is valuable to specify error propagations at the thread level to document thread-level expectations for incoming and outgoing interaction points, since no runtime guarantees can be made about fault containment within a process or partition.

When a shared resource, such as a processor, is an error source, it can affect all components bound to the resource. A shared resource with an incoming error propagation named in an error path results in an outgoing propagation to all components bound to the resource.

The outgoing error propagation from a component binding point to a connection affects the error propagation from the connection source to the connection destination. Error type transformation rules specify the resulting error type when an outgoing error propagation of a connection source is combined with an outgoing error propagation from the component performing the connection transfer (see Section 8.4).

4.2 Outgoing and Incoming Error Propagation Specification

Error propagation occurs between outgoing and incoming error propagation points of different components. In this section, we describe the role of incoming and outgoing error propagation declarations and how they are used.

4.2.1 Role of Outgoing and Incoming Error Propagation Declarations

An error propagation declaration identifies the error propagation point and specifies the type of error being propagated. Outgoing error propagation declarations specify the types of errors being propagated out of a component, while incoming error propagation declarations specify errors expected to be propagated into a component. Any time an error propagation occurs, one or more error types are specified for the instance in the error propagation declaration.

Error propagation points are

- the interaction points of components, declared as features of a component type. They include the different types of ports, data access, bus access, subprogram access, and subprogram group access features, and their aggregation into feature groups.
- binding points that reflect the deployment binding of software components to hardware components. Binding points are identified in the error propagation declaration by keywords.
- user-defined propagation points that are not present in the AADL core model. They allow the user to represent the impact of a component on other components or the environment without explicitly represented interaction points. An example is heat exposure of one processor to a physically close processor without the processor being connected. See Section 8.2 for details on user-defined propagation points.

The outgoing propagation declarations act as guarantees and the incoming error propagation declarations act as assumptions. When an outgoing propagation is connected to an incoming propagation through a propagation path, the error types are compared to ensure that outgoing error types are handled by the incoming error propagation declaration. Section 4.4 elaborates on the comparison rules for these error propagation guarantees and assumptions.

4.2.2 Using Outgoing and Incoming Error Propagations

Outgoing and incoming error propagations are declared in the **error propagations** section of the error model subclause. The declaration consists of a reference to an interaction point, a user-defined propagation point, or a keyword to identify a binding point. It is followed by a colon and the keyword **in** or **out** to identify the direction of the propagation. The error propagation declaration ends with a specification of one or more error types being propagated, enclosed in curly brackets. The syntax rules for error propagations are shown in Section 10.6.

An error propagation can be referenced by the name used to identify its error propagation point. Examples of such references can be found in the declaration of error sources, paths, and sinks (see Section 4.3.2).

Binding points are identified by the following keywords:

- **processor**: processor binding point for a software component or virtual processor, expressed by *Actual_Processor_Binding* property
- **memory**: memory binding point for a software component, expressed by *Actual_Memory_Binding* property
- **connection**: hardware binding point for a connection or virtual bus, expressed by *Actual_Connection_Binding* property
- **binding**: system binding point for a component in a functional architecture, expressed by *Actual_Function_Binding* property
- **bindings**: binding point for the target component of a binding through which all bound components can be reached

Figure 23 shows several examples of error propagation declarations for a software component. The process *MyApp* has an incoming error propagation for *sensorData* with the error type *BadData* and a similar outgoing error propagation declaration for *actuatorCmd*. The process also has an incoming error propagation (keyword **processor** with error type *NoService*) for errors propagated from the processor that *MyApp* is bound to. Finally, *MyApp* has an outgoing propagation declaration that associates an error type with one of the ports in the feature group *monitorControl*.

```
process MyApp
features
 sensorData: in data port;
 actuatorCmd: out data port;
 monitorControl: feature group Control;
annex EMV2 {**
  use types MyErrorLib;
  error propagations
    sensorData: in propagation {BadData};
    actuatorCmd: out propagation {MissingCmd};
       -- incoming error propagation from processor binding
```

```
    processor: in propagation {NoService};
    -- propagation on a feature group element
    monitorControl.reset: in propagation {NoService};
  end propagations;
**};
end MyApp;

feature group Control
features
  reset: in event port;
  shutdown: in event port;
end Control;
```

Figure 23: Examples of Error Propagation Declarations for a Software Component

Figure 24 shows error propagation declarations for hardware components. In the processor *PC*, the outgoing error propagation declaration uses the keyword **bindings**. This declaration corresponds to the incoming **processor** error propagation in *MyApp*. We also have an incoming error propagation on the bus access feature to *CANbus*. Since access connections can end directly with a bus, we must be able to identify this access interaction point on the bus and error propagation from the bus. As shown in the example, the keyword **access** is used for that purpose.

```
processor PC
features
       deviceBus: requires bus access CANbus;
annex EMV2 {**
  use types MyErrorLib;
  error propagations
       -- outgoing error propagation from processor binding
    bindings: out propagation {NoService};
    deviceBus: in propagation {NoService};
  end propagations;
**};
end PC;

bus CANbus
annex EMV2{**
  use types MyErrorLib;
  error propagations
       -- outgoing error propagation on access connections
    access: out propagation {NoService};
  end propagations;
**};
end CANbus;
```

Figure 24: Examples of Error Propagation Declarations for Hardware Components

4.2.3 Observations

For each incoming error propagation point, there can be only one incoming error propagation declaration, and for each outgoing error propagation point, there can be only one outgoing error propagation declaration. However, the declaration can list multiple error types for the same error propagation point to indicate that any of the specified error types may be propagated.

Some features—such as port, access, and abstract features—may be bidirectional, that is, both incoming and outgoing. In this case, separate error propagation declarations are used to specify the incoming error types and the outgoing error types.

If the error propagation point is a feature group, propagated error types can be specified for each element of the feature group. This is accomplished by specifying a path starting with the feature group name, followed by one dot-separated element name (or multiple element names for nested feature groups). For example, `actuatorInterface.cmdset.cmd: out propagation {MissingCmd};`

Error propagation specifications have two purposes:

1. to identify error sources and potential hazards. In this case, they specify outgoing error propagations together with error source declarations on the functional system architecture. This specification allows users to assess functional hazards (see Section 4.4).
2. to specify both outgoing and incoming error propagations, as well as corresponding error source declarations, with error paths and sinks for system components of concern. This specification helps users understand the impact of faults on other system components (see Section 4.6). Such specifications also establish contracts and assumptions between system components with respect to propagated and contained error types (see Section 4.4), which allows us to identify unhandled faults (see Section 4.7). This process is then repeated as the architecture model of a system is refined by decomposing system components.

4.3 Error Sources, Sinks, and Pass-Through

System components can be viewed as sources of errors that are propagated to other components, as sinks of propagated errors, or as passing propagated errors they receive on to other components. EMV2 supports the declaration of three forms of error flow: *error source*, *error path*, and *error sink*.

4.3.1 Role of Error Source, Error Path, and Error Sink Declarations

An *error source* identifies a component as the source of an outgoing error propagation, or component failure, whose effect on other components is represented by the identified error propagation. An error source specification may also indicate the failure source within the component, sometimes referred to as the failure mode.

An *error path* indicates that an incoming error propagation results in an outgoing error propagation. The error type of outgoing error propagation can be the same as the incoming error propagation—that is, the component passes the same error type to other components—or it can be a different type. For example, a component may recognize an incoming out-of-range value and send no value to the outgoing port.

An *error sink* indicates that a component is able to mask an incoming error propagation; that is, the error does not result in any outgoing propagation to other components. For example, a component may recognize an out-of-range value for a sensor reading and use previous values to send an approximation value.

4.3.2 Using Error Source, Error Path, and Error Sink Declarations

Error sources, paths, and sinks are declared in the **error propagations** section of the error model subclause. They are placed after the error propagation declarations, separated by the **flows** keyword. The syntax rules for error propagations and error flows are shown in Section 10.6.

An error source declaration consists of a name that identifies it and a reference to an outgoing error propagation point (or **all**). Optionally, the error source declaration may also have one or more error type constraints and a **when** or an **if** statement. The outgoing error propagation is referenced by the error propagation point identifier used in the error propagation declaration.

The error type constraint specifies a subset of error types in the error propagation for which the component is the source, by listing error types inside curly brackets (see Section 3.1). If the constraint is absent, all error types in the error propagation declaration apply.

The optional **when** statement allows the user to indicate a component failure that causes the error propagation. This component failure can be identified by an error type or an error behavior state, or simply characterized by a textual description. This component failure specification represents the failure mode in a fault impact analysis or FMEA.

The optional **if** statement allows the user to include a text statement describing the fault condition.

An example of an error source declaration is shown in Figure 25. A sensor provides sensor readings through its *SensorReading* port. Its outgoing error propagation specification indicates that the reading could be *BadData* or *NoData*. The error source declaration specifies that the sensor is the source of the error type *BadData*. The specified error type must be one of the error types specified in the error propagation.

The error source declaration also specifies that the sensor propagates bad data when it encounters a failure of type *SensorFailure*. Figure 25 shows how to provide a component failure description as text instead of using an error type.

```
device Sensor
features
   SensorReading: out data port;
   PowerSource: requires bus access;
annex EMV2 {**
 use types MyErrorLib;
 error propagations
  SensorReading: out propagation  {BadData, NoData};
  PowerSource: in propagation  {PowerFailure};
 flows
  ErrorSource: error source SensorReading {BadData} when {SensorFailure} if "Op-
erating in autoland mode";
-- ErrorSource: error source SensorReading {BadData} when "Transient Sensor Read-
ing Failure";
  ErrorPath: error path PowerSource->SensorReading {NoData};
end propagations;
**};
end sensor;
```

Figure 25: A Sensor with an Error Source and an Error Path

An error path declaration consists of a name that identifies the error path declaration; an incoming error propagation point as the source, followed by "->"; and an outgoing error propagation reference as the target.

- The incoming error propagation point can have an optional error type constraint that indicates the subset of error types from the incoming error propagation declaration. This type constraint is expressed as a type set using curly brackets. For example, `{NoData}`.

- The outgoing error propagation reference may be followed by an instance of an error type specified in curly brackets to indicate the specific error type being propagated. For example, `{NoData}`.

An error path declaration specifies that any error propagation instance matching one of the types specified in the type constraint is mapped into the type instance specified on the right-hand side.

If the error type constraint is absent, then any error type specified in the incoming error propagation is accepted as an incoming error propagation instance.

If the resulting type instance specification is absent, then the error type of the incoming instance becomes the error type of the outgoing instance. Those error types must still be contained in the type set specified for the outgoing error propagation.

The example in Figure 25 shows an error path specification that applies to all incoming error types for the error propagation point *PowerSource*. The result is an outgoing error propagation of type *NoData*.

An error sink declaration consists of a name that identifies it and a reference to an incoming error propagation point, as shown in Figure 26. The reference is optionally followed by an error type constraint, and it may refer to all incoming propagation points by using the declaration **all**. The constraint indicates that the error sink declaration applies to a subset of the error types specified in the referenced error propagation declaration.

```
ErrorSink: error sink SensorReading {BadData};
```

Figure 26: Example of an Error Sink Declaration

4.3.3 Observations

If a component has multiple outgoing error propagation points, all of them may be affected by a component failure. In this case, the user can declare an error source and use the keyword **all** instead of a specific error propagation reference, as shown in Figure 27.

```
ErrorSource: error source all{BadData} when {SensorFailure};
```

Figure 27: Example of an Error Source on All Outgoing Propagation Points

The keyword **all** can also be used in error path and error sink declarations. For error path declarations, **all** can be used instead of a specific incoming error propagation reference; this keyword indicates that any incoming error propagation matching the error type constraints, if specified, results in an outgoing error propagation. If **all** is also declared instead of the outgoing error propagation reference, then all outgoing propagations are affected by any incoming error propagation. If a specific error propagation is referenced as incoming and **all** is used as outgoing, then an instance of the specified error propagation is passed on through all outgoing error propagations, either with the incoming type instance or with the specified type instance.

An incoming error propagation can be referenced by both an error sink and an error path. In this case, the component is the sink for one error type and passes on a different error type. For example, a component may be a sink for *BadData* and pass on *NoData*.

A component can even be an error sink and an error path for an incoming error propagation of the same type. This represents situations where the component acts as a sink in some circumstances and passes on errors in others. For example, a component may handle *BadData* while fully operational, but pass on *BadData* if in a degraded error behavior state. The error propagation and error flow declarations specify the fact that both forms of error flow exist. We can associate a probability value with each to indicate an occurrence distribution between the two. The component error behavior specification will elaborate the specifics of how the component handles incoming error propagations; for example, the handling method is conditional based on the error behavior state (see Section 5).

An outgoing error propagation can be specified as both an error source and the target of an error path. This means that an error of a specified type is propagated by the internal failure of a component (error source) or by passing on an incoming error propagation (error path). For example, computational errors in a component may produce bad values, or bad values in the incoming data may cause the component to output bad values.

A component can be the error source and error sink for an error propagation of the same error type. For example, a component may recognize an out-of-range value and mask it by calculating an approximate value. The computation of correct input may be erroneous and introduce a data value error.

Note that if no error flows are specified, the fault impact analysis assumes a pessimistic flow. In this case, the component is the error source for all outgoing error propagations. Furthermore, every incoming error propagation is passed on through all outgoing error propagation points.

Users may use error flow declarations in conjunction with error propagation declarations in several ways:

- The user may initially focus on error propagations for which the component is the source. The user may specify several error types that are propagated through an error propagation point.
- The user may initially focus on the functional interface with other components—that is, the communication through ports—and later add bindings as propagation points.
- The user may identify a subset of the propagated error types as an error source.
- The user may further characterize the error source with a *Severity* property to mark different error propagation types of an error propagation or error source.

Declarations of error propagations and flows allow the user to record error types that may result in hazards. Users may elaborate this initial architecture fault model with incoming error propagations, error paths, and error sinks, leading to an understanding of fault impact on other system components.

Users may map the fault model of a functional architecture to a system architecture. This mapping helps designers understand the impact of a system component failure on system functions and vice versa. Users may further refine the architecture fault models by specifying component error behavior. Such a specification must be consistent with the specification of error propagations and flows in order to maintain validity of early safety analysis results and analysis results of the refined model. For more details, see *AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment* [Delange 2014].

4.4 Fault Propagation Contracts and Unhandled Faults

In Section 4.2, we introduced outgoing and incoming error propagation declarations to specify types of propagated errors. In this section, we introduce *error containment* declarations that allow us to specify error types that are not propagated. The combination of explicitly specified error propagations and containments supports a contract model between components and allows users to identify unhandled faults.

4.4.1 Role of Error Containment Declarations

Error containment declarations on outgoing error propagation points indicate that certain error types are not propagated out but contained within the component. Error containment declarations on incoming error propagation points indicate that certain error types are not expected to be received.

Because EMV2 supports the declaration of both error types being propagated and error types being contained, we can check for completeness of error propagation specifications. In this case, if certain propagation specification types are not specified, they are not misinterpreted as absent propagations (error containment). Instead, they are identified as incomplete specification—the user may not have specified that error type.

Figure 28 illustrates a fault model specification for a system interface expressed as a collection of contracts and assumptions about outgoing and incoming error propagation points. It shows an expected incoming propagation of type *OutOfRange*, which will be transformed into an outgoing propagation of type *NoData*, while ensuring that no outgoing *OutOfRange* error is propagated. The textual specification is shown in Figure 29.

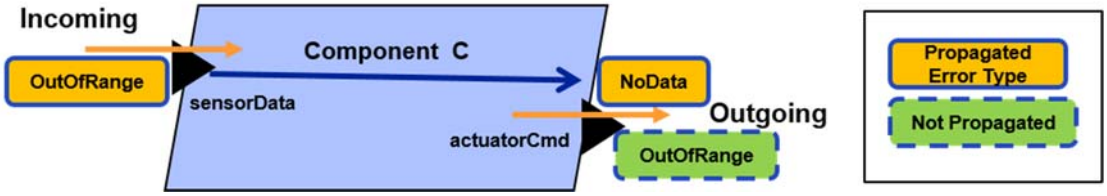

Figure 28: Fault Model Specification of a System Interface as Contracts, Assumptions, and Flows

4.4.2 Using Error Containment Declarations

Outgoing and incoming error containments are declared in the **error propagations** section of the error model subclause. An error containment declaration specifies the set of error types expected to be contained. Error containment declarations differ from error propagation declarations by the addition of the keyword **not** after the colon. The syntax rules for error containment declarations are shown in Section 10.6.

Figure 29 shows an example of an error containment declaration for the feature *actuatorCmd*. It declares that the component expects to filter incoming out-of-range values and not propagate them through the *actuatorCmd* feature. The error path specification *OORHandling* indicates that the component will handle the error type by not sending data.

```
error propagations
  sensorData: in propagation {OutOfRange};
  actuatorCmd: out propagation {NoData};
  actuatorCmd: not out propagation {OutOfRange};
flows
  OORHandling: error path sensorData{OutOfRange} -> actuatorCmd{NoData};
end propagations;
```

Figure 29: Example Error Containment Declaration

4.4.3 Observations on Propagation Guarantees and Assumptions

Outgoing error propagation and containment declarations specify the *guarantees* that a component makes about the propagation of various error types to components that interact with it. Incoming error propagations and containment declarations specify the *assumptions* that a component makes about propagations of various error types that it receives from other components.

Error propagation paths identify the source and target of interacting components. This relationship allows us to ensure that the propagation guarantees of components that are the source of propagation paths to another component meet the assumptions that component makes about incoming propagations. This is similar to checking the data types of connected data ports and event data ports. The rules for checking guarantees against assumptions are illustrated in Figure 30.

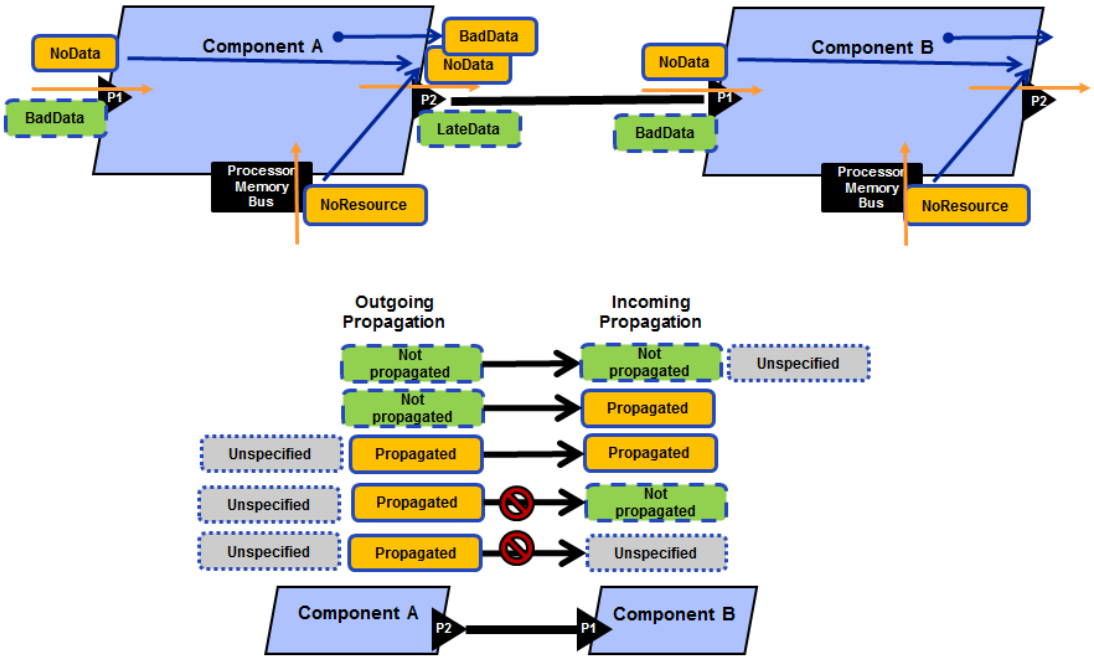

Figure 30: Matching Rules for Outgoing and Incoming Error Propagations

Outgoing error types must be contained in the incoming error type sets. Containment means that the outgoing set of error types must be smaller than the incoming set of error types. An outgoing set of contained error types must include any declared incoming error containment types, thereby clearly indicating that if an incoming error type is declared as contained, it will not be propagated. However, outgoing contained error types need not be specified as incoming contained error types.

Incoming *not propagated* error types must not contain outgoing error propagation types. Unspecified error propagations on incoming propagation points that receive outgoing error propagations are considered unhandled faults.

An outgoing *not propagated* error type can be contained in the error types of an incoming error propagation. In this case, even though the component does not expect them, it is robust to incoming error propagations of the specified type. For example, the implementation of a sending component may propagate an out-of-range error due to a coding error despite a specification indicating that it contains out-of-range errors, but the receiving component is prepared to deal with this compliance violation.

For type hierarchies, the outgoing error type must be a subtype of the incoming error type. For example, an outgoing error type *EarlyDelivery* is acceptable to the recipient if its error propagation specification indicates *TimingError*, since *EarlyDelivery* is a subtype of *TimingError*. However, an outgoing error type *TimingError* is not acceptable if the incoming error type constraint is *EarlyDelivery*.

Users can check error type matching rules on the declarative model when both ends of a connection declaration have an EMV2 annotation with an **error propagations** section. Although it is desirable, users do not have to declare EMV2 annotations at every level of the component hierarchy. For example, users may specify an EMV2 annotation for devices, but not for the system that contains the device. The device may be connected to a thread within a process in another system. If the thread has an EMV2 annotation, we check for consistency between the device and the thread error propagation and containment specifications. If the system that contains the process with the thread has an EMV2 annotation, we also check for consistency between the thread and system-level error propagation and containment specifications.

4.5 Error Sources Resulting in Hazards

A technique known as Functional Hazard Assessment (FHA)—not to be confused with Fault Hazard Analysis (see the Federal Aviation Administration *System Safety Handbook* [FAA 2010]) —is defined as part of SAE ARP4761 [SAE 1996]. FHA is a systematic examination of systems and subsystem functions to identify and classify the failure conditions of those functions according to their severity.

This process is supported by error propagation and error source specifications of the system or subsystems of interest. Users can annotate these specifications with properties relevant to the production of FHA reports.

EMV2 includes a set of properties that are defined in a property set called *EMV2*. One such property is *Hazards*. It allows modelers to provide descriptive hazard information within the model. See Section 7.4 for details on predeclared properties in EMV2.

The property values are associated with error sources and outgoing error propagations of components and can be specific to a particular error type. Section 7.1 explains how to associate properties with error model elements.

Figure 31 illustrates an example hazard specification. The *Hazards* property is associated with the error behavior state that is the error source. Such hazard specifications are characterized with severity and criticality.

```
device PositionSensor
features
  PositionReading: out data port;
flows
  f1: flow source PositionReading {Latency => 2 ms .. 3 ms;};
annex EMV2 {**
use types ErrorLibrary;
use behavior ErrorModelLibrary::Simple;
error propagations
  PositionReading: out propagation {ServiceOmission, ItemOmission, ValueError};
flows
  ef1: error source PositionReading when Failed;
end propagations;
properties
  EMV2::Hazards =>
    ([crossreference => "1.1.1";
       failure => "Loss of sensor readings";
      phases => ("all");
      severity => 1;
      likelihood => C;
      description => "No position readings due to sensor failure";
      comment => "Becomes major hazard, if no redundant sensor";
      ])
    applies to ef1.ServiceOmission;
**};
end PositionSensor;
```

Figure 31: Hazard Specification

The set of hazards to be reported is determined as follows:

- Each component instance in a system instance model that has an EMV2 subclause with an **error propagations** section is a candidate for hazard specifications. If the section contains error flow specifications, then every error source is a candidate if it has a hazard property. Otherwise, every outgoing error propagation is a candidate if the hazard property identifies it as a hazard.
- If the error source has a **when** clause, then the specified error behavior state, and optionally its error type, can be the hazard source. The hazard property is associated with the state by identifying the error source, followed by the state, in the **applies to** clause. The hazard property can also be associated with a specific error type of the state. Alternatively, the **when** clause can specify a type set. In this case, each type in the type set can be identified as a hazard, and the error source acts as the first level of failure effect.

Figure 32 shows a sample FHA report generated from an AADL model annotated with EMV2.

A	B	C	D	E	F	G	H	I
Component	Error	Crossreference	Functional Failure (Hazard)	Operation	Envir	Effects of Hazard	Severity	Criticality
PositionSensor	Failed on ef1	1.1.1	Loss of sensor readings	all		No position readings due to sensor failure	1	C
Actuator1	ServiceOmission on operation	1.1.2	Loss of actuator functionality	all		Failure to move actuator into desired position	Catastrop	Remote
Actuator2	Failed on ef2	1.1.3	Loss of actuator action	all		No action due to sensor failure	Critical	Remote

Figure 32: Sample FHA Report

For a full discussion of FHA modeling with AADL and EMV2, see *AADL Fault Modeling and Analysis* [Delange 2014].

4.6 Understanding the Fault Impact

Fault impact analysis utilizes error sources, paths, and sinks to identify error flow from incoming to outgoing error propagations and error propagation paths along connections and bindings; this analysis generates error propagation traces from error sources through the system to error sinks. It can do so for a system where all components are operational or where one or more components are already in a nonworking state. Propagation traces begin at components with error source declarations. The origin of an error source declaration becomes the failure mode, and the outgoing propagation is the first-level effect. Every outgoing error propagation represents another level of effect in the fault impact analysis. An example snippet of a trace report is shown in Figure 33.

Component	Initial Failure Mode	1st Level Effect	Failure Mode
PositionSensor	Failed	{ValueError} PositionReading -> Actuator1:ActCmd	Actuator1 {ValueError}
PositionSensor	Failed	{ValueError} PositionReading -> Actuator2:ActCmd	Actuator2 {ValueError}

Figure 33: Example of a Fault Impact Report

The fault impact analysis takes into account error propagation paths within the system's software, hardware, and physical components; binding propagation paths; and user-defined propagation paths. As it traces instances of error types through the propagation path graph, it may transform some error types into different error types based on the error path specifications, type mapping, and type transformation rules. Furthermore, if an incoming error type has subtypes and a component is sensitive to the subtypes, the analysis generates a separate propagation trace for each subtype.

For a full discussion of fault impact analysis, particularly FMEA, see *AADL Fault Modeling and Analysis* [Delange 2014].

4.7 Identifying Unhandled Faults

In the following example, we illustrate how a consistency check on error propagation types can identify mismatched assumptions in an architecture design. The example is representative of an actual occurrence in an aircraft system. Typically, a PSSA is performed early in the development process; since it is a manual process, it is often performed only once. During this activity, participants consider major hazards and their impact on a system.

In our example, an embedded GPS/inertial navigation system (EGI) provides airspeed data to a flight management system. Failure of the EGI results in a service omission by not providing airspeed data when it is expected. This is shown as a propagation of error type *NoData* from the EGI to the flight management system (FMS) in Figure 34. Note that users can define aliases for the predeclared error types that are more meaningful to the domain of the component. Modelers can also extend an existing type hierarchy and introduce new error type hierarchies.

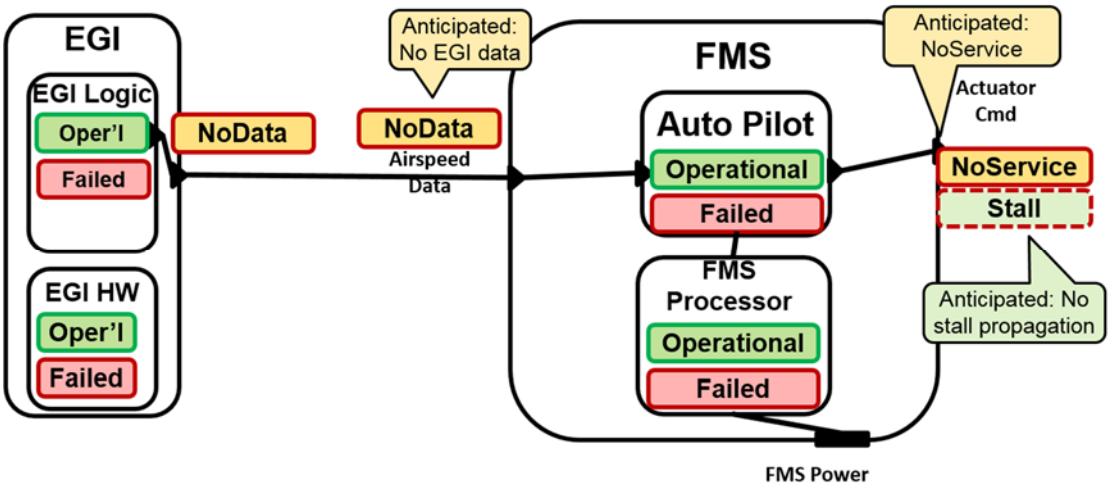

Figure 34: Error Propagations Between Subsystems

The EMV2 specification for the FMS shows that it recognizes a *NoData* propagation from the EGI. The specification also shows that the FMS is intended to operate as "fail silent" in that it either operates without functional failure, such as *Stall*, or it goes into fail stop, shown as *NoService*.

In our scenario, an engineer responsible for the EGI performs a lab test, encounters transmission of corrupted airspeed data, and identifies the root in the EGI hardware. One of two boards, which are positioned back to back with tight tolerances, is slightly out of spec. During a vibration test, the boards touched, causing transient corruption of the airspeed data being transmitted to the FMS. Under normal circumstances, the engineer assumes that such a case has been addressed in the PSSA or SSA. The engineer records the identified error and error propagation in the EMV2 specification for the EGI but takes no other action. The change of the specification is shown in Figure 35.

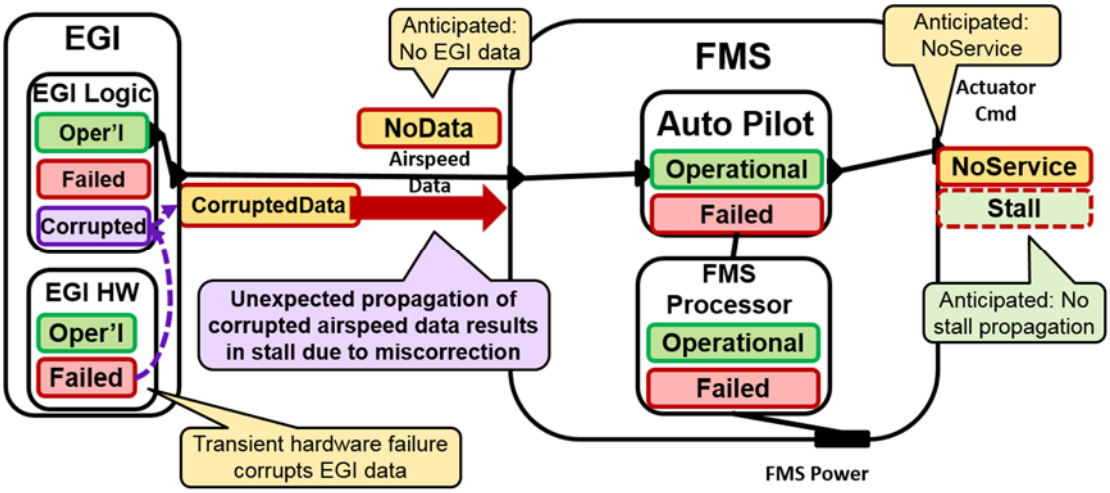

Figure 35: Updated Specification of Error Propagations

This specification is virtually integrated with the FMS specification as part of a routine virtual integration and analysis activity. EMV2 specifies consistency rules regarding outgoing and incoming propagated error types along error propagation paths (see Section 4.4.3). The enforcement of

this consistency rule by a tool identifies the mismatch between propagation assumptions and guarantees, and the tool reports it as an unhandled propagation, as shown in Figure 36.

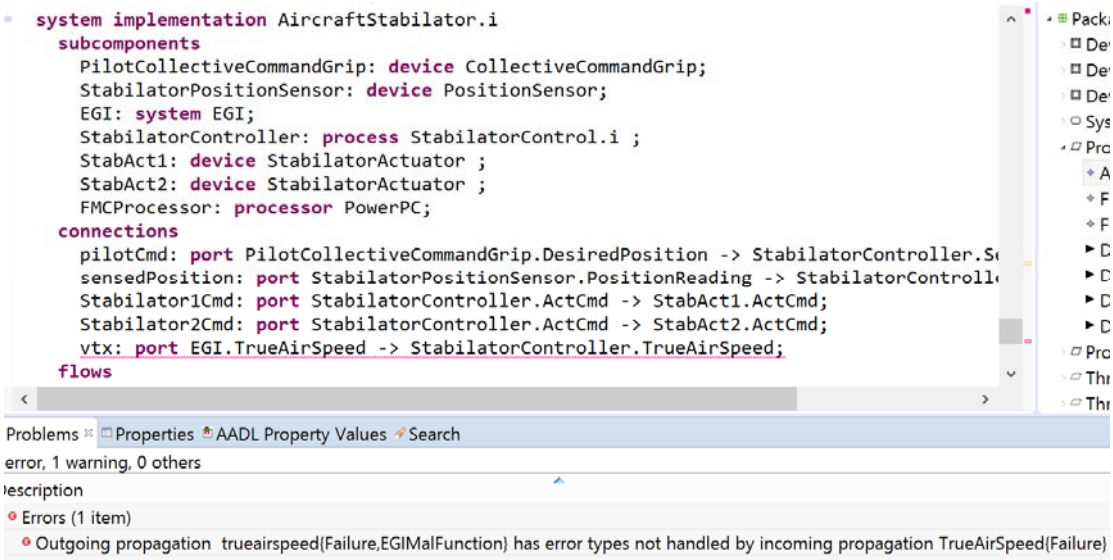

Figure 36: Mismatch Between Error Propagation Specifications

5 Component Error Behavior

This section introduces the concepts necessary to characterize the abstract error behavior of a component. A component error behavior specification represents the anomalous behavior of an AADL component using a state machine. Faults in a system are manifested as errors within the system or propagated to the system, and these errors can lead to failure, defined as behavior that does not comply with the intended service). A component error behavior specification is useful in representing component faults, the handling (or lack of handling) of faults, and the propagation of faults as errors. Component error behavior specifications can be used throughout a project life cycle, by developing them early in the life cycle and refining those models as the project progresses.

As a system architecture is refined one layer at a time, component error behavior specifications can be associated with subsystems. These error behavior specifications can then be related to the error behavior specification of the enclosing system to support compositional safety analysis (see Section 6).

EMV2 provides reusable error behavior state-machine declarations in error model libraries and component-specific error behavior declarations in error model subclauses. Component-specific error behavior declarations augment error behavior state-machine declarations with component-specific information, such as component-specific error behavior events, impact of incoming error propagations on the error behavior of the component, conditions under which component error behavior propagates errors, the intent for the system to detect certain errors, and the effect of error recovery or repair on the error behavior state of the system.

5.1 Reusable Error Behavior State Machines

An error behavior state machine defines a set of error behavior states and transitions, as well as error behavior events that can trigger the transitions. In this section, we describe the role and usage of each of these concepts, a library of predefined error behavior state machines, and the use of error types in error behavior state-machine declarations.

5.1.1 Role of Error Behavior States, Events, and Transitions

Error behavior states represent working states and failure modes. A working state indicates that the component is operational, while a nonworking state indicates that the component is erroneous (has malfunctioned or lost its function). A component can have one or more working states and one or more nonworking states.

An error behavior transition specifies a transition from a source state to a target state if a transition condition is satisfied. Transition conditions are the occurrence of error behavior events or incoming error propagations. Note that incoming error propagations are specific to components; thus, they can be specified as transition triggers only in component-specific error behavior declarations (see Section 5.2).

EMV2 supports the concept of a branching transition, which is a transition with multiple target states. Once a branching transition is triggered, the target state is determined according to a specified probability. Branching transitions allow users to specify that once an error has occurred, it

will result in a transient or persistent failure state without requiring users to introduce an intermediate error behavior state or separate error events for error occurrences with transient or persistent failure effects.

EMV2 distinguishes between three kinds of error behavior events: error events, recover events, and repair events. Error events represent fault activation within a component and result in a state transition to an error behavior state that represents the resulting failure mode in an outgoing error propagation. Recover events represent recovery from a nonworking state to a working state. They are used to model recovery from transient errors. Repair events represent repair action of longer duration, whose completion results in a transition back to a working state.

Separately declared error, recover, and repair events are considered to occur independently. Simultaneously occurring events may be handled in nondeterministic order. For example, users may declare one error event to represent the occurrence of out-of-range values and another error event to represent late delivery of data.

Users can also associate occurrence probabilities with error behavior events. An occurrence probability is declared in the **properties** section of the error behavior state machine, in which case it applies to all uses of the state machine. Component type-specific values can be declared as part of the component error behavior declaration in the error model subclause specified for a component type or component implementation. In this case, the value applies to all instances (subcomponents) of the classifier.

5.1.2 Using Error Behavior State-Machine Declarations

Figure 37 illustrates the declaration of an error behavior state machine in an error model library. The state machine is named *PermanentTransientFailure*. You can refer to the error behavior state machine in an error model subclause by qualifying it with the error model library name or the name of the package that contains the error model library.

```
package ExampleErrorLibrary
public
annex EMV2 {**
error behavior PermanentTransientFailure
events
  Failure: error event;
  Recovery: recover event;
states
  Operational: initial state;
  FailedTransient:  state;
  FailedPermanent:  state;
transitions
  FailTransition: Operational-[Failure]->
        (FailedTransient with 0.9, FailedPermanent with others);
  RecoveryTransition: FailedTransient-[Recovery]->Operational;
properties
  StateKind => Working applies to Operational;
  StateKind => NonWorking applies to FailedTransient, FailedPermanent;
end behavior;
**};
end ExampleErrorLibrary;
```

Figure 37: Reusable Declaration for an Error Behavior State Machine

The error behavior state machine consists of three error behavior states: *Operational*, *FailedTransient*, and *FailedPermanent*. Figure 37 shows that the *Operational* state is tagged as *Working* by the *StateKind* property; the other two states are tagged as *NonWorking*. Tagging with the *StateKind* property is optional.

An error behavior state machine can include one or more error events, recover events, or repair events. The example shows two error behavior events: an error event called *Failure* and a recover event called *Recovery*.

An error behavior transition specifies a transition from a source state to a target state when a trigger condition is met. The source state can be a specific state, possibly annotated with an error type constraint (see Section 5.1.4) or the keyword **all** to indicate that it represents a transition out of any error behavior state. The target state can be a specific error behavior state, possibly with a specific error type or the keywords **same state** to indicate that the state machine remains in the source state. The transition trigger can be one or more error behavior events or error propagations. Logical operators **or**, **and**, **ormore**, and **orless** are available to express the trigger condition for more than one trigger with a precedence ordering of **ormore**/**orless** over **and** over **or**. The latter two operators have the following syntax:

integer **<*operator*> (** *trigger reference* (**,** *trigger reference*)* **)**

The *Failure* event triggers a transition out of the *Operational* state. This transition is shown as a branching transition; *FailTransient* is the target state with a probability of 0.9, and *FailPermanent* is an alternative state with the remaining probability of 0.1. The sum of branch probabilities is expected to be 1.0. The *Recover* event triggers a transition out of the *FailTransient* state back to the *Operational* state.

5.1.3 Predefined Set of Error Behavior State Machines

The EMV2 standard includes several predeclared error behavior state machines. They are defined in the error model library *ErrorLibrary*, which also contains the predeclared error types. All predeclared error behavior state machines include an *Operational* state and a *Failure* error event. Users can associate a probability of occurrence with the *Failure* error event for a specific component through a property in the error model subclause. The following error behavior state machines are included:

- The *FailStop* error behavior state machine represents the error behavior of components whose failure occurrence (*Failure* error event) results in the *FailStop* error behavior state without recovery back to *Operational*. The effect of the *FailStop* on other components can be an outgoing propagation of service omission, which will be specified in the component-specific declaration of error behavior.
- The *DegradedFailStop* error behavior state machine represents the error behavior of components whose first failure occurrence (*Failure* error event) results in the *Degraded* error behavior state and whose second *Failure* error event results in the *FailStop* error behavior state. There is no recovery included in the state-machine specification.
- The *FailAndRecover* error behavior state machine represents the error behavior of components that have a failure (*Fail* state) and are able to recover to full operation. The recovery is

modeled by a *Recovery* recover event. Users can associate a probability of occurrence with the error and recover events.

- The *DegradedRecovery* error behavior state machine represents the error behavior of components that operate in degraded mode after one failure (*Degraded* state), are able to recover from *Degraded* to *Operational*, and enter the *FailStop* state if a failure occurs while in the *Degraded* state. The recovery is modeled by a *Recovery* recover event. Users can associate a probability of occurrence with the error and recover events.
- The *PermanentTransientFailure* error behavior state machine represents the error behavior of components that have transient (*FailedTransient* state) and permanent failures (*FailedPermanent* state). Users can specify component-specific values for the proportion of transient failures through the *EMV2::TransientFailureRatio* property associated with the transition. Figure 37 shows this error behavior state machine, but with a specific value of 0.9 supplied for the branch to the *FailedTransient* state instead of a reference to *EMV2::TransientFailureRatio*. Users can also specify component-specific values for the occurrence probability of recovery to transition back to *Operational*.
- The *FailRecoveryFailure* error behavior state machine represents the error behavior of components that have a failure (*Failed* state) and with specified probability are able to recover to full operation. This recovery may fail and result in the *FailStop* state. Users can specify component-specific values for the proportion of recoveries that fail through the *EMV2::RecoveryFailureRatio* property.

5.1.4 Typed Error Behavior State Machines

Error types can be associated with error events and error behavior states in the same way they are used on error propagations. This allows users to specify error behavior in a more compact form. The error types act as typed tokens on the state machine.

Figure 38 graphically shows an error behavior state machine with one operational state and three *NonWorking* states, one for each type of failure (*BadValueState*, *NoValueState*, and *LateValueState*). Transition to each of those nonworking states is triggered by a separate error event (shown as associated with the transition by a dashed arrow). Note that the names of these events and resulting states are chosen to indicate the type of error that results in the transitions shown.

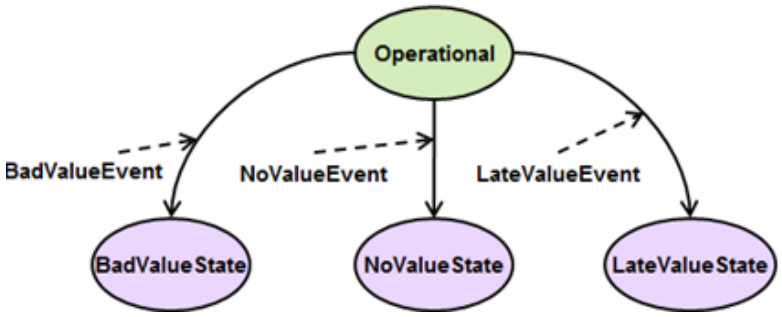

Figure 38: Example Model of an Error Behavior State Machine

The textual specification for the error behavior state machine is shown in Figure 39.

```
package UntypedExampleLibrary
public
annex EMV2 {**
error behavior UntypedStateMachine
events
  BadValueEvent: error event;
  NoValueEvent: error event;
  LateValueEvent: error event;
states
   Operational: initial state;
   BadValueState: state;
   NoValueState: state;
   LateValueState: state;
transitions
   BadValueTransition: Operational-[BadValueEvent]->BadValueState;
   NoValueTransition: Operational -[NoValueEvent]-> NoValueState;
   LateTransition: Operational-[LateValueEvent]->LateValueState;
end behavior;
**};
end UntypedExampleLibrary;
```

Figure 39: Untyped Specification for an Error Behavior Model

Figure 40 shows the same error behavior state machine, using error types on error events and error behavior states. Three error types are grouped into a type set called *MyTypes*. In the error behavior state-machine declaration, **use types** indicates the desire to reference a type in the error type library. A reference to the type set *MyTypes* is associated with the error event *FailEvent*. This indicates that error events of any of the three types may occur. Similarly, *MyTypes* is associated with the error behavior state *FailedState*. This indicates that any of the three types in the type set are acceptable. The transition *FailTransition* refers to the error event as a trigger and the target state *FailedState* without identifying an error type. This means that the error type of the error event instance becomes the error type of the error behavior state.

```
package TypedExampleLibrary
public
annex EMV2 {**
error types
   NoValue: type;
   BadValue: type;
   LateValue: type;
   MyTypes: type set { NoValue, BadValue, LateValue };
end types;

error behavior TypedStateMachine
use types TypedExampleLibrary;
events
   FailEvent: error event { MyTypes };
states
   Operational: initial state;
   FailedState: state { MyTypes };
transitions
   FailTransition: Operational -[FailEvent]-> FailedState;
end behavior;
**};
end TypedExampleLibrary;
```

Figure 40: Error Behavior State Machine with Error Types

5.1.5 Observations

Typically you will define only a small number of reusable error behavior state machines.

An error behavior state machine may consist of only error behavior states, or it may include error behavior events and transitions that are triggered by the error behavior events.

You can specify the error event with a particular error type as trigger. This error type must be one of the error types specified in the error event declaration. You can also specify a particular error type for the target error behavior state. For example, you can specify that several error types of error events or several different error events result in a transition to an error behavior state with a particular error type.

You will refine these reusable error behavior state machines with component-specific information in error model subclause declarations, as described in the next section.

5.2 Component-Specific Error Behavior Specification

In this section, we describe how to specify component-specific fault behavior. In the next section, we describe how to specify fault detection and recovery behavior for specific components.

5.2.1 Role of Component-Specific Error Behavior Specifications

Component-specific error behavior declarations are associated with component types and implementations through the error model subclause. The declaration identifies the error behavior state machine to be used and refined.

You add component-specific error behavior events to reflect fault occurrences that are specific to the component. For example, a sensor may produce a bad sensor reading. You then add transitions as appropriate to indicate that such an error behavior event results in a change of error behavior state.

You also specify conditions under which the component is an error source. That is, you specify whether an error behavior state results in the component propagating an outgoing error.

You can also specify how the component responds to incoming error propagations. You do so in two ways. First, you specify whether an incoming error propagation affects the component's error behavior state. This is not always the case. For example, an incoming value error may just be passed on without changing the error behavior state of the component.

Second, you can specify in more detail than in the error path declaration how the component responds to an incoming error propagation (detailed in Section 4.3). A component may respond to an incoming error propagation differently depending on its error behavior state. For example, a component may be able to mask an incoming value error propagation while in an operational state but pass it on as a missing data item when in a degraded error behavior state.

5.2.2 Using Component-Specific Error Behavior Specifications

The component-specific error behavior declaration utilizes the **use types** and **use behavior** declarations in the error model subclause (see Section 2.2). The component-specific error behavior declaration itself uses the keywords **component error behavior** and ends with **end component**. It consists of

- a **use transformations** declaration to indicate the type transformation sets to be used when the resulting types are not explicitly declared (see Section 8.5)
- a set of component-specific error behavior event declarations beginning with the keyword **events**
- a set of component-specific transition declarations beginning with the keyword **transitions**
- a set of outgoing error propagation condition declarations beginning with the keyword **propagations**
- a set of error detection declarations beginning with the keyword **detections**, which indicates that the component is expected to detect the specified error state or condition (see Section 5.2.3)
- a **mode mappings** declaration to specify how error behavior states (failure modes) relate to operational modes in the AADL core model (see Section 8.6)

Error behavior event declarations use the same syntax as error behavior state-machine declarations. They represent errors that are specific to the component type for which the error model subclause is declared.

Error behavior transition declarations use the same syntax as error behavior state-machine declarations. In this case, you can refer to incoming error propagations in addition to error behavior events as transition triggers. This allows you to indicate that an incoming error propagation affects the component behavior in succeeding executions by reflecting it in a transition to a different error behavior state.

Outgoing error propagation condition declarations use a syntax similar to transitions; see Figure 41 for an example. The declaration consists of a source error behavior state, possibly annotated with an error type constraint or the keyword **all**, followed by a possibly empty condition involving error behavior events or incoming error propagations within the symbols **–[ ]->**, followed by an outgoing error propagation reference and optionally an error type or **NoError**. The keyword **all** can be used instead of an outgoing error propagation to indicate that the condition applies to all outgoing error propagations. An outgoing error propagation declaration with an empty condition indicates that the source error behavior state results in the specified outgoing propagation of the specified type. The example includes an outgoing error propagation condition to represent the fact that a flight controller in the *FailStop* state will result in service omission.

```
system FlightController
features
  CurrentAirSpeed: in data port;
  NewAirSpeed: out data port;
annex EMV2 {**
   use types ErrorLibrary;
   use behavior ErrorLibrary::FailStop;

   error propagations
     CurrentAirSpeed: in propagation {BadValue};
     NewAirSpeed: out propagation {BadValue, ServiceOmission};
   end propagations;

   component error behavior
   events
      IncorrectComputation: error event;
   propagations
      Transient: Operational -[IncorrectComputation
                  or CurrentAirSpeed {BadValue}]-> NewAirSpeed {BadValue};
      Failed: FailStop -[]-> NewAirSpeed {ServiceOmission};
   end component;
**};
end FlightController;
```

Figure 41: Example of a Component-Specific Error Behavior Declaration

You can specify that incoming error propagations are passed on as outgoing error propagations only when the component is in a particular error behavior state by specifying the state and the appropriate incoming error propagation(s) as a condition. You can also specify that an incoming error is not propagated by specifying **NoError** (or **all**) for the outgoing error propagation point.

Finally, you can specify propagation of a transient error event, an error event that does not change the error behavior state. The example in Figure 41 shows an error event representing an incorrect computation. This error occurs occasionally as specified by an occurrence probability, but it does not affect succeeding computations. Therefore, it only propagates a *BadValue* at the time the error event occurs.

5.2.3 Observations

When error type constraints are specified in transitions and outgoing error propagation conditions, these constraints must be consistent with error types specified as part of the referenced element. For example, in Figure 41 the error type associated with the reference to the incoming error propagation *CurrentAirSpeed* must be contained in the set of error types specified in the error propagation declaration. Similarly, the error type specified for *NewAirSpeed* in the outgoing error propagation condition declaration must be contained in the set of error types specified in the error propagation declaration.

The error model subclause of a component may consist of error propagation and flow declarations as well as component error behavior declarations. These declarations are expected to be consistent with each other. For example, if an error sink has been declared for an incoming error propagation, then the component error behavior specification cannot declare an outgoing error propagation as a result of this incoming error propagation.

5.3 Error Response and Fault Tolerance

In this section, we present the use of EMV2 in modeling the fault response by the application systems. The resulting model contains the expectations on the fault management component of a safety-critical system. This includes identifying the component responsible for detecting a failure condition, specifying assumptions about the use of redundancy to provide fault tolerance, and specifying the impact of an actual system recovery on the error behavior state of the system.

5.3.1 Role of Error Detection, Transition, and Propagation Conditions and Recovery or Repair Events

In Section 5.1.3, we introduced a set of predeclared error behavior state machines. Some of these state machines represent the error behavior of a component with respect to errors as well as recovery or repair action. Figure 42 presents one of the predeclared error behavior state machines (*DegradedRecovery*) from the *ErrorModel* error model library. This state machine includes both failure behavior and recovery behavior. A single *Failure* event results in a *Degraded* state. With a user-specified occurrence probability, the component recovers to the *Operational* state. A second *Failure* event results in the *FailStop* state, from which there is no recovery.

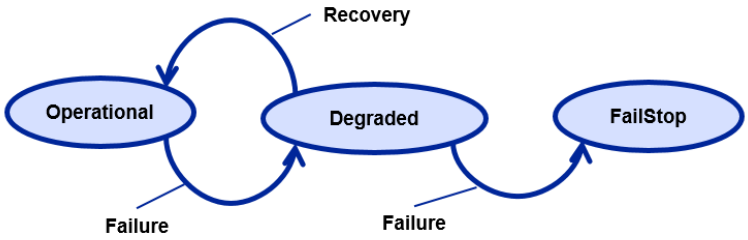

Figure 42: Representation of the DegradedRecovery *Error Behavior State Machine*

You can use the predeclared *FailRecoveryFailure* error behavior state machine to represent the fact that a recovery action itself may fail. If it is necessary to represent the duration of a recovery action, you can associate the *EMV2::DurationDistribution* property with a recover event to indicate how long the recovery takes without introducing an intermediate error behavior state. You can also define a new error behavior state machine that has an explicit *InRecovery* error behavior state.

When declaring a recover or repair event, you can identify a model element in the AADL core model that represents the recovery action. Use a **when** clause that lists one or more mode transitions, ports, or internal events.

The component-specific error behavior declaration supports error detection declarations (see Section 5.2). This declaration allows you to specify that a component is expected to detect the specified error state or condition. This component might not be the one with the failure but a component that receives the error propagation due to the failure. As part of the error detection declaration, you specify an internal event or a port together with an error code through which the detected failure is reported.

Finally, the logical expression of error behavior transitions and outgoing error propagation conditions allows you to specify redundancy assumptions about the fault management. For example, a component may receive sensor readings from two different sensors. You can specify that failure

to receive readings from one sensor may result in degraded operation with less precise control, while failure to receive readings from both sensors results in the inability to provide service.

5.3.2 Using Error Detection, Transition, and Propagation Conditions and Recovery or Repair Events

Error detection declarations have a syntax similar to outgoing error propagation condition declarations (see Section 5.2.2). The declaration consists of a source error behavior state, possibly annotated with an error type constraint or the keyword **all**; followed by a possibly empty condition involving error behavior events, incoming error propagations, or outgoing error propagations of subcomponents within the symbols **-[ ]->**; followed by an outgoing port reference or an internal event reference with an optional error code.

Figure 43 illustrates the use of error detection declarations and logical conditions to express redundancy assumptions. The example is a flight controller that gets airspeed information from two sources. The first transition declaration specifies that the system enters the *Degraded* error state if one airspeed input is absent (**or**). The second transition declaration indicates that the system enters *FailStop* from any error behavior state if both airspeed inputs are absent (**and**). The declarations for the outgoing error propagation condition indicate that in *Degraded* approximate values are delivered, while in *FailStop* all output is omitted. Finally, the error detection declarations indicate that the flight controller is expected to detect failure of one or both airspeed sources to provide data and report it with the appropriate error code. In the example, we assume that an error type library called *FCErrorLibrary* defines the error type *ApproximateValue*.

```
system DualSensorFlightController
features
  CurrentAirSpeed1: in data port;
  CurrentAirSpeed2: in data port;
  NewAirSpeed: out data port;
  Status: out data port;
annex EMV2 {**
   use types ErrorLibrary, FCErrorLibrary;
   use behavior ErrorLibrary::DegradedFailstop;

   error propagations
     CurrentAirSpeed1: in propagation {ServiceOmission};
     CurrentAirSpeed2: in propagation {ServiceOmission};
     NewAirSpeed: out propagation {ServiceOmission};
   end propagations;

   component error behavior
   transitions
      SingleFailure: Operational -[ CurrentAirSpeed1{ServiceOmission}
                  or CurrentAirSpeed2{ServiceOmission}]-> Degraded;
      DualFailure: all -[ CurrentAirSpeed1{ServiceOmission}
                  and CurrentAirSpeed2{ServiceOmission}]-> FailStop;
   propagations
      LowPrecision: Degraded -[]-> NewAirSpeed {ApproximateValue};
      Failed: FailStop -[]-> NewAirSpeed {ServiceOmission};
   detections
      ReportLowPrecision: all -[ CurrentAirSpeed1{ServiceOmission}
              or CurrentAirSpeed2{ServiceOmission}]-> Status(LowPrecisionCode);
      ReportNoSensors: all -[ CurrentAirSpeed1{ServiceOmission}
              and CurrentAirSpeed2{ServiceOmission}]-> Status(NoSensorCode);
   end component;
```

```
**};
end DualSensorFlightController;
```

Figure 43: Example with Error Detection and Redundancy Logic Declarations

5.3.3 Observations

The combination of error detection declarations and specification of conditions for the occurrence of recover events in terms of AADL modes, events on ports, or internal events links the error behavior specified through EMV2 with the implementation of fault management in the actual system as represented by the AADL core model. In addition, the condition logic represents assumptions made about a redundancy policy used by the fault management mechanisms in the actual system. Results from safety analysis, including reliability and availability analysis, use these assumptions. Thus, those results are valid only if the system meets these assumptions. In other words, we must ensure that the fault management implementation reflected in the AADL model is consistent with the EMV2 annotations.

6 Compositional Abstraction of Error Behavior

EMV2 supports the specification of error behavior for a system or component as a black-box abstraction. It does so through error propagation (Section 3) and error behavior (Section 5) specifications for a system or a system component. This abstraction is used in determining the architecture fault model of a system in terms of the fault model of its components.

In this section, we introduce the composite error behavior specification to define the condition for a system error behavior state expressed in terms of the error behavior states of its subcomponents. This allows the user to specify the intended relationship of externally visible system error behavior to the error behavior of its parts. Such a specification must be consistent with the combined error behavior of the connected subcomponents. A tool that checks these consistencies can also derive the composite error behavior specification or error flow specification in the black-box abstraction from the flow through the connected subcomponents.

Figure 44 illustrates an abstraction of error behavior. A flight guidance system (FGS) is shown on the left-hand side as a black-box abstraction with its abstracted error behavior and propagation specification. It has an operational state and a *Failed* state. On the right-hand side, the realization of the FGS is shown in terms of its parts: a flight guidance (FG) component, an auto pilot (AP) component, and an actuator (AC) component. FG and AP are replicated for redundancy. Each component has its own abstracted error propagation and behavior specification.

In addition, incoming error propagations to FGS must be consistent with those components of FGS that receive its input; that is, they accept the same set of error types. Similarly, the outgoing propagations of any component interacting with other components through the interface of the enclosing system must be consistent with the outgoing error propagation specification of FGS. In other words, the FGS propagation specification must cover the error types declared by all components connected to the outgoing error propagation point.

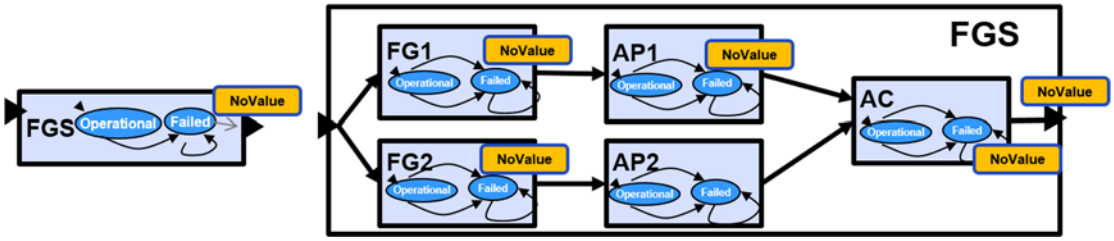

Figure 44: Flight Guidance System Fault Model at Two Levels of Abstraction

6.1 Composite Error Behavior Specification

6.1.1 Role of Composite Error Behavior Specification

A composite error behavior specification explicitly records the relationship between the error behavior states of a system and the error behavior states of its parts. A composite error behavior specification expresses the abstracted error behavior states of the composite component as a logical condition in terms of its subcomponent error behavior states. For example, an operational error

behavior state may reflect the fact that a component with redundant subcomponents can continue to be operational even though one of its parts has failed.

The composite error behavior specification reflects the fault tree logic based on the error behavior states of the subcomponents. The composite error behavior can be used to generate an FTA as an AADL component [Delange 2014].

The composite error behavior specification also allows us to derive reliability models, such as mean time to failure, of a system or subsystem in terms of its subcomponent probabilistic error behavior. The probability of FGS being in the *Failed* state is determined by a combination of the probability of the subcomponents being in the *Failed* state according to the logic expression of the composite error behavior state specification. This specification is equivalent to a reliability block diagram [Delange 2014].

6.1.2 Using Composite Error Behavior Specifications

Composite error behavior specifications are declared in the **composite error behavior** section of an error model subclause. As the composite error behavior specifies the error behavior state from the subcomponents' states, it should be declared in a subclause associated with a component instance. The composite error behavior section consists of a list of composite state declarations. An individual composite state declaration defines the error behavior state of the component in terms of a condition on subcomponent error behavior states. The condition is enclosed within the brackets of the symbol **[ ]->**, followed by the error behavior state name and an optional type instance of the state for which the condition is defined. The condition itself is a logical expression using **and**, **or**, **ormore**, and **orless** operators. The syntax rules for composite state declarations are shown in Section 10.8.

The composite state condition can also reference an incoming error propagation to the component for which the composite state is defined. In this case, the reference to the error propagation is preceded by the keyword **in** and followed by a type constraint, which includes **NoError** to indicate that no incoming propagation is expected. This allows us to reflect the impact of an external failure on the operational state of the component.

Figure 45 shows the composite error behavior specification for the example in Figure 44. It takes into account that each FG–AP pair is an element in a dual-redundant configuration such that it is sufficient for only one pair to be operational in order for FGS to stay operational. The example shows the condition under which FGS is in the *Failed* state and in the *Operational* state. The **ormore** operator is true if one or more of the listed elements are true.

```
composite error behavior
states
  [1 ormore(FG1.Failed, AP1.Failed) and
   1 ormore(FG2.Failed, AP2.Failed) or AC.Failed]-> Failed;

  [(FG1.Operational and AP1.Operational) or
   (FG2.Operational and AP2.Operational) and AC.Operational]-> Operational;
end composite;
```

Figure 45: Composite Error Behavior Specification

The error behavior state for the component (declared to the right of the arrow) must be one of the states listed in the error behavior state machine identified by the **use behavior** declaration earlier in the error model subclause. The error behavior states in the condition (between the square brackets) must exist in the error behavior state machine associated with the subcomponent (declared in the error model subclause of the component type or implementation associated with the subcomponent). The condition can also include a reference to an incoming error propagation, indicated by the keyword **in**, and it must exist as an incoming error propagation of the component for which the composite state condition is declared.

The error behavior state of a subcomponent can have an error type associated with it. The reference to the subcomponent's error behavior state may include a type set, which indicates that any error type in the type set is considered a match and will evaluate to true.

The error behavior state of the composite component may also be typed. In this case, the composite component state on the right-hand side is followed by a type instance declaration of an error type in curly brackets. An example of a composite behavior specification with error types can be found in Figure 79 in Section 9.1.2.

6.1.3 Observations

The **ormore** and **orless** operators provide a convenient way of specifying *m out of n* redundancy strategies as well as processing chains that require all steps to be operational.

A composite error behavior specification may be explicitly declared by a safety analyst to reflect a particular fault tree for analysis. In the FGS example, the composite state condition shown in Figure 45 indicates that one of the FG–AP pairs is sufficient to maintain the *Operational* state. This becomes the requirement specification for the system implementation to meet because it reflects the assumptions made about redundancy that the implementation must meet.

In Section 9.1, we elaborate the FGS specification with component error behavior for each FGS subcomponent. The component error behavior must refer to the same error behavior states identified in the composite state condition. Furthermore, the component error behavior of the AC subcomponent includes conditions on error behavior state transitions and outgoing propagations that reflect the intended redundancy strategy. These conditions must be consistent with the conditions for the composite state. In our example, the condition must reflect the fact that input from one of the two AP components is sufficient for AC to operate correctly by specifying that AP1 and AP2 must fail for FGS to fail.

7 Use of Properties in Architecture Fault Models

Properties can be associated with error model elements—such as error events, error propagations, and error behavior states—to annotate them with additional information. EMV2 provides a set of predeclared properties in a property set named *EMV2*. In addition, users can define their own properties for error model elements, as they can for AADL core model elements. In this section, we explain how property associations are declared and how users can define properties that apply to error model elements, and we summarize the set of predeclared properties.

7.1 Property Associations on Error Model Elements

Property values can be associated with the following named elements in the error model: error types, type sets, error propagations, error sources, error paths, error sinks, error events, recover events, repair events, error behavior states, transitions, outgoing propagation conditions, error detections, composite states, user-defined propagation points, and paths.

A property association looks like a contained property association in the AADL core model. It specifies the property name, a single value or a list of values of the appropriate property type, and an **applies to** clause. The **applies to** clause identifies one or more elements. Each element is identified by a containment path.

A containment path consists of a sequence of one or more EMV2 identifiers separated by dots. If the error model element resides in a subcomponent, then the containment path has two parts. The first part identifies the subcomponent by a sequence of identifiers separated by dots (standard AADL containment path syntax). The second part consists of the **@** sign and a sequence of one or more dot-separated identifiers of error model elements.

An error model can have a type set as part of its definition; for example, an incoming propagation is associated with the *NoValue* and *BadValue* error types. If an element in the error model is typed, then a different property value can be associated with each error type or type set specific to the element. This is done by naming the error model element, followed by a dot and the error type or type set name. For example, if we declare an error event *e* to represent *TimingError*, then we can associate a different occurrence distribution for an error event instance of type *EarlyDelivery* and of type *LateDelivery*. In this case, the **applies to** clause of each property association refers to *ev.EarlyDelivery* and *ev.LateDelivery*.

Property associations for elements in the error model must be declared in **properties** sections within an error model library or an error model subclause. Error model libraries have a **properties** section in the error type library (see Section 10.2) and in each error behavior state machine (see Section 10.4). Error model subclauses have a single **properties** section at the end of the subclause.

For property associations in an error type library, the containment path consists of an error type or a type set. Figure 46 shows two different *Persistence* property values associated with different error types.

For property associations in an error behavior state machine, the containment path can include an error event, recover event, repair event, error behavior state, or transition. For error events and error behavior states, the second element in the path may be an error type or a type set. Figure 46 shows an *OccurrenceDistribution* property value associated with the error event *Failure*.

```
annex EMV2 {**
error types
  NoValue : type ;
  BadValue : type ;
  LateValue : type ;
  NoService : type ;
properties
  EMV2::Persistence => Transient applies to BadValue, LateValue, NoValue;
  EMV2::Persistence => Permanent applies to NoService;
end types ;

error behavior Simple
events
  Failure : error event;
states
  Operational : initial state;
  Failed : state;
transitions
  BadValueTransition : Operational -[ Failure ]-> Failed;
properties
  EMV2::OccurrenceDistribution =>
    [ ProbabilityValue => 0.1 ; Distribution => Fixed;] applies to Failure;
end behavior ;
**};
```

Figure 46: Property Associations in an Error Model Library

For property associations in error model subclauses, the containment path may identify an error model element of the component that contains the subclause, or it may identify an error model element for a subcomponent. An error model element is identified by its defining identifier, optionally followed by an error type or a type set.

Figure 47 shows the property association *Likelihood* for an error source (*ef1*) of error type *ServiceOmission*. The property *Likelihood* is declared in the property set *EMV2*, and the label *Remote* is defined in the property set *ARP4761*.

```
device PositionSensor
  features
    PositionReading : out data port;
  annex EMV2 {**
    use types ErrorLibrary;
    error propagations
      PositionReading : out propagation {ServiceOmission};
    flows
      ef1 : error source PositionReading {ServiceOmission};
    end propagations;
    properties
      EMV2::likelihood => ARP4761::Remote applies to ef1.ServiceOmission;
  **};
end PositionSensor;
```

Figure 47: Property Association to an Error Model Element with an Error Type

Figure 48 shows a property association in the **properties** section of an error model subclause in a system implementation. The system implementation has a subcomponent named *Actuator2*. The containment path identifies an error source *ef2* of type *ServiceOmission* in this subcomponent. The **@** is used instead of a dot to separate the core model portion of the containment path from the error model element portion.

```
EMV2::likelihood => ARP4761::Remote applies to Actuator2@ef2.ServiceOmission;
```

Figure 48: Subcomponent-Specific Property Association

Note that the containment path may identify a component further down the component hierarchy by naming a sequence of subcomponent identifiers.

7.2 Determining a Property Value

For property associations of error model elements, the following inheritance rules apply. We explain them in terms of retrieving a property value for a specific error model element, which we call the *error model element of interest*.

Containment paths of a property association may include a sequence of subcomponent identifiers showing that the property is associated with a component down the component hierarchy. The property association highest in the component hierarchy referencing the same error model element applies to the element. This is the same as for AADL core model property associations.

An error model element of interest is contained in a *component of interest*. This is the component with the error model element of interest or an enclosing component whose property association containment path identifies the component of interest in its subcomponent sequence.

The property value may be retrieved for an error model element of interest without a specific error type:

- First, look for a property association that identifies the error model element in the **properties** section of the error model subclause of the component of interest.
- If not found and the error model element is defined in an error behavior state machine (error event, recover event, repair event, error behavior state, or transition), look for a property association for the element of interest in the state-machine **properties** section.
- If not found and the error model element is defined in the error type library (error type or type set), look for a property association in the **properties** section of the error type library that defines the error type of interest.

The property value may be retrieved for an error model element of interest with a specific error type. These elements are error event, error behavior state, error propagation, and error source, path, and sink:

- First, look for a property association that identifies the error model element and the error type of interest in the **properties** section of the error model subclause of the component of interest.
- If not found, look for a property association that identifies the closest error (super) type that has the type of interest as a subtype, or a type set that contains the error type.
- If not found, look for a property association that identifies the error model element without an error type or type set.

- If not found and the error model element is defined in an error behavior state machine, look for a property association in the state-machine **properties** section. Again, first look for a property association that identifies the error type of interest. If not found, look for a property association of the closest error (super) type that has the type as a subtype, a type set that contains the type, and finally the element itself.
- If not found, look for a property association in the error type library that identifies the error type of interest. If not found, look for a property association of an error type that has the type as a subtype or a type set that contains the type.

7.3 User-Defined Error Model Properties

Users can define properties that apply to error model elements by using the property declaration structure of the AADL core language standard. In the **applies to** clause, the user identifies EMV2 meta-model elements that can have this property association. The meta-model element name is prefixed with the Error Model Annex name *EMV2*, as shown in Figure 49.

```
property set MyErrorProperties is
SelectTypes: type enumeration (midvalue, averagevalue);
SignalSelectionMethod: MyErrorProperties::SelectTypes applies to
({EMV2}**error detection);
end MyErrorProperties;
```

Figure 49: Definition of an EMV2 Property

Users can refer to the following meta-model elements for EMV2: error type, type set, error types (super class of error type and type set), error propagation, error source, error sink, error path, error flow (super class of error source, sink, and path), error behavior state machine, error behavior state, error behavior transition, error event, recover event, repair event, error behavior event (super class of error recover and repair events), outgoing propagation condition, error detection, composite state, connection error source, propagation path, and propagation point.

7.4 Predeclared EMV2 Properties

The properties described here are defined in the property set *EMV2*.

In addition, the property sets *ARP4761* and *MILSTD885* define properties and constants that are specific to these standards (see Section 7.4.11 and *AADL Fault Modeling* [Delange 2014] for further details). These property sets are provided in the OSATE tool.

7.4.1 Occurrence Distribution

The *OccurrenceDistribution* property is associated with error events, states, propagations, and flows. It represents the probability that an internal error event or external error propagation will occur. The property is a record with different fields used to identify a distribution function and capture parameters to characterize the occurrence.

Safety and reliability analyses use different metrics and methods to describe the occurrence probability as a distribution function. The field *Distribution* specifies the distribution function that characterizes the occurrence probability:

- *Fixed* represents a fixed distribution and takes a single parameter *OccurrenceRate*.

- *Poisson* represents the number of occurrences per time interval and takes a single parameter *OccurrenceRate*.
- *Exponential* represents an exponential distribution of occurrences and takes a single parameter *OccurrenceRate*.
- *Normal* or *Gauss* represents a distribution with an explicitly specified *MeanValue* and *StandardDeviation*.
- *Weibull* represents a shaped distribution with a *ShapeParameter* and a *ScaleParameter*.
- *Binominal* represents a discrete distribution with a *SuccessCount*, a *SampleCount*, and a *Probability* parameter.

Figure 50 shows an example of assigning values to the *OccurrenceDistribution* property. The value is a record of values (shown as []) specifying the distribution function and the relevant parameters.

```
EMV2::OccurrenceDistribution => [OccurrenceRate => 1.0e-5;
          Distribution => Poisson;] applies to PowerLoss;
```

Figure 50: Example Specification of Occurrence Distribution

The *OccurrenceDistribution* property can be associated with error propagations, error sources, error paths, error sinks, error events, recover events, repair events, error behavior states, error types, type sets of an error event, error behavior states, and error propagations. If an error type has error subtypes, then *OccurrenceDistribution* represents the occurrence probability for any of the subtypes.

An occurrence distribution property value can be associated with an error source to indicate the probability that an error of any of the specified types will occur. It can also be associated with a specific error type of an error source to indicate the probability that an error of a specific error type will occur.

An occurrence distribution property value can be associated with an error sink or a specific error type of a flow sink. This indicates the probability that an incoming error propagation is not passed on.

An occurrence distribution property value can be associated with an error path or a specific error type of an error path. This indicates the probability that an incoming error propagation is passed through or transformed as an outgoing error propagation.

The occurrence distribution property value of an outgoing error propagation is determined by the probability that the component is the error source and by the probability that an incoming error propagation is passed through or transformed into an outgoing error propagation.

The occurrence distribution property value of an incoming error propagation is determined by the error probability of the outgoing error propagations along all error propagation paths with the error propagation point as the destination.

When applied to error behavior events—including error events, recover events, and repair events—the property specifies a probability according to a specified distribution based on the error behavior events that are expected to occur. This property can be specific to an error type.

When applied to a recover event, the *OccurrenceDistribution* property specifies the probability with which recovery is initiated.

When applied to a repair event, the *OccurrenceDistribution* property specifies when a repair is initiated. The value of this property takes into account the role of a "repair asset," or the resources required to perform such a repair. The actual value may be a computed value (**compute**) to account for the availability of the repair asset.

7.4.2 Exposure Period

The *ExposurePeriod* property specifies the scaling factor that is applied to the *OccurrenceRate* in the *OccurrenceDistribution* property. The units of the occurrence rate and the exposure period must be consistent. For example, the occurrence rate may be specified as failures per hour, and the exposure period is a flight with a maximum duration of 12 hours, resulting in a failure rate per flight.

This property can be attached to error behavior events, error propagations, error flows, error behavior states, error types, and type sets.

7.4.3 Propagation Time Delay

A *PropagationTimeDelay* property indicates the delay in propagating an error as a distribution over a time interval. It can be associated with a connection in the AADL core model or a user-defined propagation point connection declared in EMV2. The *PropagationTimeDelay* property is a record with a *Duration* field that specifies a time range and a *Distribution* field that specifies the distribution function.

7.4.4 Duration Distribution

The *DurationDistribution* property specifies a time range to reflect the duration as a distribution over the time range. This property can be attached to repair or recover events to indicate their duration, once they begin. When applied to a recover event, it represents the duration of the recovery; when applied to a repair event, it represents the duration of the repair. The *DurationDistribution* property is a record with a *Duration* field that specifies a time range and a *Distribution* field that specifies the distribution function.

7.4.5 Transient Failure Ratio

The *TransientFailureRatio* property is used to determine the proportion of a failure transition that results in a transient failure state. It can be attached to error behavior transitions.

7.4.6 Recovery Failure Ratio

The *RecoveryFailureRatio* property is used to determine the proportion of recovery transitions that result in recovery failure. It can be attached to error behavior transitions.

7.4.7 State Kind

The *StateKind* property specifies whether an error behavior state is considered to be a working state or a nonworking state. A component can have multiple error behavior states that are tagged

as working states or nonworking states. This allows analysis tools to assess working and nonworking states.

Note that typing of error behavior states plays a similar role. We may define an *Operational* and a *Nonworking* state, where the *Nonworking* state has an error type set that represents a collection of substates.

7.4.8 Detection Mechanism

The *DetectionMechanism* property allows users to specify the detection mechanism used to detect an error as a string value. Users can also associate the property with an error detection declaration. This property provides traceability to an implementation mechanism in the AADL core model or source code that implements the detection of the specified error type.

7.4.9 Fault Kind

The *FaultKind* property allows the user to specify whether an error source—the occurrence of a fault activation or a propagation—is due to a design fault or an operational fault. Design faults are faults that could be eliminated at design time, but if present result in an error. Operational faults are faults that inherently occur during operation and should be detected and managed during operation.

The *FaultKind* property can be associated with error events, propagations, sources, types, and type sets. This property allows us to reflect design faults in the architecture fault model and address whether the architecture assumes zero design defects or whether it is tolerant of residual design errors.

7.4.10 Persistence

The *Persistence* property allows the user to specify whether an error is permanent, transient, or a singleton. A *permanent* error typically requires a repair action to the component with the fault occurrence. A *transient* error has a limited duration and typically requires a recovery action. In a discrete event system, a transient error may persist over several discrete events; for example, a corrupted message may be sent over multiple periods. A *singleton* error occurs in the context of a single discrete event. For example, a divide-by-zero error may be specific to a computation on a particular input.

The *Persistence* property can be associated with error types, type sets, error behavior states, error events, and error propagations. This property allows us to consider the distinction between persistent and transient fault behavior as observed in error propagations early in development before elaborating the error behavior of individual components.

7.4.11 Severity and Likelihood

The *Severity* property value indicates the severity of a hazard ranging from 1 (high) to 5 (low). MIL-STD 882D suggests these descriptive labels as property constants to be used as property values: *Catastrophic*, *Critical*, *Marginal*, and *Negligible*. ARP4761 defines these descriptive labels as property constants to be used as property values: *Catastrophic*, *Hazardous*, *Major*, *Minor*, and *NoEffect*.

A *Likelihood* property value indicates the likelihood with which a hazard occurs. Likelihood is expressed in terms of levels ranging from 1 (high) to 5 (low) or descriptive labels *A* through *E*. Each level typically has an associated probability of occurrence (*p*) threshold.

MIL-STD 882D suggests the following likelihood levels for probability of occurrence over the life of an item:

- *Frequent*: $p > 10^{-1}$
- *Probable*: $10^{-1} > p > 10^{-2}$
- *Occasional*: $10^{-2} > p > 10^{-3}$
- *Remote*: $10^{-3} > p > 10^{-6}$
- *Improbable*: $p < 10^{-6}$

These labels are defined as property constants and included in the property set *MILSTD882*.

ARP4761 identifies the following descriptive labels, also used by the Joint Aviation Authorities and the FAA, for probability of occurrence per operational hour:

- *Probable*: $p > 10^{-5}$
- *Remote*: $10^{-5} > p > 10^{-7}$
- *ExtremelyRemote*: $10^{-7} < p > 10^{-9}$
- *ExtremelyImprobable*: $p < 10^{-9}$

These labels are defined as property constants and included in the property set *ARP4761*.

These properties can be associated with error types, type sets, error behavior states, error sources, error propagations, and error events. An example is shown in Figure 51.

```
EMV2::Likelihood => ARP4761::ExtremelyRemote applies to Failed;
```

Figure 51: Example Use of a Likelihood Property Association

Note that *Likelihood* and *Severity* are also fields in the *Hazards* property (see Section 7.4.12). The value in the *Hazards* specification overrides the value in the stand-alone property.

7.4.12 Hazards

EMV2 provides three variants of the *Hazards* property defined in the property sets *EMV2*, *ARP4761*, and *MILSTD882*.

The *EMV2::Hazards* property accepts a list of records with the following record fields:

- *Crossreference*: string value for a cross-reference into an external document
- *HazardTitle*: short descriptive phrase for the hazard
- *Description*: string value providing a textual description of the hazard
- *Failure*: system deviation resulting in a failure effect
- *FailureEffect*: description of the effect of a failure (mode)
- *Phases*: a list of string values to identify the operational phase (mode) in which the hazard is relevant
- *Environment*: string value to describe the operational environment in which the hazard is relevant

- *Mishap*: description of event (or series of events) resulting in unintentional death or other unplanned impacts (MIL-STD 882)
- *FailureCondition*: string value description of the event (or series of events) resulting in unintentional death or other unplanned impacts (ARP4761)
- *Risk*: string value to textually describe the potential risk of the hazard
- *Severity*: hazard-specific severity level with the same values as the *Severity* property in Section 7.4.11
- *Likelihood*: hazard-specific likelihood level with the same values as the *Likelihood* property in Section 7.4.11
- *Probability*: a real value for the probability in the range of 0.0 to 1.0
- *TargetSeverity*: the acceptable (target) level of risk expressed as severity
- *TargetLikelihood*: the acceptable (target) level of risk expressed as a likelihood (probability)
- *DevelopmentAssuranceLevel*: level of rigor in development assurance (ARP4761)
- *VerificationMethod*: string value describing the verification method used
- *SafetyReport*: string value describing an analysis or assessment of system hazards
- *Comment*: string value containing additional comments about the hazard

The *MILSTD882::Hazards* property accepts a list of records with the following record fields:

- *Crossreference*: string value for a cross-reference into an external document
- *HazardTitle*: short descriptive phrase for the hazard
- *Description*: string value providing a textual description of the hazard
- *Failure*: system deviation resulting in a failure effect
- *FailureEffect*: description of the effect of a failure (mode)
- *Phases*: a list of string values to identify the operational phase (mode) in which the hazard is relevant
- *Environment*: string value to describe the operational environment in which the hazard is relevant
- *Mishap*: description of event (or series of events) resulting in unintentional death or other unplanned impacts (MIL-STD 882)
- *Risk*: string value to textually describe the potential risk of the hazard
- *SeverityLevel*: hazard-specific severity level with MIL-STD 882-specific labels (see Section 7.4.11)
- *SeverityCategory*: severity level expressed as a value from 1 to 4
- *QualitativeProbability*: MIL-STD 882-specific likelihood labels (see Section 7.4.11)
- *ProbabilityLevel*: labels *A* through *E*
- *QuantitativeProbability*: a real value for the probability in the range of 0.0 to 1.0
- *TargetSeverityLevel*: target severity level using MIL-STD 882-specific labels
- *TargetProbabilityLevel*: the acceptable (target) level of risk expressed by MIL-STD 882-specific likelihood (qualitative probability) labels

- *VerificationMethod*: string value describing the verification method used
- *SafetyReport*: string value describing an analysis or assessment of system hazards
- *Comment*: string value to textually describe additional comments about the hazard

The *ARP4761::Hazards* property accepts a list of records with the following record fields:

- *Crossreference*: string value for a cross-reference into an external document
- *HazardTitle*: short descriptive phrase for the hazard
- *Description*: string value providing a textual description of the hazard
- *Failure*: system deviation resulting in a failure effect
- *FailureEffect*: description of the effect of a failure (mode)
- *Phases*: a list of string values to identify the operational phase (mode) in which the hazard is relevant
- *Environment*: string value to describe the operational environment in which the hazard is relevant
- *Mishap*: description of event (or series of events) resulting in unintentional death or other unplanned impacts (MIL-STD 882)
- *FailureCondition*: string value description of the event (or series of events) resulting in unintentional death or other unplanned impacts (ARP4761)
- *Risk*: string value to textually describe the potential risk of the hazard
- *FailureConditionClassification*: hazard-specific severity level with ARP4761-specific labels (see Section 7.4.11)
- *QualitativeProbability*: ARP4761-specific likelihood labels (see Section 7.4.11)
- *QuantitativeProbability*: a real value for the probability in the range of 0.0 to 1.0
- *QualitativeProbabilityObjective*: target severity level using ARP4761-specific labels
- *QuantitativeProbabilityObjective*: target qualitative probability in the range 0.0 to 1.0
- *DevelopmentAssuranceLevel*: level of rigor in development assurance (ARP4761)
- *VerificationMethod*: string value describing the verification method used
- *SafetyReport*: string value describing an analysis or assessment of system hazards
- *Comment*: string value containing additional comments about the hazard

All variants of the *Hazards* property can be associated with error types, type sets, error behavior states, error sources, error propagations, and error events.

7.4.13 Description

The *Description* property has a string value and allows the user to attach descriptive information with error model elements. This property can be associated with any error model element.

8 Advanced Topics in EMV2

This section discusses several topics for advanced users of EMV2:

- designating inheritance of error model declarations in error model subclauses between component types and component implementations or extensions of component types and implementations
- using error model declarations with feature groups and feature group types
- defining propagation points and paths that are not present in the AADL core model
- using type transformation sets to specify connection error behavior
- specifying a mapping between operational modes in the AADL core model and error behavior states (failure modes) in the error model

For a full discussion of consistency rules between different aspects of the error model specification and the AADL core model, the reader is referred to the companion report [Delange 2014].

8.1 Error Model Subclauses and Inheritance

Error model subclauses can be declared in component types and component implementations. This means that the fault model specifications declared in these subclauses apply to all component instances that reference the classifier with the subclause.

EMV2 follows the inheritance rules of the core model standard. For example, if the error model subclause of a component type declares incoming and outgoing error propagations, then these error propagations are inherited by any component implementation of that component type, even if those component implementations do not have error model subclauses. The same applies to component types that are extensions of other component types and component implementations that are extensions of component implementations.

A component type or implementation that inherits error model elements from others can add new error model elements or override inherited error model elements by declaring them, using the same name, in its error model subclause. By overriding an inherited error model element, we can change its specification. For example, we can change the set of error types for an outgoing error propagation.

An error model element that is added must have a defining name that differs from the inherited elements. For an error model element to be overridden, the declaration must have the same name as the inherited element.

The following override rules apply to error model elements that are inherited, effectively replacing the original declaration:

- **Use types** clause: replaces the set of type libraries, that is, the set of types made accessible without qualification
- **Use type equivalence** clause: replaces the type equivalence mapping reference. Note that this declaration may be inherited from the enclosing component instance.

- **Use behavior** clause: no replacement allowed. The referenced error behavior state machine must be the same
- **Error propagation**: changes the type set associated with the error propagation point named by the error propagation
- **Error containment**: changes the type set associated with the error propagation point named by the error propagation
- **Error source, sink, and path**: changes the flow type, such as changing a sink to a path; changes the source and its type set; changes the target and its type instance; or changes the component's internal cause of an error source (**when** clause). This includes adding a type set or type instance when there was none previously, or specifying the source or target without type set or type instance when there was one previously.
- **Error events**: changes the type set or sets associated with an error event, including addition or removal. Note that error events declared in an error model subclause can override error events declared as part of an error behavior state machine that is made accessible through **use behavior**.
- **Transition**: changes the condition, the originating state or its type set, and the resulting state or its type instance. Note that transitions declared in an error model subclause can override transitions declared as part of an error behavior state machine that is made accessible through **use behavior**.
- **Outgoing propagation conditions**: changes the condition, the originating state or its type set, and the outgoing propagation point or its type instance
- **Error detection**: changes the condition, the originating state or its type set, and the resulting event or its error code
- **Composite error behavior state**: changes the condition, the composite state, or its type instance
- **Mode mapping**: changes the set of modes for the specified error behavior state and type set

Figure 52 shows an example in which a user has added a new incoming event port *command* to the extended device type and the corresponding incoming error propagation to its error model subclause. The extended device inherits the error model subclause from its parent, with the error source on *PositionReading*, and defines an incoming propagation on the command data port.

```
device PositionSensor
features
  PositionReading: out data port;
flows
  f1: flow source PositionReading {
    Latency => 2 ms .. 3 ms;
    };
annex EMV2 {**
  use types ErrorLibrary;
  use behavior ErrorModelLibrary::Simple;
  error propagations
   PositionReading: out propagation  {ServiceOmission};
  flows
   ef1:error source PositionReading {ServiceOmission} when Failed;
```

```
end propagations;
**};
end PositionSensor;

device SecurePositionSensor extends PositionSensor
features
  command: in event data port;
annex EMV2 {**
  use types ErrorLibrary;
  error propagations
    command: in propagation  {ServiceOmission};
  end propagations;
**};
end SecurePositionSensor;
```

Figure 52: Addition of Port and Error Propagation

Figure 53 shows an example in which the error model subclause of the device type extension changes (and overrides) the type set of the outgoing error propagation *PositionReading* to *Service-Omission* and *ValueError*. It also adds the error source *ef2* to specify the additional error type as an error source and the causing fault.

```
device PositionSensorOverride extends PositionSensor
  annex EMV2 {**
  use types ErrorLibrary;
  error propagations
    PositionReading: out propagation {ServiceOmission, ValueError};
  flows
    ef2: error source PositionReading {ValueError} when {BadValue};
  end propagations;
  **};
end PositionSensorOverride;
```

Figure 53: Override of an Error Propagation Specification

8.2 Error Models and Feature Groups

Users can associate error propagations with elements of a feature group type by declaring them in an error model subclause in the feature group type. These error propagations apply to all contexts in which a feature group with this type is declared. In other words, the error types being propagated are independent of the use context.

If users want to associate error propagation types with elements of feature groups in the context of a component with that feature group, they can do so by identifying the feature group and the element of interest in the error propagation declaration of the component error model subclause. See Figure 54 for an example.

```
feature group fgt
features
      inport1: in event port;
      inport2: in event port;
end fgt;

system receiver
features
```

```
  infg: feature group fgt;
annex EMV2 {**
  use types ErrorLibrary;
  error propagations
    infg.inport1: in propagation {ServiceOmission};
    infg.inport2: in propagation {ValueError};
  end propagations;
**};
end receiver;
```

Figure 54: Error Propagations on Feature Group Elements

8.3 User-Defined Propagation Points and Paths

In this section, we introduce the concept of user-defined propagation points and propagation paths that do not rely on elements from the core AADL.

8.3.1 Role of User-Defined Propagation Points and Propagation Paths

Error propagations occur through error propagation points that are declared in the AADL core model, such as the external interfaces (e.g., features) and binding points. For an architecture fault model, sometimes it is desirable to specify that two components can affect each other although no explicit connection has been specified in the AADL core model. For example, two processors may be physically located close to each other but are not connected via any bus. If one processor overheats, it can affect the other processor by corrupting its memory and through other related physical limitations. Figure 55 illustrates this user-defined propagation path.

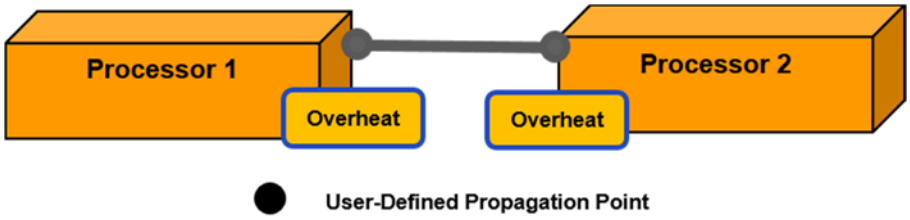

Figure 55: User-Defined Propagation Points and Propagation Path

The user-defined *propagation point* allows users to associate an error propagation point with a component for the purpose of specifying a possible error propagation path that the user knows exists. Such user-defined propagation points can then be referenced in error propagation and error containment declarations to interaction points and binding points in the AADL core model.

A second use of user-defined propagation points is to model observable misbehavior of a system that is not communicated through an interaction point. For example, a failed brake may not be communicated through a feature, but users can observe that the car does not slow down.

These user-defined error propagation points can be connected to other user-defined error propagation points to specify propagation paths. These connections are declared the same way as port connections, namely, through propagation point connection declarations.

8.3.2 Using User-Defined Propagation Points and Propagation Paths

Users can define a propagation point in the **propagation paths** section of an error model subclause. It consists of the keywords **propagation paths**, a number of propagation point declarations, followed by a number of propagation point connection declarations, and ending with **end paths;**.

A propagation point declaration consists of a defining identifier, followed by a colon and the keywords **propagation point**. A propagation point connection declaration consists of a defining identifier, followed by a source propagation point, an arrow, and a destination propagation point. The propagation point may be a subcomponent (identified by the subcomponent name, a dot, and the propagation point name) or the component containing the connection declaration (identified by the propagation point name).

In Figure 56, a user has declared a heat propagation point on the processor *pc* and, within a system containing two instances of these processors, a propagation path.

```
processor pc
annex EMV2{**
propagation paths
   HeatDissipation: propagation point;
end paths;
**};
end pc;

-- within a system containing two instances of processor pc
annex EMV2{**
propagation paths
   HeatPropagation: pc1.HeatDissipation -> pc2.HeatDissipation;
end paths;
**};
```

Figure 56: User-Defined Propagation Point

8.3.3 Observations

Propagation points are nondirectional; they can represent outgoing and incoming error propagations.

User-defined propagation points are useful when the system being modeled includes mechanical components, such as the brakes on a wheel. Using propagation points, we can represent the potential impact a component can have, for example, in the form of a hazard. In effect, we use propagation points to expose a component error behavior state (failure) to its operational environment if no explicit interaction is specified in the AADL core model.

8.4 Error Type Mappings and Equivalence

In this section, we introduce the concept of a type mapping set and its use to define equivalence between independently developed error type libraries. Type mappings help map error type libraries developed by different users in which types and type set names could be inconsistent but semantically equivalent: the mapping rules reunify them.

8.4.1 Role of Type Mapping Sets and Error Type Equivalence

An error type is mapped into a different error type in two contexts: in error flow paths and when defining equivalence between independently developed error type libraries.

Error path declarations indicate how incoming error propagations are mapped into outgoing error propagations of the same or different error types (see Section 4.1). Users may encounter the same mapping from a set of incoming error types to specific outgoing error types in different parts of the architecture fault model specification. Instead of declaring multiple error paths for different error types, users can define reusable sets of mapping rules between source and target error types as *type mapping sets*. The rules of a type mapping set specify how an error type or a type product matching a type set constraint is mapped into another type instance. A type mapping set can be associated with error flow paths.

In large projects, different teams may independently develop architecture fault model specifications for different subsystems. In the process, they may define error type libraries independently. When these subsystem models are virtually integrated, the EMV2 consistency checker compares the error types of outgoing propagations from one subsystem defined in terms of one error type library with those of incoming propagations of the second subsystem defined in terms of a second error type library (see Section 4.4). In doing so, it will not recognize them as matching error types unless an equivalence mapping is provided between the error types defined in each error type library. A *type equivalence* specification that utilizes type mapping sets allows users to indicate an equivalence relationship between the elements of two error type libraries.

8.4.2 Using Type Mapping Sets and Error Type Equivalence

A type mapping set is defined in an error model library. It consists of the keywords **type mappings** and a defining name, a specification of error type libraries being made accessible (**use types**), and a set of *type mapping rules*, ending with the keywords **end mappings**. There may be multiple type sets in a single library. However, the defining name must not conflict with any other type mapping sets, type transformation sets, or error behavior state machines defined in the same error model library. The syntax rules for error type mapping sets are shown in Section 10.3.

A type mapping rule consists of a source type set and a resulting single error type or type product in curly brackets as the target. Type mapping rules for a type mapping set are expected to result in unambiguous mappings; only one mapping rule should apply to a given error type, type set, or type product.

The example type mapping set shown in Figure 57 defines a mapping of all timing and value errors, including their type products, into an item omission. It also shows mapping of omission errors. This is a typical mapping in a fault-tolerant system with fail-silent behavior.

```
type mappings FailSilent
use types ErrorLibrary;
{ValueError, TimingError, ValueError*TimingError } -> {ItemOmission};
{ItemOmission} -> {ItemOmission};
{ServiceOmission} -> {ServiceOmission};
end mappings;
```

Figure 57: Definition of a Type Mapping Set

Type mapping sets can be used to determine the target error type in error path declarations if they are not specified explicitly. This is done by identifying the type mapping set using **use mappings** at the beginning of the error model subclause, as shown in Figure 58. In this case, the type mapping set is expected to cover the error types of the error path source type constraint or referenced source error propagation, and the resulting error types must be contained in the type set of the outgoing error propagation.

```
use mappings MyMappingLib::FailSilent;
```

Figure 58: Use of Type Mapping in an Error Path

Type equivalence between error types and type products in different error type libraries can be specified using type mapping sets. To do this, we need to define the type mapping set. In this case, both error type libraries may be listed in the **use types** clause. If some type names exist in both error type libraries, users must qualify them when referencing the types in a type mapping rule. Figure 59 illustrates a set of equivalence mappings using qualified type references instead of the **use types** clause.

```
type mappings MyLibYourLibMapping
-- MyErrorLib to YourErrLib
{MyValveErrorLib::StuckOpen} -> {YourValveErrorLib::StuckOpenValve};
{MyValveErrorLib::StuckClosed} -> {YourValveErrorLib::StuckClosedValve};
{MyValveErrorLib::ValveLeak} -> {YourValveErrorLib::LeakingValve};
-- YourErrorLib to MyErrLib
{YourValveErrorLib::StuckOpenValve} -> {MyValveErrorLib::StuckOpen};
{YourValveErrorLib::StuckClosedValve} -> {MyValveErrorLib::StuckClosed};
{YourValveErrorLib::LeakingValve} -> {MyValveErrorLib::ValveLeak};
end mappings;
```

Figure 59: Example of Equivalence Mappings

We use the type mapping set to define the type equivalences within an error model subclause. The declaration consists of the keywords **use type equivalence** followed by a reference to the type mapping set, qualified by the error model library containing the type mapping set definition. It applies to all subcomponents recursively, unless a separate type equivalence declaration is associated with the classifier of a subcomponent.

```
annex EMV2 {**
use type equivalence MyMappingLib::MyLibYourLibMapping;
-- other declarations
**};
```

Figure 60: Declaration of Error Type Library Equivalence

8.4.3 Observations

We recommend that users always qualify the types when using type mapping rules to specify equivalence, as illustrated in Figure 59. The qualification clearly documents which types from which library get mapped into types of another library. In particular, it avoids any name issue when using two libraries containing types with the same name.

Type mapping rules specify a mapping in one direction. If the interaction between two subsystems is bidirectional, users will want to define mapping rules in both directions. They can do so in the same type mapping set.

Note that type mappings used on error paths specify how one error type results in a different error type, while type mappings in an equivalence declaration specify that the two error types represent the same kind of error. That is, type mappings in equivalence declarations are aliases of each other.

8.5 Type Transformations and Connection Error Behavior

In this section, we introduce the concept of the type transformation set and its use in defining connection error behavior as well as in determining the target error type in error behavior transitions and outgoing error propagation conditions.

8.5.1 Role of Type Transformation Sets and Connection Error Behavior

In several contexts, a resulting error type is determined from an originating error type and a contributing error type. For example, for a connection we might have a propagated error type from the sender and a contributing error type from the transport mechanism (the bus or virtual bus that the connection is bound to), which may result in a different incoming error type encountered by the receiver. Similarly, for a transition we might have an originating state with a type instance and contributing type instances from the transition condition triggers, such as an error event or incoming error propagation, resulting in a target state with a different type instance. Finally, for error propagation conditions we might have an originating state with a type instance and contributing type instances from the condition triggers, such as an error event or incoming error propagation, resulting in an outgoing propagation with a different type instance.

Type transformation sets allow users to specify reusable sets of transformation rules between source, contributor, and target error types. The rules of a type transformation set specify how an error type or a type product matching a source type set constraint, in combination with an error type or a type product matching a contributor type set constraint, is mapped into a target type instance.

Connections represent the logical flow of information between components as well as an error propagation path (see Section 4.1). The transfer of information along connections is performed by physical buses, virtual buses, or both. The transfer mechanism may contribute error propagations to the data being transferred; for example, data may get corrupted, messages may get lost, or their delivery may get delayed. These errors will result in incoming propagated error types for the receiver that differ from the outgoing propagated types of the sender. This is shown graphically in Figure 61.

Connections themselves can also be error sources. This is the case when the information being sent and the information being received do not match. The mismatch may be in the base type of the data representation, such as a signed versus an unsigned 16-bit integer; in the user-defined application data type, such as a temperature sensor reading versus an air pressure sensor reading; or in the measurement unit used to represent the values, such as temperature in Celsius versus Fahrenheit. If these design errors are not eliminated, they can result in errors or failures during operation.

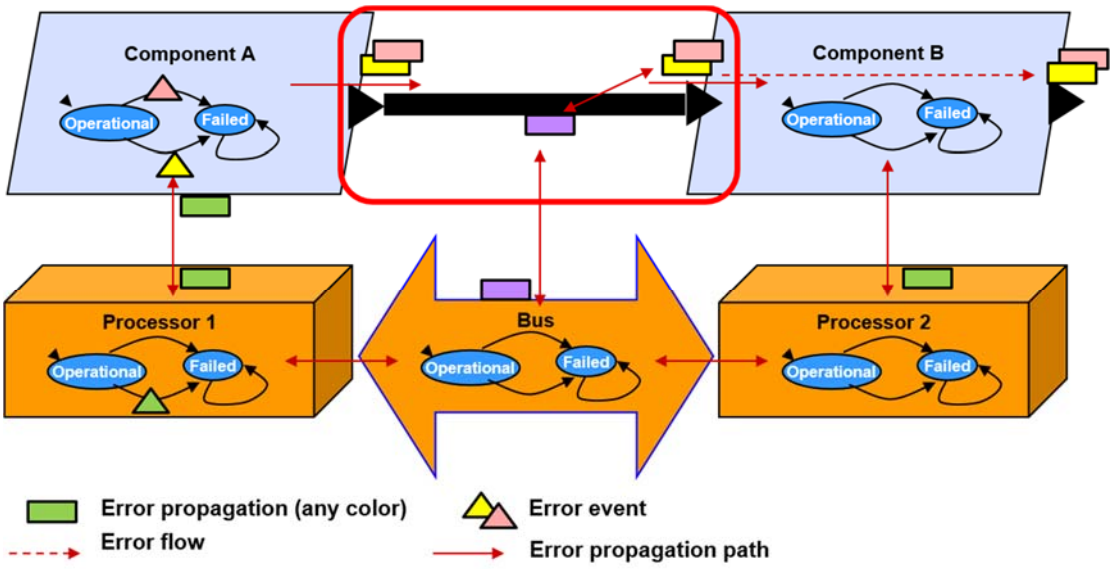

Figure 61: Contributing Error Propagation in Connections

A *connection error behavior* specification allows users to represent the connection itself as an error source, such as when the sender and receiver have an interface mismatch. It also allows users to specify how incoming error propagations from the hardware, such as a network bus, affect the incoming error types of the receiver through the use of a type transformation set.

8.5.2 Using Type Transformation Sets and Connection Error Behavior

A type transformation set is defined in an error model library. It consists of the keywords **type transformations** and a defining name, a specification of error type libraries being made accessible (**use types**), and a set of type transformation rules, ending with the keywords **end transformations**. The defining name must not conflict with any other type mapping sets, type transformation set, or error behavior state machines defined in the same error model library.

A type transformation rule specifies a source type set, a contributor type set, and a resulting error type or type product instance. Type transformation rules in a type transformation set are expected to result in unambiguous transformations. The specific source and contributor error type or type product instances, if matched by more than one rule, must result in the same type instance.

The syntax rules for error type transformation sets and rules are shown in Section 10.3.

The example type transformation set shown in Figure 62 defines transformations for data communication, where the transport mechanism contributes timing, value corruption, message loss, and lack of service. The first two rules show how an error-free source is affected by the contributors *LateDelivery* and *ValueCorruption*. The third rule illustrates how a source error type (*ValueError*) when combined with *LateDelivery* results in a type product. The last two rules show how to specify transformations, whose result is determined only by the contributor or only by the sender.

```
type transformations DataXfer
   use types ErrorLibrary, CommunicationErrors;
   {NoError} -[{LateDelivery}]-> {LateDelivery};
   {NoError} -[{ValueCorruption}]-> {ValueError};
   {ValueError} -[{LateDelivery}]-> {ValueError*LateDelivery};
```

```
   all -[{LostMessage}]-> {NoData};
   {NoData} -[]-> {NoData};
end transformations;
```

Figure 62: Defining a Type Transformation Set

A type transformation set can be associated with transitions, outgoing propagation conditions, and connection error behavior. For transitions and outgoing propagation conditions, this is done by identifying the type transformation set with a **use transformations** declaration (shown in Figure 63) in the error behavior state machine or in the component error behavior declaration. The **use transformations** declaration in the component error behavior overrides that in the error behavior state machine.

Connection error behavior is specified in the **connection error** section of an error model subclause (see Section 2.2). It consists of an optional **use transformations** declaration and a list of zero or more **connection error source** declarations. A **connection error source** declaration consists of a defining identifier, a connection identifier or **all** following the keywords **error source**, and an optional type set as the effect type. As with error source declarations for components, users can optionally specify the original fault and condition (see Section 4.3). **All** indicates that all connections within a component implementation for which the error model subclause applies are potential error sources. The syntax rules for connection error behavior are shown in Section 10.9.

An example declaration is shown in Figure 63. The original fault is indicated by the **when** as an incorrect value and the condition by **if** as a unit mismatch.

```
connection error
  use transformations MyMappingLib::DataXfer;
  UnitMismatch: error source sensorconn {ValueError} when {IncorrectValue} if
“Unit Mismatch”;
end connection;
```

Figure 63: Connection Error Behavior Specification

8.5.3 Observations

When a type transformation set is used on transitions or outgoing propagation conditions, multiple contributor type instances may be encountered. For example, the transition trigger condition may hold if two incoming error propagations have type instances. In this case, the type transformation rules are applied for the first instance first. The result becomes the new source, and the second contributor determines the final result type instance.

EMV2 defines default rules for determining the result error type if no explicit target type instance and no type transformation set are specified to be used. These rules are as follows:

- For connections, the resulting type instance is the type product of the error types from the source and the error type from the binding (the contributor). If both the source and the contributor are the same type, it becomes the result type. If one of them is *NoError*, then the result type is that of the source or the binding.
- For transitions, the resulting type instance is the type product of the error types from the source and the error type from the contributor. If both the source and the contributor are the same type, it becomes the result type. If one of them has no error type associated, then the result type is that of the other.

8.6 Mapping Between Operational Modes and Failure Modes

This section discusses an abstract way of specifying restrictions that error behavior states (from EMV2) can place on operational modes (from the AADL core) of a system.

8.6.1 Role of Mapping Between Error Behavior States and Modes

A component can have operational modes represented in the AADL core model. Error behavior states represent failure modes and can place restrictions on which operational modes can be active and which mode transitions can be initiated when the component is in a particular error behavior state. Mode transitions can be initiated only between modes that are supported in a given error behavior state. When a transition occurs between error behavior states and the new error behavior state does not support the current mode, then a forced mode transition occurs out of the current mode.

Figure 64 shows a GPS system on the left that consists of two sensors and a processing unit. It can operate in high precision mode (*HiP*) using two sensors and low precision mode (*LoP*) using one sensor, or it can be turned off (*Off*). Color coding shows that in *LoP* mode Sensor 1 and Processing are active (green background), in *HiP* mode both sensors and Processing are active (purple outlines), and in *Off* mode none of the components are active. The user of the component can initiate a change in operational mode by command, shown as an incoming event that triggers the appropriate mode transition. As the figure shows, the user can command from *Off* to *LoP* to *HiP* and vice versa.

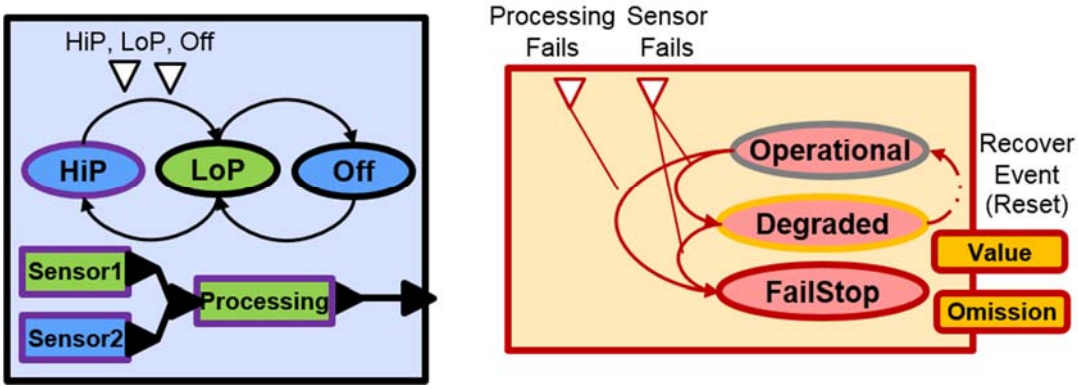

Figure 64: Operational Modes and Failure Modes

The right side of Figure 64 shows the abstracted error model of the GPS system. It shows that a sensor failure event causes a transition into the *Degraded* state. A successful reset attempt of the failed sensor can result in a transition back to *Operational*. A second sensor failure or a processing unit failure causes a transition into *FailStop*. Externally observable effects are error propagations of the types *ValueError* and *ItemOmission* or *ServiceOmission.*

The role of an error behavior state in mode mapping is to specify a set of constraints on modes imposed by error behavior states. The mapping specification superimposes the error behavior states onto the mode state machine, as shown in Figure 65. It shows that when the component is in the *Operational* error behavior state, the user can command the component to switch between all three operational modes. When it is in the *Degraded* error behavior state, only *LoP* and *Off* are

available, and in *FailStop* only *Off* is available. It also shows that if the GPS is in *HiP* mode and a sensor fails—that is, it transitions from *Operational* to *Degraded*—there is a forced transition from *HiP* to *LoP* (shown as a dashed arrow). Similarly, a transition to *FailStop* can force a mode transition.

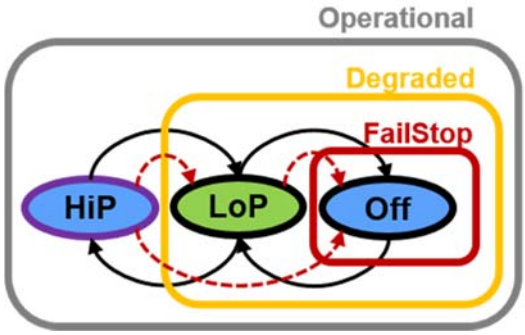

Figure 65: Superimposed Error Behavior States

8.6.2 Using the Mapping Between Error Behavior States and Modes

Mappings between error behavior states and modes are specified in the **mode mappings** section of a component error behavior specification. Each mode mapping specifies which modes are supported by a given error behavior state. Figure 66 specifies a mapping of error behavior states onto modes.

```
mode mappings
  Operational in modes (HiP, LoP, Off);
  Degraded in modes (LoP, Off);
  FailStop in modes (Off);
end component;
```

Figure 66: Example of Mapping Error Behavior States onto Modes

This mode mapping specification is shown in the context of the GPS example model in Figure 67. In this model, we specified the operational modes as part of the GPS component type specification to indicate that users of the GPS are aware of the modes and can control them by command. The commands are represented as event ports that affect mode transitions. The mode transitions are defined to enforce the mode transition ordering from *Off* to *LoP* to *HiP*. Location information is available on the outgoing *LocationPosition* data port as a periodic data stream.

The error model subclause includes an error propagation specification for the *LocationPosition* output and a component error behavior specification. This abstracted error behavior includes error events to represent the failure of a sensor and the processing units within the GPS system with an occurrence probability. A descriptive annotation on the error events indicates what condition, in one of the subcomponents, triggers the event.

```
system GPS
features
-- each user command is represented by a separate event port
  RequestOff : in event port;
  RequestLoP : in event port;
  RequestHiP : in event port;
-- GPS output signal stream
```

```
  LocationPosition : out data port;
-- sensor fail status
  SensorFailStatus : out event data port;
-- input command to reset the GPS
  ResetSensor1 : in event port;
  ResetSensor2 : in event port;
modes
  Off : initial mode;
  LoP : mode; -- low precision operation using one sensor
  HiP : mode; -- hi precision operation
  TurnOffFromLoP : LoP -[ RequestOff ]-> Off;
  TurnOffFromHiP : HiP -[ RequestOff ]-> Off;
  TurnOnLoP : Off -[ RequestLoP ]-> LoP;
  TurnonHiP : Off -[ RequestHiP ]-> HiP;
  SwitchToHiP : LoP -[ RequestHiP ]-> HiP;
  SwitchToLoP : HiP -[ RequestLoP ]-> LoP;
annex EMV2 {**
use types ErrorLibrary;
use behavior GPSEMLibrary::DegradedSM;
error propagations
  LocationPosition : out propagation {ServiceOmission};
end propagations;
component error behavior
events
  Sensor1Failed : error event;
  Sensor2Failed : error event;
  ProcessingFailed: error event;
transitions
  Operational -[Sensor1Failed or Sensor2Failed]-> Degraded;
  Operational -[ProcessingFailed]-> FailStop;
  Degraded -[ Sensor1Failed or Sensor2Failed
              or ProcessingFailed]-> FailStop;
detections
  all -[Sensor1Failed{ServiceOmission}]-> SensorFailStatus!(1);
  all -[Sensor2Failed{ServiceOmission}]-> SensorFailStatus!(2);
mode mappings
  Operational in modes (HiP, LoP, Off);
  Degraded in modes (LoP, Off);
  FailStop in modes (Off);
end component;
properties
 EMV2::Description => "sensor1.sensorData{ServiceOmission}"
  applies to Sensor1Failed;
EMV2::Description => "sensor2.sensorData{ServiceOmission}"
  applies to Sensor2Failed;
EMV2::Description => "processing.locationPosition{ServiceOmission}"
  applies to ProcessingFailed;
EMV2::OccurrenceDistribution =>
  [ProbabilityValue => 0.005; Distribution => Poisson;]
  applies to ProcessingFailed;
EMV2::OccurrenceDistribution =>
  [ProbabilityValue => 0.012; Distribution => Poisson;]
  applies to Sensor1Failed, Sensor2Failed;
**};
end GPS;
```

Figure 67: GPS Operational Modes and Abstracted Error Model

8.6.3 Observations

From the mapping of error behavior states to modes, we derive a combined component behavior state machine, shown in Figure 68. When the component is in *HiP* mode and a *SensorXFailed* error event causes a transition into the *Degraded* error behavior state, it results in a forced mode transition to the *LoP* mode (shown as a dashed arrow). Such a forced mode transition is specified in the AADL core model as an emergency mode transition that is performed immediately, while a planned mode transition is performed when a set of active components reaches a specified synchronization point, typically the hyper-period. While the component is in the *Degraded* error behavior state, the user-initiated mode transitions are limited to those between mode states that are part of the *Degraded* error behavior state. This means that a user cannot transition to *HiP*. Similarly, a second sensor failure (*SensorYFailed*) or a *ProcessingFailed* error event forces a mode transition to the *Off* mode.

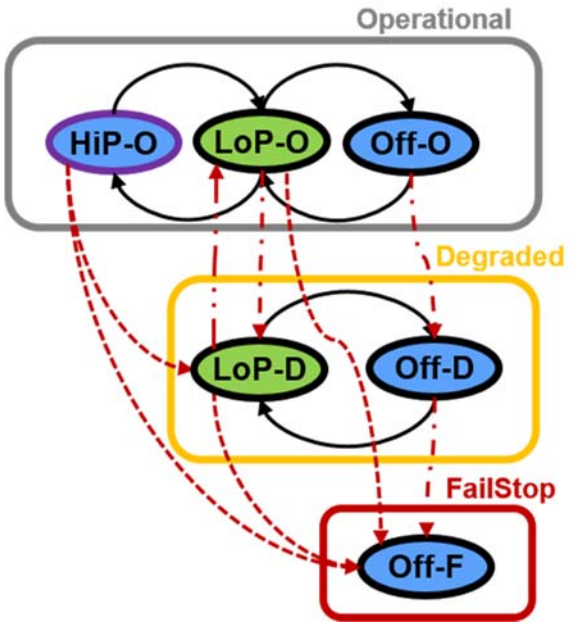

Figure 68: Composite State Diagram of Operational and Failure Mode

In effect, we have a half matrix of mode states and error behavior states. The solid arrows show mode transitions that are initiated by events. Dashed-line arrows show transitions to a different mode that are forced by the occurrence of an error event. Dash-dot arrows indicate transitions between substates of a given mode state; this mapping records the fact that an error behavior transition has occurred without changing the operational mode state. The double-dot-dash arrow indicates a recover event transition from *Degraded LoP* to *Operational LoP*. The resulting state machine can then be analyzed through model checking, such as by exporting it into a model checker like AltaRica [LaBRI 2015].

Figure 69 shows which error behavior changes produce error propagations that can be observed from outside the component. When the component is operating in *HiP* mode, the *Degraded* error behavior state propagates a *value* error, which may be detectable if it is out of range, and the *FailStop* error state propagates a detectable *omission* error. When the component is operating in *LoP* mode, no error is perceived by another component when the component transitions to *Degraded*.

Only a forced transition to *Off* results in a detectable *omission* error propagation. When the component is in *Off* mode, no output is expected; thus, an *omission* propagation is not perceived as an error.

Expected Mode	Operational	Degraded	Fail Stop
HiP	HiP	Force transition to LoP {ValueError}	Force transition to Off {Omission}
LoP	LoP	LoP Ok: LoP <-> Off	Force transition to Off {Omission}
Off	Off	Off Perceived {NoError}	Off Perceived {NoError}

Figure 69: Detectable Error Behavior of a Component

Note that a component may not have an *Off* mode or may not have an explicit mode specification. In this case, a *Failed* error behavior state can be mapped into an **in mode** declaration with an empty list. This declaration indicates that the component is inactive; that is, the enclosing component's mode must not include the failed component.

8.6.4 The Composite GPS Error Model

The GPS system implementation can be annotated with an error model specification

- by specifying the interaction with actual system behavior to represent the recovery attempt by resetting the failing sensor
- by specifying the abstracted error behavior states of GPS in Figure 67 in terms of the error behavior states of its parts (the sensors and the processing unit)

Figure 70 shows the GPS implementation declaration with two sensor instances and the processing instance. Each is declared to be active in a mode-specific manner. In *LoP* mode, the *Processing* component and the *Sensor1* component are active.

```
system implementation GPS.impl
subcomponents
  sensor1: device sensor in modes (LoP,HiP);
  sensor2: device sensor in modes (HiP);
  processing: device GPSprocessing in modes (LoP, HiP);
connections
  sconn1: port sensor1.sensedData -> processing.sensor1input;
  sconn2: port sensor2.sensedData -> processing.sensor2input;
  pconn: port processing.locationPosition -> locationPosition;
  resetconnS1: port ResetSensor1 -> sensor1.reset;
  resetconnS2: port ResetSensor2 -> sensor2.reset;
annex EMV2 {**
use behavior GPSEMLibrary::DegradedSM;
component error behavior
events
  Recovered: recover event;
transitions
  Degraded -[Recovered]-> Operational;
end component;
composite error behavior
states
```

```
  [sensor1.Operational and sensor2.Operational
   and processing.Operational]-> Operational;
  [(sensor1.FailStop and sensor2.Operational and processing.Operational)
   or (sensor1.Operational and sensor2.FailStop and processing.Operational)
  ]-> Degraded;
  [(sensor1.FailStop and sensor2.FailStop)
    or processing.FailStop]-> FailStop;
end composite;
properties
  EMV2::Description => "Sensor Reset Successful" applies to Recovered;
 **};
end GPS.impl;
```

Figure 70: GPS Composite Error Model Specification

The error model subclause for the implementation includes a specification that a sensor failure is detected and triggers a reset of the sensor. If the reset is successful, it results in a recover event, which puts the error behavior back to *Operational*. The error model subclause also includes a composite error behavior state specification that indicates which combination of sensor and processing unit failures results in *Degraded* or *FailStop*.

Note that the composite state declarations reflect and elaborate on the composite state machine of Figure 68. They also imply a particular input processing logic, as discussed in Section 6.1.3.

9 Architecture Fault Model Examples

The following sections illustrate the use of EMV2 for architecture fault modeling with four examples:

1. a dual-redundant flight guidance system with two operational modes, showing the use of component and composite error behavior specifications
2. a multilayered network protocols architecture
3. a dual-channel network system
4. a logical and physical triple-redundant system

9.1 A Dual-Redundant Flight Guidance System with Operational Modes

This example illustrates modeling of a dual-redundant system from three perspectives:

1. error modeling for a system with two operational modes
2. differences in the component error model specification and the composite error model specification
3. the relationship between the error model specification and the actual behavior specification of the redundancy logic

The example is a dual-redundant flight guidance system (FGS) that consists of a flight guidance (FG) component, an autopilot (AP) component, and an actuator subsystem (AC). The FG and AP are dual-redundant pairs. The system is illustrated in Figure 71 and is the same as the one discussed in Section 6.

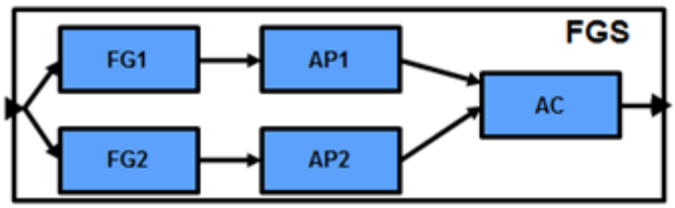

Figure 71: Overview of the FGS

The FGS operates in two modes:

- *critical* mode, requiring both redundant pairs to be in working condition
- *noncritical* mode, when only one pair must be in working condition

We represent the error behavior of the system using three state machines:

- a two-state error behavior state machine that reflects an error-free and a *Failed* state for the FG and AP components
- a three-state error behavior state machine that distinguishes between a noncritical and a critical operational mode for the AC component
- a three-state error behavior state machine that distinguishes between a failure in the noncritical mode and a failure in the critical mode for the FGS

9.1.1 Error Behavior of FGS Components

We use a simple error behavior state machine with an *Operational* working state and a *Failed* nonworking state. An error event named *Failure* triggers a transition from *Operational* to *Failed*. The definition of this state machine is shown in Figure 72.

```
error behavior Simple
events
  Failure : error event;
states
  Operational : initial state;
  Failed : state;
transitions
  FailureTransition : Operational -[ Failure ]-> Failed;
end behavior ;
```

Figure 72: Reusable Two-State Error Behavior

This error behavior state machine is associated with the FG component, the AP component, and the AC component. The *Failed* state reflects the fact that the component itself failed. For the FG, we add a condition for the outgoing propagation in terms of the component *Failed* state. For the AP, we specify an outgoing propagation condition in terms of the *Failed* state and in terms of an incoming *NoValue* propagation when the component is error free.

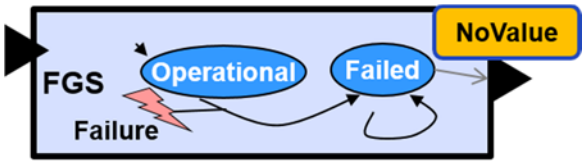

Figure 73: Two-State Error Behavior of the FG Subsystem

In the FG component, this state machine is utilized to characterize the failure of FG itself (see **use behavior** declarations in Figure 74). The state machine is augmented in FG with a specification that the *Failed* state results in an error propagation of type *NoValue*. For AP, we specify that both an AP failure and an incoming propagation of *NoValue*—reflecting the failure of FG—result in an out propagation of *NoValue*.

```
system FG
features
   InPort: in data port;
   OutPort: out data port;
annex emv2 {**
use types ErrorModelLibrary;
use behavior ErrorModelLibrary::Simple;
error propagations
  OutPort: out propagation {NoValue};
end propagations;
component error behavior
propagations
  Failed-[]->OutPort{NoValue};
end component;
**};
end FG;

system AP
```

```
features
   InPort: in data port;
   OutPort: out data port;
annex emv2 {**
use types ErrorModelLibrary;
use behavior ErrorModelLibrary::Simple;
error propagations
   InPort: in propagation {NoValue};
   OutPort: out propagation {NoValue};
end propagations;
component error behavior
propagations
   Failed      -[]->OutPort(NoValue);
   Operational -[InPort{NoValue}]-> OutPort{NoValue};
end component;
**};
end AP;
```

Figure 74: Two-State Error Model for FG and AP

For the AC component, the component error behavior specification is slightly more complex. Here we use a three-state error model: *Failed* to represent the failure state of AC, and *OperationalNonCritical* and *OperationalCritical* to represent two working states that mirror the two operational modes. The three-state error behavior model is shown in Figure 75.

```
error behavior ThreeErrorStates
events
   SingleNoValueEvent: error event;
   DualNoValueEvent: error event;
   SelfFailure: error event;
states
   Operational: initial state;
   OperationalNonCritical: state;
   OperationalCritical: state;
   Failed: state;
transitions
   SingleNoValueTransition: Operational-[SingleNoValueEvent]->OperationalCriti-
cal;
   DualNoValueTransition: Operational -[DualNoValueEvent]-> OperationalNonCriti-
cal;
   FailureTransition:Operational-[SelfFailure]->Failed;
   end behavior;
```

Figure 75: Reusable Three-State Error Behavior Model

Figure 76 shows the AC system declaration with the three-state error behavior model. Within the error model declarations, the two working states are synchronized with the operational modes *NonCritical* and *Critical* through a **mode mapping** declaration. The two working error behavior states allow us to specify outgoing propagation conditions that are sensitive to the operational modes. The operational modes are declared as inherited from the FGS using the **requires modes** declaration, since the FGS manages operational mode transitions based on user commands (see Section 9.1.2).

We specify the outgoing propagation *NoValue* in the state *OperationalNonCritical* if both inputs to AC receive *NoValue*. *NoValue* is propagated in the state *OperationalCritical* if one of the inputs to AC is missing. The failure of AC itself, when AC is in the error behavior state *Failed*, results in a *NoValue* out propagation.

```
system AC
features
   FromAP1Port: in data port;
   FromAP2Port: in data port;
   OutPort: out data port;
requires modes
   Critical: mode;
   NonCritical: initial mode;
annex emv2 {**
use types ErrorModelLibrary;
use behavior ErrorModelLibrary::ThreeErrorStates;
error propagations
  FromAP1Port: in propagation {NoValue};
  FromAP2Port: in propagation {NoValue};
  OutPort: out propagation {NoValue};
end propagations;
component error behavior
propagations
  Failed-[]->OutPort{NoValue};
  OperationalNonCritical -[FromAP1Port{NoValue} and FromAP2Port{NoValue}]->
    OutPort{NoValue};
  OperationalCritical -[FromAP1Port{NoValue} or FromAP2Port{NoValue}]->
    OutPort{NoValue};
mode mappings
  OperationalNonCritical in modes (noncritical);
  OperationalCritical in modes (critical);
 end component;
properties
  EMV2::StateKind => Working applies to OperationalNonCritical,
    OperationalCritical;
  EMV2::StateKind => NonWorking applies to Failed;
**};
end AC;
```

Figure 76: Three-State Error Model for AC

9.1.2 Composite Error Behavior of the FGS

We define the composite error behavior for the FGS as a three-state behavior reflecting failure in non-critical mode (*NonCriticalModeFailure*) and critical mode (*CriticalModeFailure*) as two separate error behavior states. Figure 77 shows the *GPSErrorModelLibrary* package with the three-state error behavior declarations.

```
package GPSErrorModelLibrary
public

annex EMV2{**
error behavior ThreeState
events
  failure: error event;
  NonCriticalModeFail: error event;
  CriticalModeFail: error event;
states
   Operational: initial state;
   NonCriticalModeFailure: state;
   CriticalModeFailure: state;
transitions
   CriticalFailureTransition: Operational-[CriticalModeFail]-> CriticalModeFail-
ure;
   NonCriticalFailureTransition: Operational -[NonCriticalModeFail]->
```

```
       NonCriticalModeFailureb
end behavior;
**};
end GPSErrorModelLibrary;
```

Figure 77: The GPSErrorModelLibrary *Package*

FGS is considered to be in the state *CriticalModeFailure* when it cannot continue to operate in *Critical* operational mode. That is, AC has failed or at least one component in one redundant FG–AP pair has failed; thus, the system has one failure. FGS is considered to be in the state *NonCriticalModeFailure* when it cannot continue to operate in *NonCritical* operational mode. That is, a system failure occurs when the AC has failed or at least one element in both redundant FG–AP pairs has failed. Figure 78 shows these conditions expressed as composite error behavior state declarations in the error model subclause of the FGS implementation.

```
system FGS
features
   inport: in data port;
   outport: out data port;
   goCritical: in event port;
   goNonCritical: in event port;
   externalPower: requires bus access PowerSupply;
modes
   Critical: mode;
   NonCritical: initial mode;
   transCritical: NonCritical -[goCritical]-> Critical;
   transNonCritical: Critical -[goNonCritical]-> NonCritical;
annex emv2 {**
use types ErrorModelLibrary;
use behavior ErrorModelLibrary::ThreeState;
error propagations
   inport: in propagation {NoValue};
   outport: out propagation {NoValue};
   externalPower: in propagation {NoPower};
end propagations;
**};
end FGS;

system implementation FGS.threestate
subcomponents
   AP1: system AP;
   AP2: system AP;
   FG1: system FG;
   FG2: system FG;
   AC: system AC;
connections
   FGStoFG1: port inport -> FG1.inport;
   FGStoFG2: port inport -> FG2.inport;
   FG1toAP1: port FG1.outport -> AP1.inport;
   FG2toAP2: port FG1.outport -> AP2.inport;
   AP1toAC: port AP1.outport -> AC.FromAP1Port;
   AP2toAC: port AP2.outport -> AC.FromAP2Port;
   ACtoFGS: port AC.outport -> outport;
annex emv2 {**
   use types ErrorModelLibrary;
   use behavior GPSErrorModelLibrary::ThreeState;
composite error behavior
states
    [AP1.Operational and AP2.Operational and FG1.Operational
```

```
      and FG2.Operational and AC.OperationalCritical
      and in externalPower{NoError}]->Operational;

    [((AP1.Operational and AP2.Operational) or
     (FG1.Operational and FG2.Operational)) and AC.OperationalNonCritical
    and in externalPower{NoError} ]->Operational;

    [AC.Failed or (1 ormore (FG1.Failed, AP1.Failed) and
       1 ormore (FG2.Failed, AP2.Failed))
       or in externalPower{NoPower}]-> NonCriticalModeFailure;

    [1 ormore (AC.Failed, AC.OperationalNonCritical, AP1.Failed,
               FG1.Failed, AP1.Failed, FG2.Failed)
     or in externalPower{NoPower}]->CriticalModeFailure;
end composite;
**};
end FGS.threestate;
```

Figure 78: Three-State Error Model for FGS

The composite state logic can be interpreted using a reliability block diagram (RBD) to determine the probability of an FGS failure in critical or noncritical mode. This probability is derived from the probability of each subcomponent being in its *Failed* state. For more on reliability analysis by RBD or Markov chain analysis, see *AADL Fault Modeling and Analysis* [Delange 2014].

When operating in critical mode, the FGS has a replicated FG–AP pair of components. This allows us to consider errors of type *ReplicationError*, including asymmetric value, timing, and omission errors. If we consider one of the replicated FG–AP sequences to have an omission error (*NoValue*), and AC is operating in critical mode, then it effectively encounters an *Asymmetric-Omission* error.

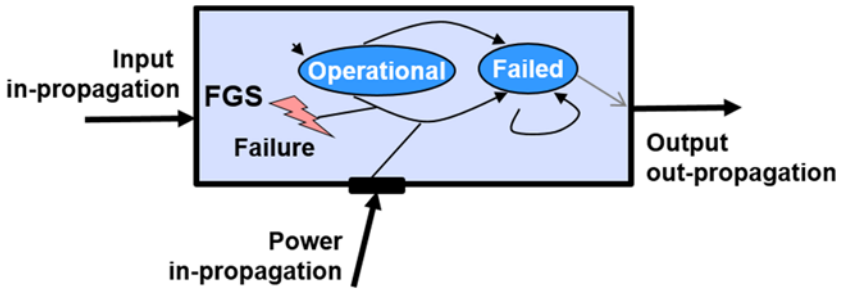

Figure 79: Impact of Electrical Power Loss

The availability of FGS may be affected by the resources it needs to operate. In Figure 79, we show that FGS depends on electrical power. Any incoming *ServiceOmission* of power causes FGS to enter a *Failed* state. A single power supply affects all subcomponents, and FGS cannot operate without power. This is specified in the composite state condition and will be reflected in the RDB or Markov chain analysis.

9.2 Error Propagations Through Networks and Protocols

In this section, we illustrate how to model an error propagation through a multilayered protocol stack. Figure 80 shows a sender communicating with a receiver. The sender and receiver are bound to different processors, and the connection is bound to a cyclic redundancy check (CRC)

protocol, which in turn is bound to a delivery protocol (DP), which finally is bound to network hardware. The sender may occasionally send data with value errors.

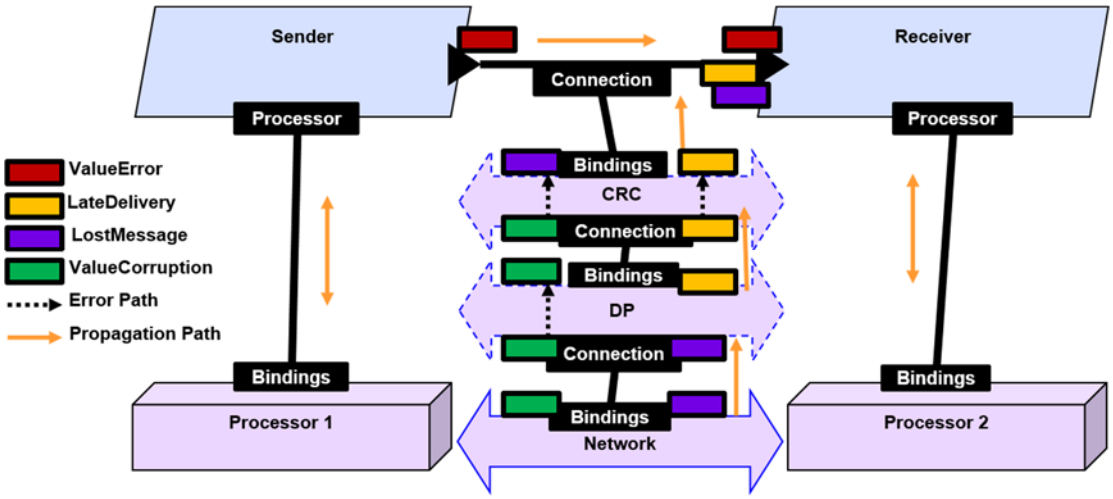

Figure 80: Error Propagation in a Multilayered Network

For the connection, we use type transformations to indicate how error propagations from the network and the protocols affect the error types being received. The virtual buses, which are used to represent protocols, have two binding points: one to represent their binding to the next lower level and one to represent the binding of higher level protocols or the connections to them.

The AADL model for this system, including binding and type transformation specification, is shown in Figure 81.

```
system implementation MySystem.basic
subcomponents
  Sender: system Sender;
  Receiver: system Receiver;
  PC1: processor PC;
  PC2: processor PC;
  Network: bus Network;
  DP: virtual bus DP;
  CRC: virtual bus CRC;
connections
  dataxfer: port Sender.output -> Receiver.input;
properties
  Actual_Processor_Binding => (reference(PC1)) applies to Sender;
  Actual_Processor_Binding => (reference(PC2)) applies to Receiver;
  Actual_Connection_Binding => (reference(CRC)) applies to dataxfer;
  Actual_Connection_Binding => (reference(DP)) applies to CRC;
  Actual_Connection_Binding => (reference(Network)) applies to DP;
annex EMV2 {**
connection error
  use transformations CommunicationErrors::DataXfer;
end connection;
**};
end MySystem.basic;
```

Figure 81: Network and Protocol Binding Specification

The processors are interconnected by network hardware (*Network*) with a basic transport protocol. This network hardware is the potential source of dropped packets (*LostMessage*) and data corruption (*ValueCorruption*). The *Network* fault model specification is shown Figure 82.

```
bus Network
annex EMV2 {**
use types ErrorLibrary, CommunicationErrors;
error propagations
  bindings: out propagation {LostMessage, ValueCorruption};
flows
  HWFault: error source bindings {LostMessage, ValueCorruption};
end propagations;
**};
end Network;
```

Figure 82: Network Fault Model Specification

On top of the network hardware, we have a DP that ensures delivery of messages by retransmitting them if necessary. This is indicated by specifying DP as an error sink for incoming *LostMessage* errors from the network. *ValueCorruption* errors are passed through to the next layer. Due to transmit retries, *DP* is a potential source for timing errors in the form of *LateDelivery*. The *DP* fault model specification is shown in Figure 83.

```
virtual bus DP
annex EMV2 {**
use types ErrorLibrary, CommunicationErrors;
error propagations
  bindings: out propagation {LateDelivery, ValueCorruption};
  connection: in propagation {LostMessage, ValueCorruption};
flows
  RetryTiming: error source bindings {LateDelivery};
  MaskLoss: error sink connection {LostMessage};
  PassCorruption: error path connection {ValueCorruption} -> bindings;
end propagations;
**};
end DP;
```

Figure 83: Fault Model Specification for the DP

The next protocol layer is a CRC protocol; its role is to detect data corruption and map it into a *LostMessage*. The CRC passes through any late delivery to the application. The *CRC* fault model specification is shown in Figure 84.

```
virtual bus CRC
annex EMV2 {**
use types ErrorLibrary, CommunicationErrors;
error propagations
  bindings: out propagation {LateDelivery, LostMessage};
  connection: in propagation {LateDelivery, ValueCorruption};
flows
  DetectCorruption: error path connection {ValueCorruption}
     -> bindings {LostMessage};
  PassTiming: error path connection {LateDelivery} -> bindings;
end propagations;
**};
end CRC;
```

Figure 84: Fault Model Specification for the CRC

Initially the receiver specification may have included only those error types propagated by the sender. The unhandled fault checker will then identify the timing errors and item omissions as not expected by the receiver, and the appropriate corrective action can be taken.

9.3 An Error Propagation and Mitigation Contract for a Dual-Channel Network

Some networks, such as the SAFEbus™ avionics network by Honeywell, support dual-channel application operation. In these networks, a pair of sending hosts sends data to their respective receivers, such as the FG–AP pairs in the FGS example of the previous section. We will use EMV2 to annotate SAFEbus with a fault model that treats the SAFEbus as a black-box abstraction with a fault model. The objective of this fault model is to capture the assumptions and guarantees that SAFEbus makes to the application channel pair.

SAFEbus uses two hardware channels to deal with hardware failures in a physical channel by detecting asymmetric failure and data corruption and mapping them into symmetric omission to the receiving host pair. Figure 85 shows the specification of SAFEbus as an AADL bus annotated with an EMV2 specification. In this abstraction, it consists of two virtual channels, one for each application channel connection. Note that in an implementation model, each virtual bus is then mapped as a replicated communication across two lower level virtual buses that are bound to two of the four hardware channels. The lines marked with "2x" in Figure 85 indicate a pair of bus access connections coming from the application host hardware, which can be represented as a feature group. The black arrow entering the black rectangle in Figure 85 shows the binding point from the application connection binding. In this example, we will specify both incoming and outgoing error propagations through the binding point.

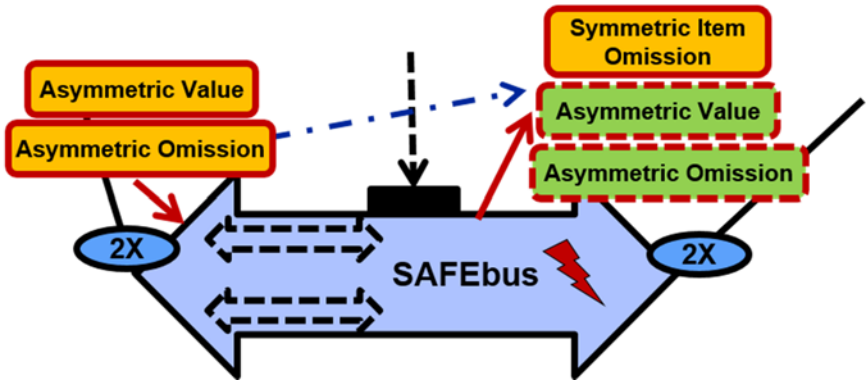

Figure 85: Errors Related to the SAFEbus

Note that we do not show error propagations coming from lower layers of the SAFEbus protocol of hardware here, but we describe such a representation in the network example of the previous section.

The error model for SAFEbus specifies that SAFEbus can be an error source, but that any error introduced by SAFEbus manifests itself only as *SymmetricItemOmission* for the receiver pair. The SAFEbus protocol will detect error sources whose originating faults are asymmetric value corruption or loss and map them into item omissions on both channels. We also specify that we do not expect SAFEbus to propagate asymmetric value errors or asymmetric item omissions (shown as

green boxes with dashed outlines in Figure 85). This is due to the SAFEbus internal voting logic, which detects asymmetric errors introduced by a SAFEbus internal failure. As a side effect, SAFEbus will act as a gatekeeper for asymmetric value and asymmetric omission errors on behalf of the application channel pair. In other words, SAFEbus maps asymmetric value and omission error propagations from the sending application pair into symmetric item omission (shown as a dot-dash arrow and specified as an error path in Figure 85).

This SAFEbus fault model interface specification allows us to validate whether an application uses SAFEbus in a way that does not result in mismatched assumptions. We will illustrate three such assumptions.

The first assumption is that SAFEbus will always be used at full dual-channel operation. For example, the FGS as a dual-redundant system may have a critical operational mode and a noncritical operational mode. In critical mode, it operates both channels, while in noncritical mode it may operate a single channel with the second channel in standby to free resources for other services. However, when FGS operates in noncritical mode, the effect of the SAFEbus gatekeeping function is that it will not deliver any data. Because operating the FG–AP pairs as standby appears to SAFEbus as an inconsistent omission error, the SAFEbus fault-tolerance logic maps this mode into a symmetric omission, as illustrated in Figure 86. The EMV2 consistency checker detects this mode as a mismatch in contract and assumption (see also Section 4.4).

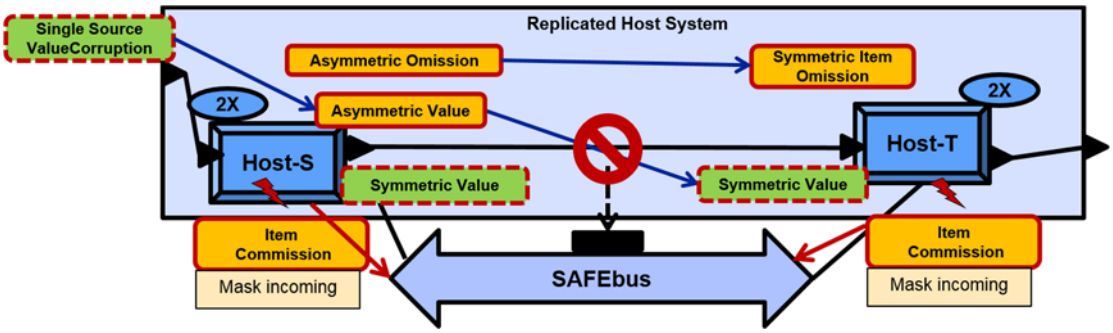

Figure 86: Error Propagation Related to Components Using the SAFEbus

The second assumption is that SAFEbus includes functionality to deal with a babbling host. A host may send messages when it is not supposed to. The architecture fault model expresses this error type as *ItemCommission*. After we enhance the SAFEbus specification by indicating that it is the error sink for such incoming error types, the EMV2 consistency checker can determine that a babbling host will not impact the rest of the system.

The third assumption is that although SAFEbus acts as a gatekeeper for asymmetric value and asymmetric omission errors, it will not deal with incoming symmetric value or symmetric omission errors, but passes them through. Thus, if the receiving host pair does not expect incoming symmetric value errors (shown as green/dashed incoming error propagation in Figure 86), we must ensure that the sending pair does not send symmetric value errors. We can accomplish this by performing a backward-tracing impact analysis. It must ensure that no single source of data corruption error will replicate to the two sending hosts.

9.4 A Reconfigurable Triple-Redundant System

This section presents an example high-level specification of a reconfigurable system with redundant subsystems. This system begins operation with three redundant copies of a subsystem operating and a fourth spare subsystem that is initially powered down. The working subsystems do a cross-vote to detect and isolate failures. When a failure of a subsystem is detected, the fourth subsystem can be powered up and used to replace it. The overall system is considered operational as long as at least two of the powered-up subsystems are error free and no more than one subsystem has failed unsafely (is active and produces errors that must be masked).

For this example, we assume that a subsystem includes self-checking algorithms, so it can detect and fail-stop many but not all of its own failures. The error model declares two types of faults, one that can be successfully detected and still permit a subsystem to fail-stop (*No_Data*) and another that produces a subsystem failure that either cannot be self-checked or that prohibits a subsystem from fail-stopping (*Bad_Data*).

We also assume that a subsystem can be dynamically powered up or powered down by a mode switch during operation. Fault rates (e.g., as determined by MIL-HDBK-217F) are significantly different for powered versus unpowered equipment. Mode-specific property values permit different occurrence rates to be specified depending on whether a subsystem is powered or unpowered. The powered and unpowered modes do not have to be represented as separate error behavior states, as we did in the version of this example included in the Error Model Annex, Version 1.

A subsystem is initially in the *Operational* error behavior state. A failed system can exist in one of two error behavior states, *Fail_Stopped* if self-checking has worked or *Fail_Unknown* if it has not. A system in the *Fail_Unknown* state is assumed to exhibit arbitrarily bad behavior; for example, it remains active and sends misleading data, and it cannot be powered down at a mode switch. Figure 87 shows the three-state error behavior model.

The error event *Self_Checked_Fault* represents a fault occurrence detected by self-checking, causing a transition to *Fail_Stopped*. The error event *Uncovered_Fault* represents an undetected fault occurrence, causing a transition to *Fail_Unknown*. It is assumed that once a subsystem is in the *Fail_Stopped* state, it will stay in *Fail_Stopped* upon subsequent error events. It is assumed that when the subsystem is in the *Fail_Unknown* state, it transitions to the *Fail_Stopped* state if a subsequent *Self_Checked_Fault* event occurs.

```
package TripleErrorModel

public
annex EMV2 {**
error behavior Example
use types TripleErrorTypes;
events
-- both events will have mode-specific occurrence values for
-- powered,unpowered
 Self_Checked_Fault: error event;
 Uncovered_Fault: error event;
states
 Operational: initial state;
 Fail_Stopped: state;
 Fail_Unknown: state;
transitions
```

```
  SelfFail: Operational -[Self_Checked_Fault]-> Fail_Stopped;
  UFailSFault: Fail_Unknown -[Self_Checked_Fault]-> Fail_Stopped;
  SFailSFault: Fail_Stopped -[Self_Checked_Fault]-> same state;
  UncoveredFail: Operational -[Uncovered_Fault]-> Fail_Unknown;
  UFailUFault: Fail_Unknown -[Uncovered_Fault]-> same state;
  SFailUFault: Fail_Stopped -[Uncovered_Fault]-> same state;
end behavior;
**};
end TripleErrorModel;
```

Figure 87: Triple-Redundant Error Behavior State Machine

Next, we specify error propagation behavior of a component as perceived by other components using the **error propagations** section. The propagations *No_Data* and *Bad_Data* represent different classes of errors propagated from a failed subsystem. *No_Data* means it is visible to the receiver that the failed subsystem has fail-stopped. *Bad_Data* means the receiver gets erroneous and misleading data from the failed subsystem, and the receiver must determine the correct value using other redundant sources of data and detect the failure. The error flow declarations indicate that the component can be the source of propagations, pass the propagations on, or act as an error sink by detecting and masking propagations. The conditions under which the component will mask (error sink) or pass on (error path) an incoming propagation will be specified in the component error behavior specification. The fault model with error paths and voting logic is shown in Figure 88.

```
package LargeExample
public
with TripleErrorModel, TripleErrorTypes,EMV2;

system Subsystem
features
  A: in data port;
  B: in data port;
  O: out data port;
  A_Failed: out event port;
  B_Failed: out event port;
modes
   Powered: initial mode;
   Unpowered:  mode;
annex EMV2 {**
use types TripleErrorTypes;
error propagations
  A: in propagation {Data_Fault};
  B: in propagation {Data_Fault};
  O: out propagation {Data_Fault};
flows
  es: error source O{Data_Fault};
  ep1: error path A -> O{Bad_Data};
  ep2: error path B-> O{Bad_Data};
  es2: error sink A;
  es3: error sink B;
end propagations;
component error behavior
propagations
  Fail_Unknown -[ ]-> O{Bad_Data};
  Fail_Stopped -[ ]-> O{No_Data};
-- these conditions represent the Guard_In declaration in the EM V1 Example
  Operational -[B{Data_Fault} or A{Data_Fault}]-> O{noerror};
  Operational -[A{Data_Fault} and B{Data_Fault}]-> O{Bad_Data};
detections
```

```
  Operational -[1 ormore(A{No_Data},A{Bad_Data})]-> A_Failed!;
  Operational -[1 ormore(B{No_Data},B{Bad_Data})]-> B_Failed!;
end component;
properties
  EMV2::OccurrenceDistribution =>
    [ ProbabilityValue => 1.0e-4; Distribution => poisson; ]
    in modes (Powered),
    [ ProbabilityValue => 1.0e-5; Distribution =>  poisson; ]
    in modes (Unpowered) applies to Self_Checked_Fault;
  EMV2::OccurrenceDistribution =>
    [ ProbabilityValue => 0.05e-4; Distribution =>  poisson; ]
    in modes (Powered),
    [ ProbabilityValue => 0.05e-5; Distribution =>  poisson; ]
    in modes (Unpowered) applies to Uncovered_Fault;
  EMV2::OccurrenceDistribution =>
    [ ProbabilityValue => 1.0e-4; Distribution => fixed; ]
    applies to O.Bad_Data;
**};
end Subsystem;
```

Figure 88: Subsystem Fault Model with Error Paths and Voting Logic

The **component error behavior** section specifies the details of the component error behavior of the subsystem. The conditions for outgoing error propagations are declared to propagate a particular error type on an outgoing error propagation point (port) based on a given error behavior state. While a subsystem is in the *Fail_Stopped* state, it sporadically propagates *No_Data* errors. While a subsystem is in the *Fail_Unknown* state, it sporadically propagates *Bad_Data* errors according to the *OccurrenceDistribution* property specified for *Bad_Data* while remaining in the *Fail_Unknown* state.

The error behavior declaration also specifies a default set of Poisson occurrence rates for each kind of error event. Uncovered faults are 20 times less likely than covered ones, with self-testing coverage at approximately 95%. Faults in unpowered subsystems occur at 1/10 the rate of faults in powered ones.

An active subsystem produces one output stream (declared as an *out* port O) and receives the outputs of the two other active subsystems (declared as *in* ports A and B). Each subsystem votes its own result with that of its sibling subsystems to detect and isolate failures. When a failure in a sibling subsystem is detected and isolated, an event will be raised. This is specified by the **detections** clause.

The **detections** clause of the subsystem places requirements on the voting protocols that are to be realized in the final physical system. The behavior of the actual physical subsystem must be verified against this specification. These specifications of nominal behavior are written assuming that this subsystem is error free. Typically they only name error behavior states of other components. An erroneous subcomponent by definition no longer obeys its nominal specification; in particular, an erroneous subsystem may not exhibit the specified voting behaviors. The voting behaviors of other error-free subsystems will determine how the overall system responds to errors in a given subsystem.

The voting (comparison) protocol must be able to detect *No_Data*, such as by the absence of a fresh value on the incoming port, and *Bad_Data*, such as by comparison of incoming values. A subsystem votes by comparing three values, its two inputs and its own internal result. Among the

three values being voted, if two agree and one persistently disagrees, then the one that persistently disagrees is assumed to have failed.

If a subsystem believes it has failed, then it does not raise an event; it simply attempts to fail-stop and relies on the other two subsystems to detect this. Thus, each subsystem has only two outgoing events, one for each of the other subsystems that it monitors, and does not have a third outgoing event to signal a self-check failure.

An actual physical subsystem would likely make a decision only after seeing persistent disagreement—for example, *n* miscompares out of *m* consecutive values—in order to be robust to transient faults and errors. Transient fault events and transient error behavior states could be included in an error model, but this example models only permanent faults and failures.

In this model, any incoming error propagation from the two other subsystems is conservatively assumed to place a subcomponent in an unknown error behavior state. This is declared by the outgoing propagation condition, resulting in the outgoing error type *Bad_Data*. For example, if a sibling subsystem has visibly fail-stopped, and the internal result disagrees with the value received from the only other operating subsystem, then the model assumes worst-case behavior. An actual subsystem might attempt to continue operating or might attempt to fail-stop, but these are both modeled in this specification as unsafe failures of that subsystem.

The difficulty in modeling this protocol lies in deciding what assumptions to make about its behavior when one of the subsystems is in a *Failed_Unknown* error behavior state and propagating *Bad_Data* errors. This specification assumes that an error-free subsystem will detect *Bad_Data*, if at least one of the other subsystems is error free. In an actual system, a Byzantine error may occur, which is to say that a subsystem in a *Failed_Unknown* state may send error-free data to one sibling subsystem (which detects no errors) and erroneous data to the second sibling subsystem (which detects an error). The two error-free subsystems do not have a consensus on whether a failure has occurred. We will return to this issue in a few paragraphs.

The reconfigurable system, as shown in Figure 89, consists of four subsystems and four modes of operation. In each mode of operation, three of the four subsystems are active. The mode names in this example have the form *Sxyz*, where *x*, *y*, and *z* denote the subsystems that are active in that mode.

```
system Dependable_System
end Dependable_System;

system implementation Dependable_System.Notional
subcomponents
      S1: system Subsystem in modes (S123, S124, S134);
      S2: system Subsystem in modes (S123, S234, S124);
      S3: system Subsystem in modes (S123, S134, S234);
      S4: system Subsystem in modes (S124, S134, S234);
connections
      c1: port S2.O -> S1.A in modes (S123);
      c2: port S3.O -> S1.B in modes (S123);
      c3: port S1.O -> S2.A in modes (S123);
      c4: port S3.O -> S2.B in modes (S123);
      c5: port S1.O -> S3.A in modes (S123);
      c6: port S2.O -> S3.B in modes (S123);
```

```
      c11: port S2.O -> S4.A in modes (S234);
      c12: port S3.O -> S4.B in modes (S234);
      c13: port S4.O -> S2.A in modes (S234);
      c14: port S3.O -> S2.B in modes (S234);
      c15: port S4.O -> S3.A in modes (S234);
      c16: port S2.O -> S3.B in modes (S234);

      c21: port S4.O -> S1.A in modes (S134);
      c22: port S3.O -> S1.B in modes (S134);
      c23: port S1.O -> S4.A in modes (S134);
      c24: port S3.O -> S4.B in modes (S134);
      c25: port S1.O -> S3.A in modes (S134);
      c26: port S4.O -> S3.B in modes (S134);

      c31: port S2.O -> S1.A in modes (S124);
      c32: port S4.O -> S1.B in modes (S124);
      c33: port S1.O -> S2.A in modes (S124);
      c34: port S4.O -> S2.B in modes (S124);
      c35: port S1.O -> S4.A in modes (S124);
      c36: port S2.O -> S4.B in modes (S124);
modes
      S123: initial mode;
      S234: mode; S134: mode; S124: mode;
      T1_4: S123 -[ S2.A_Failed, S3.A_Failed ]-> S234;
      T2_4: S123 -[ S1.A_Failed, S3.B_Failed ]-> S134;
      T3_4: S123 -[ S1.B_Failed, S2.B_Failed ]-> S124;
annex EMV2 {**
 use behavior TripleErrorModel::Example;
 composite error behavior
 states
      [2 ormore(S1.Operational,S2.Operational,S3.Operational,S4.Operational) and
      1 orless(S1.Fail_Unknown, S2.Fail_Unknown, S3.Fail_Unknown, S4.Fail_Un-
known)]-> Operational;
-- original model did not have a specification for  Fail_Stopped state
   [3 ormore(S1.Fail_Stopped,S2.Fail_Stopped,S3.Fail_Stopped,S4.Fail_Stopped)
      ]-> Fail_Stopped;
-- we do not have others as catchall
      [ others]-> Fail_Unknown;
      end composite;
**};
annex behavior_specification {**
transitions
      T1_4: S123 -[ S2.A_Failed and S3.A_Failed ]-> S234;
      T2_4: S123 -[ S1.A_Failed and S3.B_Failed ]-> S134;
      T3_4: S123 -[ S1.B_Failed and S2.B_Failed ]-> S124;
**} ;
end Dependable_System.Notional;
```

Figure 89: Reconfigurable Triple-Redundant System Model

In each mode of operation, the three subsystems that are supposed to be powered up and active are cross-connected. Each subsystem listens to both of the other active subsystems on different input ports, and its output is connected to both of the other active subsystems.

Subsystems that detect failures also raise events. These events are routed to a monitoring component and trigger a mode transition. An actual physical system might implement this, for example,

by having each processor vote on the incoming error connections and enter a mode switch consensus protocol only when the two events specified for the current mode of operation occur within a maximum time separation.

The mode transition specification shows that a false transition might occur if one of the two subsystems has failed in an unknown state and falsely raises an event. This event can take an error-free subsystem offline and leave a failed subsystem online. It does not necessarily lead to immediate failure in the initial mode of operation, since the newly powered-up subsystem together with the remaining error-free subsystem can mask errors from the failed subsystem. However, it makes suboptimal use of redundant resources by wasting an error-free subsystem. The system is now only one fault away from unsafe system failure.

To give an example illustrating the previous two paragraphs, if both S1 and S3 are *Operational* and request a mode transition, then that transition will occur. If both S1 and S3 have failed babbling (are in a *Fail_Unknown* error behavior state and propagate *Bad_Data*) and request a mode transition, then that transition will occur. The former case is a correct and desired behavior; the latter case is an incorrect and undesired behavior that nevertheless occurs because both S1 and S3 have simultaneously failed babbling and simultaneously erroneously requested a mode change.

As noted earlier, it is also possible that a mode transition would be lost if a Byzantine error occurs and only one of the error-free subsystems detects that subsystem failure. This event would be taken into consideration if the modeling and analysis tool considered all possible subsets of consequence in error propagation transitions, in particular the case where a *Bad_Data* error propagated to one *Operational* subsystem but not the other. Again, this does not necessarily lead to immediate failure because the two error-free subsystems can mask the errors of the failed one, but it makes suboptimal use of the available resources by failing to bring the available spare online. The system is now only one fault away from unsafe system failure.

The error behavior state of the system as a whole is defined as a function of the error behavior states of the subsystems. The system is considered to be operating acceptably when at least two subsystems are active and error free and no more than one subsystem has failed unsafely.

10 EMV2 Syntax Rules

The syntax rules use the following symbols:

- [] to represent an optional construct
- | to separate alternatives
- ()* to represent zero or more elements
- ()+ to represent one or more elements
- () to group syntax elements
- -- to indicate an explanatory annotation

10.1 Error Model Library

```
-- placed directly in a package
error_model_library ::=
 annex EMV2 none;
 |
 annex EMV2 {**
   [ error_type_library_definition ]
   ( error_behavior_state_machine_definition )*
   ( type_mapping_set_definition )*
   ( type_transformation_set_definition )*
 **};
```

10.2 Error Type Library, Error Type, Type Set, and Alias

```
error_type_library_definition ::=
 error types
   [ use types error_type_library ( , error_type_library )* ; ]
   [ extends error_type_library with ]
   ( error_type_definition | error_type_alias_definition
     type_set_definition | type_set_alias_definition
   )+
   [ properties ( contained_property_association )+ ]
end types;

error_type_library ::=
  package_name     -- package containing the error type library

error_type_definition ::=
  defining_identifier : type [ extends error_type ] ;

error_type_alias_definition ::=
  defining_identifier renames type error_type;

error_type ::=
  [error_type_library::]error_type_identifier |
  [error_type_library::]error_type_alias_identifier

type_set_definition ::=
  defining_identifier : type set { type_set_element ( , type_set_element )* } ;

error_type_set_alias_definition ::=
```

```
  defining_identifier renames type set type_set ;

type_set ::=
  [error_type_library::]type_set_identifier |
  [error_type_library::]type_set_alias_identifier

type_set_element ::=
  error_type | type_set | type_product

type_product ::=
  error_type_reference ( * error_type_reference )+

type_set_constructor ::=
  { type_set_element ( , type_set_element )* }

type_set_constraint ::=
  type_set_constructor | {NoError}

type_instance ::=
  { error_type | type_product }
```

10.3 Type Mapping Set and Type Transformation Set

```
type_mapping_set ::=
  type mappings defining_identifier
  [ use types error_type_library ( , error_type_library )* ; ]
   ( type_mapping_rule )+
  end mappings;

type_mapping_rule ::=
    source_type_set -> target_type_instance ;

type_transformation_set ::=
  type transformations defining_identifier
  [ use types error_type_library ( , error_type_library )* ; ]
   ( type_transformation_rule )+
  end transformations;

type_transformation_rule ::=
(source_type_set_constraint | all)
  -[[contributor_type_set_constraint]]-> target_type_instance;

type_mapping_set ::=
    [package_name::]type_mapping_set_identifier

type_transformation_set ::=
    [package_name::]type_transformation_set_identifier
```

10.4 Error Behavior State Machine

```
error_behavior_state_machine_definition ::=
  error behavior defining_state_machine_identifier
  [ use types error_type_library ( , error_type_library )* ; ]
  [ use transformations type_transformation_set ; ]
  [ events (error_event | recover_event | repair_event )+ ]
  [ states ( error_state )+ ]
  [ transitions (transition | branching_transition )+ ]
  [ properties ( contained_property_association )+ ]
  end behavior ;
```

```
error_behavior_state_machine ::=
    [package_name::]error_behavior_state_machine_identifier

error_event ::=
  defining_identifier : error event [ type_set ]
    [ if “error_event_condition” ];

recover_event ::=
  defining_identifier : recover event
     [ when initiator_reference ( , initiator_reference )* ] ;

initiator_reference ::=
  mode_transition_reference | event_port_reference | self_event_reference

repair_event ::=
  defining_identifier : repair event
     [ when initiator_reference ( , initiator_reference )* ] ;

error_state ::=
  defining_identifier : [ initial ] state [ type_set ] ;

transition ::=
    [ defining_identifier : ]
    transition_source -[ error_condition ]-> transition_target ;

branching_transition ::=
    [ defining_identifier : ]
    source_state -[ error_condition ]-> (transition_branches);

source_state ::=
    all | (error_state_identifier  [ type_set ]

transition_target ::=
    error_state_identifier [ type_instance ]
    | same state

transition_branches ::=
    transition_target with branch_probability
     ( , transition_target with branch_probability )*

branch_probability ::=
  fixed_probability_value | others

fixed_probability_value ::=
    real_literal | [ package_identifier::]real_property_constant_identifier

error_condition ::=
  condition_trigger | ( error_condition )
  | error_condition and error_condition
  | error_condition or error_condition
  | numeric_literal ormore ( condition_trigger ( , condition_trigger )+ )
  | numeric_literal orless ( condition_trigger ( , condition_trigger )+ )

condition_trigger ::=
    error_behavior_event_identifier [ type_set ]
    | [in] incoming_error_propagation_point [ type_set_constraint ]
    | subcomponent_identifier . outgoing_error_propagation_point
          [ type_set_constraint ]
```

10.5 Error Model Subclause

```
-- placed in a component type, component implementation, or feature group type
error_model_subclause ::=
 annex EMV2 none;
 |
 annex EMV2 {**
    [ use types error_type_library ( , error_type_library )* ; ]
    [ use type equivalence type_mapping_set_reference ; ]
    [ use behavior error_behavior_state_machine_reference ; ]
    [ error_propagation_section ]
    [ component_error_behavior_section ]
    [ composite_error_behavior_section ]
    [ connection_error_behavior_section ]
    [ user_defined_point_path_section ]
    [ properties ( contained_property_association )+ ]
**} [ in_modes ];
```

10.6 Error Propagation Section

```
error_propagation_section ::=
    error propagations
     [ error_propagation | error_containment ]+
     [ flows (error_source | error_sink | error_path )+ ]
    end propagations;

error_propagation ::=
    error_propagation_point : ( in | out ) propagation type_set ;

error_containment ::=
    error_propagation_point : not ( in | out ) propagation type_set ;

error_propagation_point ::=
  feature_reference | binding_reference | propagation_point_identifier

feature_reference ::=
    ( feature_group_identifier . )*  feature_identifier
    | access

binding_reference ::=
    processor | memory | connection | binding | bindings

error_source ::=
  defining_identifier : error source
    ( outgoing_error_propagation_point | all ) [ effect_type_set ]
    [ when fault_source ] [ if fault_condition ] ;

error_sink ::=
  defining_identifier : error sink
    ( incoming_error_propagation_point | all ) [ type_set ] ;

error_path ::=
  defining_identifier : error path
      ( incoming_error_propagation_point | all  ) [ type_set ] ->
      ( outgoing_error_propagation_point | all )
       [ target_type_instance  ;

fault_source ::=
```

```
    error_behavior_state [type_set ] | type_set | “description”

fault_condition ::= string_literal;

Note: fault_condition will be a constraint expression once the Constraint Annex has
become available.
```

10.7 Component Error Behavior Section

```
component_error_behavior_section ::=
    component error behavior
      [ use transformations type_transformation_set ; ]
      [ events (error_event | recover_event | repair_event )+ ]
      [ transitions (transition | branching_transition )+ ]
      [ propagations ( outgoing_propagation )+ ]
      [ detections ( error_detection )+ ]
      [ mode mappings ( error_state_to_mode_mapping )+ ]
    end component;

outgoing_propagation ::=
  [ defining_identifier : ]
   ( source_state | all ) -[ [ error_condition ] ]-> propagation_target ;

propagation_target ::=
    ( error_propagation_point | all ) [ target_type_instance | {noerror} ]

error_detection ::=
  [ defining_identifier : ]
  ( source_state | all ) -[ [ error_condition ] ]-> error_detection_effect ;

error_detection_effect ::=
    ( port_identifier | internal_event_reference ) ! [ ( error_code_value ) ]

internal_event_reference ::=
    event_or_event_data_source_identifier

error_code_value ::=
    integer_literal | enumeration_identifier | property_constant_term

error_state_to_mode_mapping ::=
  error_state_identifier [ type_instance ] in modes (mode_name ( , mode_name )*);
```

10.8 Composite Error Behavior Section

```
composite_error_behavior ::=
  composite error behavior
      states { composite_error_state }+
  end composite;

composite_error_state ::=
  [ defining_identifier : ]
  [ ( subcomponent_state_expression | others ) ]-> composite_state_identifier
      [ target_type_instance ] ;

composite_state_expression ::=
  state_element | ( composite_state_expression )
  | composite_state_expression and composite_state_expression
```

```
  | composite_state_expression or composite_state_expression
  | numeric_literal ormore ( state_element ( , state_element )+ )
  | numeric_literal orless ( state_element ( , state_element )+ )

state_element ::=
    subcomponent_error_state [ type_set ]
    | in error_propagation_point [type_set_constraint ]

subcomponent_error_state ::=
  ( subcomponent_identifier . )+ error_state_identifier
```

10.9 Connection Error Behavior Section

```
connections_error_behavior_section ::=
connection error
  [ use transformations type_transformation_set ; ]
  ( connection_error_source )*
end connection;

connection_error_source ::=
  defining_identifier :
    error source ( connection_identifier | all )
   [ effect_type_set ]
   [ when ( fault_source_type_set | "description" ) ] ]
   [ if fault_condition ] ;;
```

10.10 User-Defined Propagation Point and Path

```
user_defined_point_path_section ::=
propagation paths
   ( propagation_point )*
( connections
   ( propagation_point_connection )* )?
end paths ;

propagation_point ::=
  defining_identifier : propagation point ;

propagation_point_connection ::=
    defining_identifier :
     source_user_defined_error_propagation_point ->
       target_user_defined_error_propagation_point ;

user_defined_propagation_point ::=
    { subcomponent_identifier . }+ propagation_point_identifier
```

References

URLs are valid as of the publication date of this document.

[Bondavalli 1990]
Bondavalli, A. & Simoncini, L. *Failure Classification with Respect to Detection, Specification and Design for Dependability (First Year Report)*. Esprit Project N°3092. PDCS: Predictably Dependable Computing Systems. 1990.

[Delange 2014]
Delange, Julien; Gluch, David; Feiler, Peter; & Hudak, John. *AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment*. CMU/SEI-2014-TR-020. Software Engineering Institute, Carnegie Mellon University. 2014. http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=311884

[FAA 2010]
Federal Aviation Administration. *System Safety Handbook*. FAA. Dec 2010. http://www.faa.gov/regulations_policies/handbooks_manuals/aviation/risk_management/ss_handbook/

[Feiler 2010]
Feiler, Peter H. Challenges in Validating Safety-Critical Embedded Systems. *SAE International Journal of Aerospace*. Volume 3. Issue 1. 2010. 109–116. https://www.sae.org/technical/papers/2009-01-3284

[Feiler 2012]
Feiler, P. H. & Gluch, D. *Model-Based Engineering with AADL*. SEI Series on Software Engineering. Addison-Wesley. 2012.

[Feiler 2013]
Feiler, P. H.; Goodenough, J. B.; Gurfinkel, A.; Weinstock, C. B.; & Wrage, L. *Four Pillars for Improving the Quality of Safety-Critical Software-Reliant Systems*. White paper. Software Engineering Institute, Carnegie Mellon University. 2013. http://www.sei.cmu.edu/library/abstracts/whitepapers/FourPillarsSWReliability.cfm

[Hecht 2011]
Hecht, Myron; Lam, Alexander; & Vogl, Chris. A Tool Set for Integrated Software and Hardware Dependability Analysis Using the Architecture Analysis and Design Language (AADL) and Error Model Annex. Pages 361–366. In *Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems*. April 2011.

[ISO 2005]
International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers. *ISO/IEC/IEEE 15408:2005 Common Criteria for Information Technology Security Evaluation*. Version 3.1, Revision 4. 2005.

[ISO 2010]
International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers. *ISO/IEC/IEEE 24765:2010 Systems and Software Engineering—Vocabulary*. December 2010. http://www.iso.org/iso/catalogue_detail.htm?csnumber=50518

[LaBRI 2015]
LaBRI. AltaRica Checker, Version 1.5. 2015.

[Leveson 2012]
Leveson, Nancy G. *Engineering a Safer World: System Thinking Applied to Safety*. MIT Press. 2012.

[NIST 2002]
National Institute of Standards and Technology. *The Economic Impacts of Inadequate Infrastructure for Software Testing.* NIST Planning Report 02-3. NIST. May 2002.

[Paige 2009]
Paige, Richard F.; Rose, Louis M.; Ge, Xiaocheng; Kolovos, Dimitrios S.; & Brooke, Phillip J. FPTC: Automated Safety Analysis for Domain-Specific Languages. In *Models in Software Engineering*. Chaudron, Michel R. [editor]. *Lecture Notes in Computer Science*. Volume 5421. Pages 229–242. Springer-Verlag. 2009.

[Powell 1992]
Powell, D. Failure Mode Assumptions and Assumption Coverage. Pages 386–395. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*. IEEE Computer Society Press. 1992.

[Redman 2010]
Redman, David; Ward, Donald; Chilenski, John; & Pollari, Greg. Virtual Integration for Improved System Design. Pages 57–64. *First Analytic Virtual Integration of Cyber-Physical Systems Workshop*. In conjunction with the IEEE Real-Time Systems Symposium (RTSS), San Diego, CA. Nov. 2010. http://www.andrew.cmu.edu/user/dionisio/avicps2010-proceedings/proceedings.pdf

[Rushby 1981]
Rushby, John. Design and Verification of Secure Systems. Pages 12–21. In *Proceedings of the 8th ACM Symposium on Operating System Principles*. ACM. 1981.

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

[SAE 2006]
SAE International. *Architecture Analysis & Design Language (AADL) Annex Volume 1: AADL Meta Model & XML Interchange Format Annex, Error Model Annex, Programming Language Annex*. Standards Document AS5506/1. SAE. 2006.

[SAE 2012]
SAE International. *Architecture Analysis & Design Language (AADL)*. Standards Document AS5506B. Sep. 2012. Originally issued in 2004.

[SAE 2015]
SAE International. *Architecture Analysis & Design Language (AADL) Annex Volume 1: Annex A: ARINC653 Annex, Annex C: Code Generation Annex, Annex E: Error Model Annex*. Standards Document AS5506/1A. Sep. 2015. Originally issued in 2006.

[Walter 2003]
Walter, C. & Suri, N. The Customizable Fault/Error Model for Dependable Distributed Systems. *Theoretical Computer Science*. Volume 290. 2003. 1223–1251.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	June 2016	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Architecture Fault Modeling and Analysis with the Error Model Annex, Version 2	FA8721-05-C-0003

6. AUTHOR(S)

Peter Feiler , John Hudak, Julien Delange, and David P. Gluch

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2016-TR-009

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Safety-critical software-reliant systems must manage component failures and conditions of anomalous interaction among components as hazards that affect a system's safety, reliability, and security so the potential effects of hazards on system operation are reduced to an acceptable risk. Standards and recommended practices for safety-critical systems outline methods for analysis, but security-related practices are typically addressed through separate guidance. This report provides guidance on using the Error Model Annex, Version 2 (EMV2), notation for architecture fault modeling and analysis, which supports automated safety, reliability, and security analyses from the same annotated architecture model to ensure consistency across analysis results. EMV2 augments architecture models expressed in the Architecture Analysis & Design Language with fault information to characterize anomalous conditions. The report introduces concepts for architecture fault modeling of systems in an operational environment at three levels of abstraction. In addition, EMV2 introduces the concept of error types to characterize exceptional conditions and their propagation. Finally, EMV2 allows users to specify which system components are expected to detect, report, and manage anomalous conditions and their propagation and to reflect the effects of recovery and repair actions as error behavior states. The report includes several example models.

14. SUBJECT TERMS	15. NUMBER OF PAGES
AADL, architecture modeling, fault modeling, safety analysis, safety-critical systems, security, software-reliant systems	123

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. Z39-18
298-102